Définition et réalisation d'un outil de vérification formelle de programmes LUSTRE

Christophe Ratel

THESE

présentée par

RATEL Christophe

pour obtenir le titre de

DOCTEUR

de

L'UNIVERSITE JOSEPH FOURIER - GRENOBLE 1

(arrêtés ministériels du 5 Juillet 1984 et du 23 Novembre 1988)

Spécialité : Informatique

Définition et Réalisation d'un Outil de Vérification Formelle de Programmes LUSTRE: le Système LESAR

Date de Soutenance : 8 Juillet 1992

Composition du Jury :

J. Voiron	Président
P. Cousot	Rapporteurs
R. De Simone	
N. Halbwachs	Examinateurs
F. Ouabdesselam	
J.L. Bergerand	

Thèse préparée au sein du Laboratoire de Génie Informatique de Grenoble

Remerciements

Cette thèse a été réalisée de Septembre 1989 à Juillet 1992 dans le cadre d'une convention CIFRE entre l'Association Nationale pour la Recherche Technique, la société Merlin Gerin et le Laboratoire de Génie Informatique de l'IMAG à Grenoble.

Je tiens à remercier:

Monsieur Jacques Voiron, Professeur à l'Université Joseph Fourier de Grenoble, pour m'avoir fait l'honneur de présider le jury de cette thèse ;

Messieurs Patrick Cousot, Professeur à l'Ecole Normale Supérieure de Paris, et Robert De Simone, Directeur de Recherches à l'Institut National pour la Recherche en Informatique et en Automatique, pour avoir accepté de juger ce travail ;

Monsieur Nicolas Halbwachs, Directeur de Recherches au Centre National de Recherche Scientifique, et chef de l'équipe LUSTRE, dont l'intérêt, la disponibilité et la compétence m'ont apporté toute la lumière sur le travail relaté dans cet ouvrage ;

Messieurs Farid Ouabdesselam, Maître de Conférences à l'Université Joseph Fourier, et Jean-Louis Bergerand, pour leur sympathie et leur gentillesse à mon égard ;

Jean-Christophe Madre et Olivier Coudert du Centre de Recherche de la société Bull, dont les grandes connaissances dans le domaine de la vérification symbolique m'ont permis de débuter ce travail, et qui ont largement collaboré aux recherches que j'ai effectuées grâce à leurs remarques nombreuses et constructives ;

Pascal Raymond et Frédéric Rocheteau, membres de l'équipe LUSTRE, dont les outils ont été nécessaires à ceux que j'ai développés, et avec lesquels j'ai eu grand plaisir à travailler ;

Xavier Nicollin, pour l'énorme coup de main qu'il m'a apporté durant la préparation de ma soutenance ;

Christian Dubois et Philippe Turlier, dont l'amitié et le soutien moral m'ont été d'un grand réconfort, surtout pendant la rédaction de cette thèse. "Si toutefois" cela est nécessaire, il faudra longtemps se rappeller qu'ils ont aussi été les brillants organisateurs de mon pot de thèse ;

Je remercie également tous les membres du projet SPECTRE, avec lesquels j'ai beaucoup aimé travailler, même si je m'apprête à les quitter... Qu'ils se rassurent (?) tout de même, ce n'est qu'un au revoir ;

Enfin, je remercie "chaleureusement" le mauvais temps qui a régné durant ce printemps 1992, pour ne pas avoir rendu moralement encore plus difficile la rédaction de cette thèse...

Je dédie cet ouvrage:

A mon Père et à ma Mère, sans lesquels RIEN de tout cela n'aurait été possible...

A Sonia, qui m'a toujours apporté son soutien moral au cours de ces trois années de travail.

A ma famille, dont l'intérêt pour les études que j'ai entreprises ne s'est jamais démenti.

Sommaire

Introduction

Systèmes Temps Réel

Dans le monde industriel, il est systématique de mécaniser le travail humain dans tous les domaines où celui-ci est *automatisable:* les robots automobiles ou les systèmes de pilotage automatique d'avions en témoignent de manière évidente. Les objectifs de cette mécanisation se situent à divers niveaux : gain de productivité, baisse des coûts de production, augmentation de la fiabilité de fonctionnement.

L'avènement de l'informatique n'a fait qu'accentuer cette évolution: les ordinateurs remplacent ou assistent l'opérateur humain dans des tâches de plus en plus complexes. Ce principe atteint ses limites dans les systèmes informatiques dits *temps réel,* auxquels sont confiées des tâches cruciales: gestion de centrales nucléaires, contrôle de procédés chimiques, régulation du trafic dans les moyens de transport, réseaux de télécommunications... Plus que tout autre, ces systèmes doivent *impérativement* respecter des contraintes de fonctionnement extrêmement sévères. En effet, leur fiabilité à ce niveau met pratiquement toujours en jeu la sauvegarde d'intérêts économiques considérables, la protection de matériels onéreux et la sécurité de vies humaines. D'autre part, leur caractère "temps réel" traduit le fait qu'ils interagissent souvent avec un environnement physique, dont le comportement est par nature *ininterruptible* et *irréversible.* Ils doivent de ce fait aussi satisfaire des contraintes temporelles strictes, consistant à pouvoir réagir "suffisamment rapidement" à tout événement en provenance de leur environnement.

Le domaine d'application des systèmes temps réel soulève donc de manière très aiguë le problème de leur fiabilité. Celle-ci englobe à la fois les aspects *matériels* (fourniture d'électricité, résistance physique des circuits à la température ou aux chocs, durée de vie des composants), et les aspects *fonctionnels* (fonctionnalités effectivement réalisées, temps de réponse). Notre intérêt se limitera ici à ces derniers.

Généralement, les aspects fonctionnels des systèmes temps réel touchent au domaine du logiciel. En pratique, les fonctionnalités de ces systèmes sont mises en œuvre par des programmes, qui conditionnent par conséquent leur fiabilité. Il est donc impératif d'atteindre avec ces programmes le plus haut niveau de *qualité logicielle* [Ger] possible. En pratique, cela consiste à définir des méthodes et des outils de développement de logiciel permettant aux programmeurs de réaliser des programmes sans erreurs ("zéro défauts"). Ces outils et méthodes peuvent être regroupés en deux domaines, selon le stade auquel ils interviennent:

- En "amont" de la réalisation des programmes: il s'agit de définir des règles (normes, méthodologies, conventions) et des outils (langages de programmation, compilateurs, ateliers logiciels) dont l'utilisation permet de réaliser des programmes corrects.

- En "aval" de la réalisation des programmes: il s'agit de définir des méthodes de validation permettant de contrôler que les programmes réalisés mettent bien en œuvre les fonctionnalités spécifiées (relecture, simulation, tests).

En ce qui nous concerne, nos efforts se sont uniquement concentrés sur la définition et la conception d'outils permettant, aussi bien en amont qu'en aval, d'obtenir des programmes respectant les spécifications pour lesquelles ils ont été réalisés.

Réalisation des systèmes

La réalisation d'un système nécessite la *modélisation* de ses fonctionnalités et des interactions avec son environnement. La programmation consiste alors à traduire *ce que* doit faire le système dans la réalité (exprimé en langue naturelle dans un cahier des charges) en *comment* il doit le faire sur le modèle qui lui a été associé, le résultat étant un programme (écrit dans un langage de programmation) . Cette phase est à l'évidence très délicate, donc source potentielle d'erreurs. Il est alors évident que la qualité des programmes réalisés peut être largement augmentée grâce à l'utilisation d'un langage adapté à la programmation des systèmes, permettant notamment de minimiser les risques d'erreurs. Un tel langage doit répondre aux critères suivants:

- Il doit permettre une description naturelle des systèmes à réaliser. En particulier, le mode de description employé doit être proche de celui utilisé traditionnellement dans le domaine d'application considéré.
- Il doit être assez simple pour minimiser les erreurs d'interprétation des programmes.
- Il doit être basé sur des concepts formels, afin de permettre la réalisation d'outils de développement fiables (compilateurs, vérificateurs).

Vérification des systèmes

En aval de la réalisation des systèmes, les programmes doivent être vérifiés afin de contrôler qu'ils respectent les spécifications pour lesquelles ils ont été développés. Le principe classique de la vérification est la simulation: le système est exécuté dans son environnement, et l'observation de son comportement par rapport à ce dernier permet de décider s'il se comporte conformément à ses spécifications.

Le principe de la simulation consiste donc à confronter trois *entités*:

- **Le système**, qui est censé mettre en œuvre les fonctionnalités requises.
- **l'environnement**, dont le comportement conditionne le comportement du système.
- **Les spécifications**, qui permettent de décider si le système se comporte correctement par rapport à ce qui est attendu.

Une première approche de la simulation est purement expérimentale et statistique: le système est exécuté un *certain nombre* de fois correspondant à des configurations différentes de son

environnement réel, chaque exécution constituant un test. A chaque test, le comportement du système par rapport à son environnement indique s'il vérifie ses spécifications. Cette approche de la vérification est pratique car elle est en général facile à mettre en œuvre: l'environnement existe déjà, donc il suffit d'interfacer le système avec celui-ci pour effectuer les tests, dont les résultats se traduisent directement en termes d'actions effectives sur l'environnement. C'est la raison pour laquelle ce type de simulation est intensivement utilisé. Cependant, par cette approche, le seul moyen de garantir la correction des systèmes de manière absolue est d'en effectuer la simulation exhaustive, c'est à dire de tester que le système se comporte correctement dans toutes les configurations possibles de son environnement. Or, cela est généralement impossible, compte tenu de la combinatoire explosive des comportements de l'environnement.

La vérification formelle consiste alors à substituer à la vérification expérimentale une preuve au sens mathématique du terme, dont le résultat garantit rigoureusement et exactement que les systèmes sont conformes à leurs spécifications. Plusieurs méthodes de preuve ont déjà été développées selon ce principe.

Contexte de l'étude

Le travail relaté dans cet ouvrage porte sur la définition et la réalisation d'un outil de vérification formelle de programmes écrits dans le langage LUSTRE. LUSTRE est un langage de programmation spécialement étudié pour la réalisation de systèmes temps réel. Il est basé sur des concepts originaux offrant aux programmeurs un formalisme adapté à leurs besoins pour décrire les fonctionnalités des systèmes. Ce langage fait naturellement partie du secteur "amont" des outils de production de programmes. Compte tenu de la vocation de LUSTRE, il est toutefois indispensable de développer des méthodes de vérification formelle situées "en aval" des programmes écrits dans ce langage. Ce problème a déjà fait l'objet de plusieurs travaux, qui constituent la base du travail présenté ici.

Parmi les différentes méthodes envisageables pour la vérification formelle de LUSTRE, nous nous sommes particulièrement intéressés à l'une d'entre elles, dont le principe s'apparente à la *simulation exhaustive*. Ce choix trouvera une justification naturelle plus tard dans l'ouvrage. Intuitivement, l'idée de la simulation exhaustive est de simuler un programme pour tous les comportements réalistes de son environnement. Elle constitue donc l'aboutissement de la simulation classique, à laquelle elle se compare naturellement. Toutefois, la simulation exhaustive ne pouvant être effectuée dans un environnement réel (puisqu'il est impossible d'en générer tous les comportements possibles), ce dernier doit être *simulé*. Cela nécessite donc de disposer d'une *description* de l'environnement, ou plus exactement de son comportement. Cela induit aussi d'avoir une description des spécifications qui doivent être respectées (car les résultats de la simulation ne sont plus directement "visualisables" dans l'environnement, comme c'était le cas en simulation classique). La simulation exhaustive reste donc basée sur trois nouvelles entités de la vérification, qui correspondent à celles évoquées précédemment:

- **Le programme**, qui définit un ensemble de *comportements possibles* du système, dont certains sont *exécutables*. Ces derniers sont les seuls pouvant effectivement correspondre à une exécution réaliste du programme.
- **La description comportementale de l'environnement**, qui va permettre de caractériser les *comportements valides* du programme: ceux-ci correspondent à des comporte-

ments réalistes de l'environnement, qui sont les seuls pour lesquels les spécifications doivent impérativement être respectées.

- **Les spécifications**, qui caractérisent les *comportements corrects* du programme: ce sont les comportements qui vérifient les spécifications.

Le principe de la simulation exhaustive consiste alors à vérifier que tous les comportements exécutables et valides du programme sont corrects. L'objet de cette étude est de la mettre en œuvre pour la vérification automatique des programmes LUSTRE.

Plan de lecture

La première partie de cet ouvrage fait un état de l'art de la vérification formelle en LUSTRE. Les chapitres 1, 2 et 3 traitent respectivement des trois entités de la vérification: programmes, spécifications et description comportementale de l'environnement. Le chapitre 1 est un rappel sur LUSTRE: syntaxe et sémantique formelle. Les chapitre 2 et 3 précisent les formalismes utilisés pour la spécification des programmes et pour la description de leur environnement. Les choix effectués à ces niveaux permettent de définir un principe de vérification original des programmes LUSTRE, qui est explicité dans le chapitre 4.

Les bases de la vérification formelle en LUSTRE étant données, la seconde partie s'attache à caractériser formellement la sémantique des spécifications sur un *modèle d'exécution* dénotant tous les comportements possibles d'un programme. Dans le chapitre 5, le modèle est défini à partir d'une sémantique opérationnelle du langage. Le chapitre 6 aborde ensuite le principe de l'évaluation des spécifications, qui est décrit en termes de manipulations formelles sur le modèle. Les difficultés inhérentes à la vérification de LUSTRE sont aussi soulevées. Enfin, les chapitres 7 et 8 constituent une étude des différentes méthodes permettant l'évaluation effective des spécifications sur le modèle d'exécution. A l'issue de cette étude, la méthode la mieux adaptée à la vérification des programmes LUSTRE est mise en évidence. En complément à la méthode de vérification, le diagnostic permet de fournir une assistance à la recherche d'erreurs dans les programmes, lorsque ces derniers ne vérifient pas leurs spécifications. Le chapitre 9 décrit le principe d'un diagnostic complémentaire de la méthode de vérification choisie.

La troisième partie constitue alors l'implémentation de la méthode sélectionnée sous la forme d'un outil de vérification. Le chapitre 10 décrit l'obtention pratique du modèle d'exécution d'un programme LUSTRE. Ce modèle doit être représenté et manipulé au sein de l'outil réalisé: une technique symbolique - appelée BDD - est introduite dans ce but au chapitre 11. Une première implémentation de la méthode de vérification basée sur les BDDs est ensuite proposée dans le chapitre 12. La méthode nécessitant de nombreux appels à des opérateurs complexes, une seconde implémentation optimisant les performances de l'outil est exposée dans le chapitre 13. Enfin, les résultats des expérimentations réalisées sur les outils développés sont recensés dans le chapitre 14.

La quatrième partie propose finalement une autre approche de la vérification réutilisant les BDDs comme technique de représentation du modèle d'exécution, mais se limitant à l'utilisation d'opérateurs de faible complexité pour le manipuler. Cette approche est développée à partir d'une interprétation particulière des BDDs, qui est exposée dans le chapitre 15. Elle donne lieu dans le chapitre 16 à l'implémentation originale d'une méthode de vérification classique, pour laquelle

plusieurs optimisations sont apportées. En outre, les problèmes inhérents à cette méthode sont potentiellement résolus dans cette implémentation.

Partie I

Contexte de l'Etude

Chapitre 1

Réalisation des systèmes en Lustre

LUSTRE [CPHP87,HCRP91a,HCRP91b] est un langage de programmation spécialement étudié pour la réalisation des systèmes réactifs [HP85]. Ces derniers forment une classe particulière de systèmes temps réel, dont la principale caractéristique est d'interagir continuellement avec leur environnement. A chaque instant, le système reçoit en entrée des événements en provenance de son environnement, auxquels il doit réagir en émettant des événements de sortie vers celui-ci. Des exemples de tels systèmes sont nombreux: distributeurs de billets de banque, processus de contrôle/commande, jeux vidéo, automatismes de contrôle divers...

Les caractéristiques des systèmes réactifs sont les suivantes:

- Ce sont des systèmes parallèles: ils sont en général constitués de plusieurs sous-systèmes s'exécutant en parallèle et communicant entre eux.
- Ils sont soumis à des contraintes temporelles strictes: ces systèmes doivent non seulement être corrects du point de vue de leur comportement, mais aussi du point de vue de leur temps de réponse aux événements en provenance de l'environnement.
- Ils sont déterministes: pour chaque événement se produisant dans l'environnement d'un système, la réaction de ce dernier est complètement déterminée.

LUSTRE a été conçu pour faciliter au maximum l'intégration de ces caractéristiques dans les programmes réalisés. Un aperçu du langage est donné ci-dessous.

1.1 Principes du langage

Afin de permettre une description la plus simple et la plus naturelle possible des systèmes réactifs, LUSTRE est basé sur deux concepts originaux: l'approche flot de données [Kah74] et l'interprétation synchrone [Mil83,Ben89].

L'approche flot de donnée consiste à considérer que toute variable ou expression du langage dénote une fonction du temps, ce dernier étant considéré comme isomorphe à l'ensemble des entiers naturels (il est assimilable à une suite d'instants). Les opérateurs du langage opèrent alors globalement sur les fonctions dénotées par leurs opérandes.

L'interprétation synchrone consiste à supposer que toutes les variables et les expressions d'un programme prennent leur n-ième valeur *au même instant.* En pratique, le comportement d'un programme est cyclique, son n-ième cycle d'exécution consistant à calculer la n-ième valeur de chaque variable ou expression.

Les capacités "temps réel" du langage sont dérivées de cette interprétation synchrone, qui est utilisée dans plusieurs autres langages: ESTEREL [BG88], SIGNAL [LGLL91], ARGOS [Mar90]. En LUSTRE, le temps est basé sur une notion logique: toute contrainte de temps portant sur le comportement d'un programme peut s'exprimer par des relations sur l'ordre ou la simultanéité des événements se produisant dans le programme et son environnement. Ainsi, une contrainte du type:

"toute occurrence d'une situation dangereuse
doit être suivie de l'émission d'une alarme dans un délai de deux secondes"

s'exprime en termes d'événements par:

"Après toute occurrence de l'événement `situation_dangereuse`,
une occurrence de l'événement `alarme` *doit se produire*
avant la deuxième prochaine occurrence de l'événement `seconde`*"*

Cet exemple démontre qu'en programmation synchrone, le temps physique n'est pas considéré comme un événement privilégié (`seconde` est traité de la même manière que `situation_dangereuse` ou `alarme`). Cette caractéristique induit une notion de temps "multiforme": ainsi, celui-ci peut aussi bien être compté en secondes qu'en mètres ou en tout autre unité de mesure, étant donné qu'il n'y a par exemple pas de différence conceptuelle entre les deux contraintes suivantes:

"le train doit s'arrêter au bout de 10 secondes"

et

"le train doit s'arrêter au bout de 100 mètres"

L'interprétation synchrone est une abstraction issue de l'hypothèse consistant à considérer que le temps de réaction d'un programme est négligeable par rapport au temps de réaction de son environnement. Les avantages liés à cette abstraction ont été mis en évidence [Ber89c, BB91]: les langages synchrones ont une sémantique propre, et ils réconcilient les notions de parallélisme et de déterminisme. En particulier, ils semblent bien adaptés à la programmation des noyaux réactifs des systèmes temps-réel. On peut toutefois se poser la question de savoir si l'hypothèse synchrone est réaliste. En effet, celle-ci consiste à supposer que le système réagit instantanément aux événements qu'il reçoit en entrée, donc qu'il fonctionne de manière *infiniment* rapide. En pratique, une telle hypothèse sera vérifiée si la vitesse de réaction du système est supérieure à la vitesse d'évolution de l'environnement, cette dernière étant proportionnelle au délai minimal nécessaire à un changement d'état *significatif* de l'environnement. Le respect de cette hypothèse peut facilement être contrôlé au niveau des programmes LUSTRE ceux-ci pouvant être implémentés de manière particulièrement efficace et *mesurable*, selon une technique initialement développée pour le langage ESTEREL [BCG88,BCG87]. Cette technique consiste à

structurer le code objet sous la forme d'un automate. Le code exécuté dans chaque transition est linéaire (sans boucles), ce qui permet de borner supérieurement son temps d'exécution de façon précise pour une machine donnée. De ce fait, la validité de l'hypothèse synchrone peut être rigoureusement contrôlée. Deux versions du compilateur LUSTRE [Pla88,Ray91] utilisent de manière effective cette technique de compilation pour la génération de code exécutable associé aux programmes LUSTRE.

1.2 Syntaxe de Lustre

Nous ne donnons ici qu'une version simplifiée de la syntaxe de LUSTRE, qui est cependant suffisante pour écrire la plupart des programmes LUSTRE et pour comprendre les exemples qui sont traités par la suite. Par ailleurs, l'intégralité de cette syntaxe est fournie dans des ouvrages se rapportant plus spécifiquement au langage [PH87].

1.2.1 Programmes

Un programme LUSTRE est appelé "nœud" (*node* en anglais), en référence à l'interprétation graphique qui peut être associée au langage, et qui sera décrite dans la section 1.2.5. De manière informelle, la structure syntaxique d'un nœud P est la suivante:

```
node P    (<liste de variables d'entrée>)
returns   (<liste de variables de sortie>);
var       (<liste de variables internes>);
let
    <liste d'équations>
tel
```

Un nœud décrit les interactions entre ses variables d'entrée et ses variables de sortie, sous forme d'un ensemble d'équations. Le style déclaratif du langage autorise la définition des équations dans un ordre quelconque. Une fois déclaré, un nœud peut être instancié de façon fonctionnelle dans toute expression, comme un nouvel opérateur du langage. De même que dans les langages classiques, les règles de typage des paramètres limitent les risques d'erreur à ce niveau.

1.2.2 Equations

Toute variable du programme qui n'est pas une entrée est définie par une unique équation, de la forme

<*identificateur*> = <*expression*>;

où <*identificateur*> est un identificateur de variable dénotant une fonction délivrant une valeur à chaque instant, et où <*expression*> est une expression du langage. Toute équation `X = E` stipule l'égalité entre `X` et `E` au sens *mathématique* du terme, c'est à dire qu'à tout instant n, la n-ième valeur de `X` est égale à la n-ième valeur prise par l'expression `E`. Cette égalité mathématique

autorise la définition d'un *principe de substitution* du langage: toute occurrence d'une variable dans le programme peut être substituée par l'expression à laquelle elle correspond dans l'équation où elle est définie.

1.2.3 Expressions

Les expressions du langage sont constituées de constantes (qui correspondent à des suites de valeurs constantes), de variables et d'opérateurs. Tous les opérateurs arithmétiques, booléens et conditionnels standard sont disponibles. Ceux-ci s'appliquent "point à point" sur leurs opérandes. Par exemple, à tout instant n, la n-ième valeur de `X+Y` est égale à la somme des n-ième valeurs de `X` et de `Y`. En plus de ces opérateurs, LUSTRE fournit seulement deux opérateurs spécifiques: l'opérateur "précédent", noté **pre**, et l'opérateur "suivi de", noté `->`. Ces opérateurs sont définis formellement de la manière suivante:

- **pre**: Si E est une expression dénotant la fonction $\lambda n.e(n)$, alors "**pre(E)**" est une expression dénotant la fonction

$$\lambda n. \begin{cases} nil & \text{si } n = 0 \\ e(n-1) & \text{si } n > 0 \end{cases}$$

  où *nil* est une valeur indéfinie.

- `->`: Si E et F sont deux expressions du même type dénotant respectivement les fonctions $\lambda n.e(n)$ et $\lambda n.f(n)$, alors "`E -> F`" est une expression dénotant la fonction

$$\lambda n. \begin{cases} e(n) & \text{si } n = 0 \\ f(n) & \text{si } n > 0 \end{cases}$$

Dans l'exemple suivant, on définit un nœud d'usage général, qui émet une sortie booléenne prenant la valeur **true** à chaque fois que son entrée passe de la valeur **false** à la valeur **true**. En d'autres termes, ce nœud émet en sortie un signal à chaque fois qu'un front montant de son signal d'entrée est détecté. Par convention, la valeur émise à l'instant initial sera égale à **true** si le signal d'entrée a lui même cette valeur. On obtient donc:

```
node Edge(X: bool) returns (edge: bool);
let
   edge = X -> X and not pre(X);
tel
```

Remarquons que le langage autorise la surcharge des identificateurs, tant qu'aucune ambiguïté existe entre les objets dénotés de la même manière. Une instanciation de ce nœud sous la forme de l'expression `Edge(not C)` délivrera alors la valeur **true** à chaque fois qu'un front descendant de `C` est détecté. La figure 1.1 illustre le fonctionnement du nœud **Edge**.

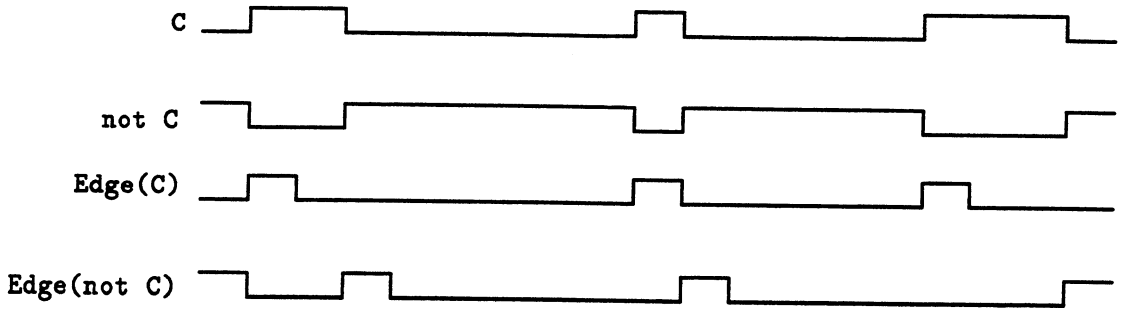


Figure 1.1: Le fonctionnement du nœud **Edge**

1.2.4 Assertions

En plus des constructions syntaxiques de base décrites précédemment, le langage LUSTRE fournit un moyen de décrire des *hypothèses* sur le fonctionnement des programmes et de leur environnement. Ceci est possible grâce au mécanisme des *assertions*, qui a été introduit dans le langage à la suite de travaux réalisés dans ce domaine pour le langage ESTEREL [BCG87]. A l'origine, l'unique intérêt des assertions était d'enrichir les programmes avec des informations permettant aux compilateurs ESTEREL ou LUSTRE d'optimiser le code exécutable généré [Ray91].

De façon très générale, les assertions permettent au programmeur de définir comme une hypothèse toute expression booléenne d'un programme, de telle sorte que cette dernière doit être considérée comme invariante (et toujours *vraie*). La syntaxe d'une assertion est la suivante:

assert <*exp_bool*>;

où <*exp_bool*> est une expression booléenne quelconque. Les assertions ne peuvent être déclarées que dans le corps des programmes, où elles constituent des expressions booléennes sur les variables de celui-ci. Un programme peut contenir plusieurs assertions. Dans ce cas, la sémantique de n assertions sur des expressions booléennes $E_1, \ldots, E_n$ est équivalente à celle d'une seule assertion sur l'expression $\Pi_{i=1}^{n} E_i$. Un petit exemple d'utilisation des assertions est traité ci-dessous.

Le programme **Interrupteur** décrit un interrupteur à deux positions modélisé par les entrées booléennes **allume** et **eteint**, servant à fermer ou à ouvrir un circuit électrique. La circulation de courant dans le circuit est modélisée par une sortie booléenne **courant**.

```
node Interrupteur(allume, eteint: bool) returns(courant: bool);
let
   courant = false -> if allume
                      then true
                      else if eteint
                           then false
                           else pre(courant);
tel
```

Une hypothèse de bon sens sur le fonctionnement de cet interrupteur consiste alors à indiquer qu'il ne peut jamais être allumé et éteint en même temps. Cela est directement traduisible par

```
node Bascule_D (D, h:  bool) returns (q, q_bar:  bool);
let
   q = false -> if pre(Edge(h)) then pre(D) else pre(q);
   q_bar = not q;
tel;
```

Figure 1.2: Description en LUSTRE d'une bascule D déclenchable sur front montant

une assertion exprimant que les entrées **allume** et **eteint** ne prennent jamais la valeur **true** au même instant. D'où on obtient:

```
assert not(allume and eteint);
```

Une telle assertion décrit en fait une caractéristique comportementale de l'environnement du système **Interrupteur**. Elle traduit une hypothèse sur la mécanique du dispositif physique constituant l'interrupteur. Un tel exemple est très intéressant, car il prouve que les assertions peuvent partiellement être utilisées comme un moyen de description comportementale de l'environnement des systèmes. Nous reviendrons sur cette possibilité dans le chapitre 3.

1.2.5 Interprétation graphique de Lustre

Les caractéristiques de LUSTRE permettent d'assimiler tout programme écrit dans ce langage à un réseau "boîtes" correspondant aux opérateurs du langage ou aux nœuds définis par le programmeur, connectés entre eux par des "fils" correspondant aux variables déclarées dans les programmes. Les données circulant sur les fils sont les suites de valeurs qu'elles prennent. Au niveau de chaque boîte, l'opération entre les valeurs fournies en entrée est effectuée, donnant une nouvelle valeur émise en sortie. L'interprétation synchrone d'un tel réseau consiste à considérer comme *nul* les temps de calcul des opérateurs et de circulation des valeurs sur les fils.

Afin d'illustrer l'interprétation graphique du langage LUSTRE, nous définissons ci-dessous un nœud décrivant une bascule D déclenchable sur front montant de son horloge. Ce circuit primaire accepte en entrée deux variables booléennes: D, qui est la valeur à mémoriser, et h qui est l'horloge. La bascule retourne en sortie deux variables booléennes q et q_bar définies comme suit:

- La valeur de q_bar est toujours le complément de la valeur de q;
- Initialement, q a la valeur *fausse*;
- A chaque fois qu'un front montant de h est détecté (c'est-à-dire à chaque fois que h passe de *faux* à *vrai*), q prend la valeur de D. Autrement, la valeur de q ne change pas. D'autre part, la bascule est supposée réagir à ses entrées après une unité de temps.

Figure 1.3: Représentation graphique des opérateurs LUSTRE

La figure 1.2 fournit une description en LUSTRE de la bascule spécifiée ci-dessus. Celle-ci utilise le nœud **Edge** défini dans la section 1.2. Le temps de réaction de la bascule est pris en compte grâce aux opérateurs **pre** apparaissant dans la définition de **q**.

Un tel programme peut très bien être visualisé sous la forme d'un réseau d'opérateurs. En utilisant les conventions graphiques de la figure 1.3, le réseau obtenu est celui de la figure 1.4. On peut remarquer sur cet exemple la similitude entre le réseau d'opérateurs et une description matérielle de la bascule.

1.2.6 Le langage Saga

Le langage SAGA [JLB88], développé par Merlin Gerin/SES (département Systèmes et Electronique de Sûreté), est utilisé en milieu industriel pour la conception de logiciels implémentant des systèmes nécessitant un haut degré de sûreté, notamment dans les domaines du nucléaire civil et de la défense. SAGA est fondé sur les mêmes principes que LUSTRE: c'est un langage de type flot de données, doté d'une interprétation synchrone. Plus précisément, SAGA emprunte à LUSTRE sa sémantique et son interprétation graphique. C'est donc un langage graphique où là encore, les programmes sont représentés sous forme de boîtes connectées entre elles par des fils. De ce point de vue, il est donc très similaire à l'interprétation graphique de LUSTRE donnée par la figure 1.3, mis à part que l'aspect externe des opérateurs n'est pas le même.

Cependant, la grande originalité liée au langage SAGA réside dans le fait qu'il n'est accessible qu'à travers un atelier logiciel du même nom, au sein duquel l'utilisation de diagrammes de type SADT [Ros77] a été adaptée à l'analyse et à la conception des programmes d'automatisme temps réel réalisés à Merlin Gerin/SES. De fait, l'atelier SAGA intègre un certain nombre de règles et de méthodes inspirées de SADT, ainsi que d'autres fonctionnalités dont les principales sont brièvement décrites ci-dessous:

- **Démarche descendante:** La conception d'un programme SAGA est réalisée de manière descendante à partir de sa spécification la plus générale. Le programme global à réaliser est appelé application. Le point de départ de la réalisation d'une application consiste à définir l'interface avec son environnement. Les entrées et les sorties sont initialement d'un type totalement inconnu. La suite de la conception va permettre l'affinement progressif de l'application en fonctions et sous-fonctions et de ses données d'interface en sous-données, jusqu'à l'obtention d'une description complète. Il est intéressant de noter que la conception descendante est guidée par l'outil, qui interdit d'abandonner la description d'une fonction tant que celle-ci n'est pas correcte.

- **Vérifications contextuelles:** Au cours de la conception d'une application, l'atelier SAGA

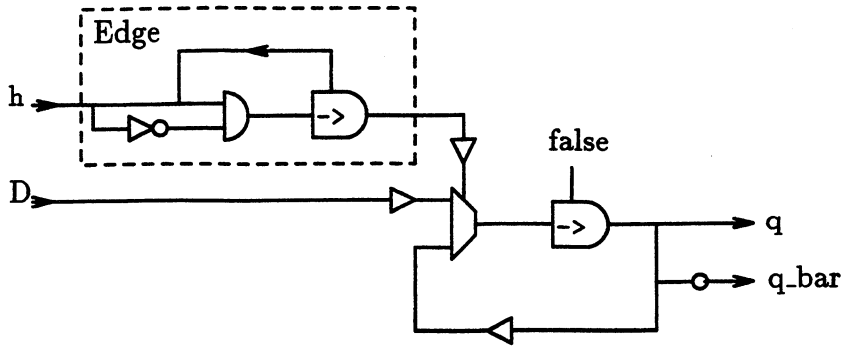


Figure 1.4: Le réseau d'opérateurs de la bascule D

réalise automatiquement un certain nombre de vérifications sur le typage des données, l'absence de valeurs indéfinies et l'absence d'interblocages au sein de l'application.

- **Documentation:** L'atelier comprend un outil permettant de mettre en forme l'ensemble des informations données par l'utilisateur lors de la phase de conception des programmes.
- **Génération de code:** Les applications SAGA peuvent être traduites automatiquement sous forme de programmes exécutables.
- **Administration:** Un outil d'administration permet de gérer l'ensemble des applications développées au sein de l'atelier, de créer de nouvelles applications et les différentes versions de ces applications, et de contrôler l'accès aux différentes applications par les utilisateurs.

SAGA peut donc être vu comme un LUSTRE "graphique", dont les contraintes d'utilisation imposées par l'atelier au sein duquel il est utilisé permettent le développement et la maintien de systèmes réactifs temps réel en milieu industriel. C'est pourquoi on dira que SAGA est un langage *de conception*, tandis que LUSTRE est un langage *de programmation*.

D'autre part, il faut noter que tout programme SAGA peut être traduit en LUSTRE, et qu'un traducteur automatique est disponible. Par conséquent, tous les outils et les méthodes développés pour LUSTRE peuvent être appliquées à SAGA. En revanche, remarquons que LUSTRE ne peut être traduit en SAGA, car son pouvoir d'expression est plus grand.

1.3 Sémantique des traces de Lustre

Dans la section précédente, la signification des principales constructions syntaxiques de LUSTRE est donnée par leur sémantique intuitive. Dans le cadre de la vérification, il est nécessaire d'avoir une sémantique formelle du langage, afin de pouvoir définir rigoureusement ce que sont les comportements *exécutables*, *valides* ou *corrects* d'un programme. Dans ce but, nous introduisons ici la *sémantique des traces* de LUSTRE, qui permet une caractérisation des comportements d'un programme en termes de *traces d'exécution*.

De façon intuitive, tout comportement d'un programme LUSTRE correspond aux suites de valeurs prises par chacune de ses variables. Ces valeurs définissent à chaque instant un "état" du

programme, dont toute suite constitue un comportement. Les notions d'état et de comportement qui viennent d'être évoquées sont définies formellement ci-dessous.

Définition 1 *Soit I l'ensemble des identificateurs de* LUSTRE, *et V l'ensemble des valeurs prises par ces identificateurs. Toute fonction σ de I dans V définit une mémoire du programme.*

Tout état d'un programme est caractérisé formellement par une *mémoire.*

Définition 2 *Une trace d'exécution Σ est une séquence non vide, finie ou infinie de mémoires.*

La sémantique de LUSTRE impose que tout comportement d'un programme est formé de suites *infinies* de valeurs de ses variables. Par conséquent, un tel comportement est caractérisé formellement par une trace d'exécution infinie. On peut alors définir une relation de compatibilité entre une trace d'exécution infinie Σ et un programme P, permettant de déterminer si Σ dénote un comportement de P. Cette relation permettra donc de caractériser pour tout programme LUSTRE l'ensemble des traces d'exécution infinies dénotant un comportement de ce programme.

Dans un premier temps, nous définissons la valeur de toute expression LUSTRE par rapport à une trace d'exécution finie. Ensuite, nous caractérisons la compatibilité d'une trace infinie avec un programme LUSTRE.

1.3.1 Sémantique des expressions Lustre sur les traces finies

La valeur de toute expression LUSTRE E sur une trace finie $\Sigma = (\sigma_0, \ldots, \sigma_n)$ est définie par induction sur la structure de E. Le fait que E a la valeur v sur Σ sera noté:

$$\Sigma \vdash \mathtt{E} \mid v$$

Les règles suivantes permettent de définir la valeur de toute expression E sur une trace Σ quelconque:

Valeur d'une constante:
Le symbole k dénotant toute constante du langage,

$$\Sigma \vdash \mathtt{k} \mid k$$

Valeur d'une variable:
Le symbole x dénotant toute variable du langage,

$$(\sigma_0, \ldots, \sigma_n) \vdash \mathtt{x} \mid \sigma_n(\mathtt{x})$$

Valeur de $\star$(E$_1$, ..., E$_m$):
E$_1$,...,E$_n$ dénotant des expressions quelconques du langage, et $\star$ dénotant tout opérateur arithmétique, booléen ou conditionnel standard,

$$\frac{\Sigma \vdash \mathtt{E}_1 \mid v_1 \ , \ \ldots \ , \ \Sigma \vdash \mathtt{E}_m \mid v_m}{\Sigma \vdash \star(\mathtt{E}_1, \ \ldots, \ \mathtt{E}_m) \mid \star(v_1, \ldots, v_m)}$$

Valeur de E_1 -> E_2:
E_1 et E_2 dénotant deux expressions quelconques du langage,

$$\frac{\sigma_0 \vdash E_1 \mid v_1}{\sigma_0 \vdash E_1 \text{ -> } E_2 \mid v_1} \qquad \frac{\Sigma.\sigma \vdash E_2 \mid v_2}{\Sigma.\sigma \vdash E_1 \text{ -> } E_2 \mid v_2}$$

Valeur de pre(E):
E dénotant une expression quelconque du langage, et *nil* dénotant une valeur *indéterminée*,

$$\sigma_0 \vdash \texttt{pre(E)} \mid nil \qquad \frac{\Sigma \vdash E \mid v}{\Sigma.\sigma \vdash \texttt{pre(E)} \mid v}$$

1.3.2 Compatibilité d'une trace infinie avec un programme

La sémantique des programmes LUSTRE n'est définie que par rapport à des comportements infinis. En terme de traces d'exécution, la sémantique d'un programme LUSTRE correspond à l'ensemble des traces d'exécution infinies compatibles avec celui-ci.

Nous définissons donc la compatibilité entre les traces d'exécution infinies et les programmes LUSTRE. Le fait qu'une trace infinie Σ est compatible avec un programme P sera noté:

$$\Sigma \vdash P$$

Une trace infinie dénotera alors un comportement de P si et seulement si tous ses préfixes finis sont compatibles avec P:

$$(\sigma_0, \ldots, \sigma_n, \ldots) \vdash P \iff \forall n \geq 0, (\sigma_0, \ldots, \sigma_n) \vdash P$$

La compatibilité des traces finies avec P est définie par les règles ci-dessous. Un programme LUSTRE est ici considéré comme un ensemble d'équations, indépendamment de sa structure syntaxique. Cette hypothèse ne modifie en rien la sémantique du programme, puisque le langage étant déclaratif, l'ordre des équations est sans importance.

Compatibilité avec les équations:

$$\frac{(\sigma_0, \ldots, \sigma_n) \vdash E \mid v \ , \ \sigma_n(\mathbf{x}) = v}{(\sigma_0, \ldots, \sigma_n) \vdash \mathbf{x} \texttt{ = } E}$$

Compatibilité avec la composition:

$$\frac{\Sigma \vdash P_1 \ , \ \Sigma \vdash P_2}{\Sigma \vdash P_1; P_2}$$

Compatibilité avec les assertions:

$$\frac{\Sigma \vdash A \mid true}{\Sigma \vdash \texttt{assert A}}$$

Par la suite, on appellera trace *exécutable* d'un programme toute trace infinie compatible avec ce programme (car elle en dénote un comportement). De même, on appellera trace *valide* d'un programme toute trace infinie compatible avec les assertions de ce programme.

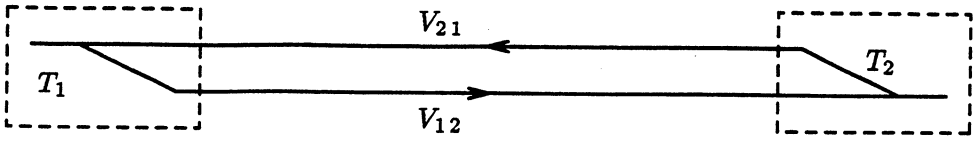


Figure 1.5: Une ligne de métro

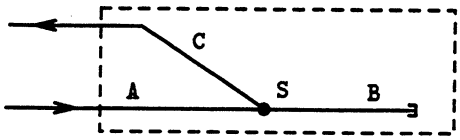


Figure 1.6: La section permettant aux rames de changer de voie

1.4 Exemple de réalisation d'un système

Afin d'illustrer l'utilisation de LUSTRE en tant que langage de programmation des systèmes réactifs, nous décrivons ici un exemple de réalisation d'un dispositif ferroviaire employé sur les lignes de métro. Celui-ci sera repris par la suite dans les différentes phases de la vérification.

1.4.1 Spécification informelle

De façon schématique, une ligne de métro est constituée de deux terminus T_1 et T_2 reliés entre eux par l'intermédiaire de deux voies unidirectionnelles, empruntées par les rames de métro. La voie V_{12} permet à celles-ci d'effectuer le trajet de T_1 vers T_2, tandis que la voie V_{21} leur permet d'aller de T_2 vers T_1 (voir figure 1.5).

A chaque terminus, les rames pénètrent dans une "section terminale", qui leur permet de changer de voie pour repartir dans la direction opposée. Cette section est composée de trois voies A, B, C et d'un aiguillage S (voir figure 1.6). La voie A étant la voie d'entrée dans la section et la voie C étant la voie de sortie, les rames allant de A vers C doivent d'abord attendre que A et B soient connectées par l'aiguillage S, pour transiter ensuite sur B et attendre à nouveau que B et C soient connectées par S avant de ressortir par C.

Etant donné que plusieurs rames circulent en même temps sur les voies et que l'aiguillage n'est pas un dispositif sûr (il peut tomber en panne), deux genres d'accidents sont susceptibles de se produire dans une section terminale:

- si plusieurs rames sont autorisées à la fréquenter en même temps, celles-ci risquent d'entrer en collision.
- si l'aiguillage n'est pas vérrouillé en position sûre au moment où une rame circule dessus, cette dernière sera victime d'un déraillement.

Il est donc clair que le contrôle d'une section terminale est une tâche critique. Tout système de

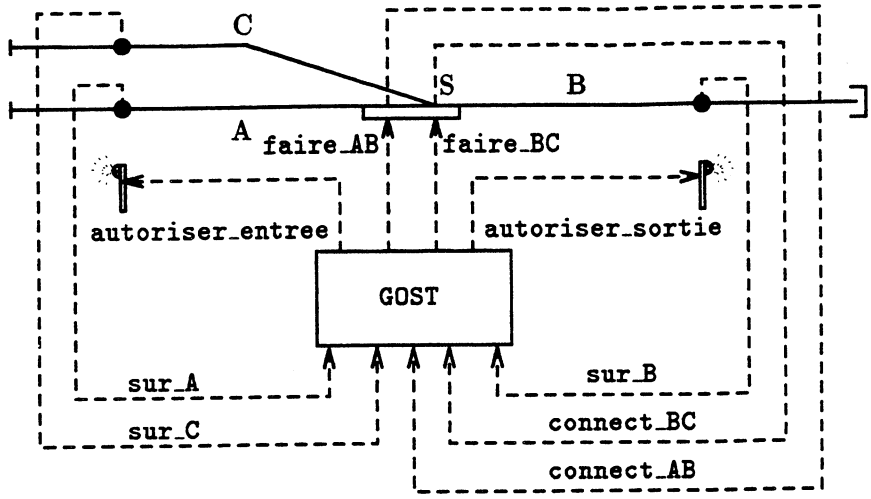


Figure 1.7: Le système GOST et son environnement

Gestion AUtomatique de Section Terminale (dorénavant appelé GOST) doit à la fois commander l'aiguillage et gérer la circulation des rames de telle sorte qu'aucun accident ne puisse se produire dans la section.

Un tel système est typiquement un système réactif: en réponse à des informations concernant le positionnement de l'aiguillage et la position des rames dans la section, il doit délivrer des requêtes de positionnement à l'aiguillage et des autorisations de circulation aux rames. Ces quatre types d'événements peuvent être modélisés par les signaux suivants:

- `connect_AB` et `connect_BC` sont émis par l'aiguillage pour indiquer si celui-ci connecte A à B ou B à C. Si aucun de ces signaux n'est actif, l'aiguillage est en position indéterminée et par conséquent, les rames ne doivent pas être autorisés à circuler dessus.

- `sur_A`, `sur_B` et `sur_C` sont des signaux émis par trois capteurs placés respectivement sur les voies A, B et C. Ils sont actifs pendant tout le temps où une rame circule sur la voie qu'ils surveillent. Ces capteurs indiquent donc uniquement la présence ou non de rames sur leurs voies respectives.

- `faire_AB` et `faire_BC` sont des signaux de requête émis par le système pour ordonner à l'aiguillage de connecter A à B ou B à C.

- `autoriser_entree` et `autoriser_sortie` sont des signaux d'autorisation de circulation des rames à l'intérieur de la section. Ils correspondent en pratique à des signaux lumineux bicolores. Le premier signal doit autoriser les rames à pénétrer dans la section terminale seulement si elle est vide et si l'aiguillage connecte A à B. Le second doit autoriser les rames à quitter la voie B seulement si l'aiguillage connecte B à C.

La figure 1.7 donne une vue d'ensemble du système GOST et de son environnement.

1.4.2 Réalisation

Le système GOST peut maintenant être implémenté en LUSTRE. On suppose qu'initialement, il n'y a pas de rames dans la section terminale. Nous définissons dans un premier temps les équations exprimant les requêtes de positionnement de l'aiguillage. Il sera demandé à l'aiguillage de connecter A à B à chaque fois que la section est vide, et de connecter B à C à chaque fois qu'une rame se trouve en transit sur B. En outre, ces requêtes resteront actives tant qu'elles ne sont pas satisfaites. En LUSTRE, on obtient alors directement les équations suivantes:

```
faire_AB = section_vide and not connect_AB;
faire_BC = transit_sur_B and not connect_BC;
```

Ici, `section_vide` doit exprimer qu'aucune rame ne se trouve dans la section, et `transit_sur_B` doit exprimer que les rames se trouvant dans la section terminale sont en transit sur B, tandis que A et C sont inoccupées. Les équations définissant ces variables sont alors:

```
section_vide = not(sur_A or sur_B or sur_C);
transit_sur_b = sur_B and not(sur_A or sur_C);
```

Il faut maintenant écrire les équations définissant les autorisations de circulation. Comme cela a été mentionné précédemment, l'entrée dans la section terminale ne sera autorisée que si elle est vide et si l'aiguillage connecte A à B. Cela se traduit alors par l'équation suivante:

```
autoriser_entree = section_vide and connect_AB;
```

D'autre part, les rames ne sont autorisées à quitter la section terminale que si elles sont en transit sur B et si l'aiguillage connecte B à C. On obtient donc:

```
autoriser_sortie = transit_sur_b and connect_BC;
```

```
node GOST(sur_A,sur_B,sur_C,connect_AB,connect_BC: bool)
returns (autoriser_entree,autoriser_sortie,faire_AB,faire_BC: bool);
var section_vide,transit_sur_B: bool;
let
   autoriser_entree = section_vide and connect_AB;
   autoriser_sortie = transit_sur_B and connect_BC;
   faire_AB = section_vide and not connect_AB;
   faire_BC = transit_sur_B and not connect_BC;
   section_vide = not(sur_A or sur_B or sur_C);
   transit_sur_B = sur_B and not(sur_A or sur_C);
tel
```

Figure 1.8: Un programme LUSTRE implémentant le système GOST

Finalement, le programme LUSTRE complet implémentant le système GOST est donné par la figure 1.8. Cet exemple simple démontre que LUSTRE est bien adapté à la programmation de ce genre de systèmes, car toutes les équations écrites dans le programme GOST sont directement et naturellement déduites des spécifications informelles du système. De plus, la possibilité d'écrire les équations dans un ordre quelconque autorise une traduction progressive des spécifications: chaque fonctionnalité du système est analysée tour à tour, nécessitant parfois l'introduction de variables auxiliaires. Il faut remarquer à ce sujet que la définition de telles variables (comme par exemple `section_vide`) permettant de nommer des expressions, n'a aucune influence sur la sémantique du programme. Leur seul but est d'en augmenter la lisibilité.

Chapitre 2

Spécification des programmes Lustre

La seconde entité intervenant dans la vérification formelle des systèmes concerne leur description comportementale. Au plus haut niveau, les spécifications du système sont contenues dans un cahier des charges rédigé en langue naturelle: celui-ci décrit les fonctionnalités que le système doit réaliser. En particulier, il décrit comment ce dernier doit se comporter par rapport à son environnement. A ce stade, l'objectif que nous nous fixons pour la vérification des systèmes est limité à une preuve de correction *partielle* des systèmes réalisés. En clair, il n'est pas question d'essayer de vérifier intégralement leur bon fonctionnement, mais simplement de s'assurer qu'ils respectent certaines contraintes *vitales* de leur comportement *observable*. Ainsi, dans un système de pilotage automatique d'avion, une erreur sur l'affichage de la vitesse a de faibles conséquences, tandis qu'il est impératif de s'assurer que le train d'atterrissage de l'avion sera bien vérrouillé au moment d'atterrir.

D'autre part, le caractère automatique de la vérification induit naturellement une interprétation automatique des spécifications, ce qui oblige à exprimer ces dernières dans un formalisme plus strict que la langue naturelle, qui est intrinsèquement ambiguë. Il est donc nécessaire de disposer d'un tel formalisme pour spécifier les programmes LUSTRE. Le choix de celui-ci dépend des deux questions suivantes:

- Quel genre de spécifications a-t-on besoin de vérifier?
- Comment exprimer ces spécifications de façon simple mais suffisamment expressive?

D'importants travaux ont été menés pour trouver un formalisme permettant d'exprimer les spécifications devant être respectées par les programmes LUSTRE. Ils fournissent une réponse aux deux questions ci-dessus, sous la forme d'un *langage de spécification*. Dans ce chapitre, nous rappelons les idées et les choix qui ont abouti à sa définition. Nous donnons ensuite la sémantique formelle des spécifications d'un programme exprimées dans ce langage. Enfin, nous traitons un exemple de spécification d'un système réalisé en LUSTRE.

2.1 Restriction aux propriétés de sûreté

Dans notre approche de la vérification, le langage de spécification utilisé doit permettre d' exprimer les fonctionnalités vitales des systèmes. Il faut par conséquent savoir ce que sont ces

fonctionnalités.

Dans le cadre des systèmes implémentés en LUSTRE, une étude de cas [Glo89] a été menée autour d'applications réelles développées par Merlin Gerin en SAGA. Son résultat a abouti sur le fait que la plupart des propriétés critiques des systèmes sont des propriétés de "sûreté" au sens de [BA85](*safety properties* en anglais), qui expriment par exemple le fait qu'une situation ne doit *jamais* arriver, ou qu'une certaine condition doit *toujours* être vérifiée. Les propriétés de sûreté expriment des *invariants* des systèmes. Elles peuvent être opposées aux propriétés de "vivacité" (*liveness properties*), qui expriment par exemple qu'une situation se produira *inévitablement* dans le futur. Ainsi, la proposition "*le train finira inévitablement par s'arrêter en présence d'un feu rouge*" est une propriété de vivacité, tandis que la proposition "*le train s'arrêtera toujours en présence d'un feu rouge*" est une propriété de sûreté. En l'occurrence, il est bien plus important de vérifier cette dernière.

Le résultat de l'étude de cas évoquée ci-dessus a des conséquences importantes pour la vérification des programmes LUSTRE. Ainsi, en ce qui concerne le choix ou la définition d'un langage de spécification, on peut se limiter à un langage permettant d'exprimer seulement des propriétés de sûreté. De plus, la vérification de ce type de propriétés est plus simple à effectuer:

- Les propriétés de sûreté peuvent être vérifiées sur des *abstractions* de programmes. Intuitivement, il est possible de simplifier un programme P en un programme P' ayant plus de comportements que P, de telle sorte que si P' satisfait les spécifications, alors P les satisfait aussi. Cette technique d'abstraction est d'autant plus réaliste que les propriétés critiques considérées dépendent rarement de relations numériques, mais très souvent de dépendances logiques entre événements. De ce fait, leur vérification peut très souvent être réalisée sur des abstractions finies des programmes.
- Des méthodes de vérification formelle ont déjà été développées pour ce type de propriétés.

Par la suite, on entendra par "spécification" toute propriété de sûreté exprimée par rapport à un programme LUSTRE.

2.2 Lustre pour la spécification

Il s'agit maintenant de trouver un langage permettant d'exprimer de manière simple et non ambiguë les spécifications de programmes LUSTRE. Des formalismes généraux permettant d'exprimer les propriétés de sûreté des systèmes ont déjà été proposés: il s'agit essentiellement des logiques temporelles [Sif82,CES86] et du μ-calcul [Koz83]. L'un d'eux a déjà été utilisé pour spécifier les programmes LUSTRE [Rat88]. Toutefois, le travail réalisé dans l'étude de cas menée à Merlin Gerin a permis de définir plusieurs langages spécialement adaptés à la spécification des programmes LUSTRE [Glo89]. Il se trouve que LUSTRE répond lui-même parfaitement au problème. En effet, LUSTRE peut être vu comme une logique temporelle exécutable, dans laquelle on peut exprimer n'importe quelle propriété de sûreté sur un programme [BFH90b]. C'est pourquoi LUSTRE a été choisi comme langage de spécification des programmes LUSTRE. Ce choix procure un avantage intéressant du point de vue méthodologique, puisque les programmes et leurs spécifications sont écrits dans le même langage, ce qui augmente l'ergonomie de la vérification. D'autres avantages liés à cette approche "mono-langage" apparaissent au niveau des méthodes de vérification formelle mises en œuvre. Ce dernier point sera évoqué dans le chapitre 4.

2.3 Sémantique des spécifications sur les traces

Intuitivement, toute propriété de sûreté φ d'un système est un invariant qui doit être *vrai* durant toute exécution de celui-ci. Si le système est implémenté par un programme LUSTRE, son exécution est modélisée par l'évolution des variables du programme. φ s'exprime alors sous la forme d'une expression booléenne $\mathtt{E}_\varphi$ sur ces variables. φ est satisfaite par le système si et seulement si $\mathtt{E}_\varphi$ est toujours vraie durant toute exécution du programme.

D'un point de vue formel, on peut noter qu'il existe une correspondance évidente entre les expressions booléennes LUSTRE et les propriétés qu'il est possible d'exprimer sur les traces finies. Toute expression booléenne peut être interprétée comme exprimant une propriété sur les traces finies, et inversement. En reprenant les notations précédentes, une trace finie Σ satisfait alors une propriété φ si et seulement si l'expression $\mathtt{E}_\varphi$ prend la valeur *vraie* sur Σ:

$$\Sigma \models \varphi \Longleftrightarrow \Sigma \vdash \mathtt{E}_\varphi \mid \mathit{true}$$

Dés lors, une propriété de sûreté de la forme $\psi = \Box\varphi$ est satisfaite par une trace infinie Σ si et seulement si tout préfixe fini de Σ satisfait φ. Une telle trace sera alors dite *trace correcte* vis à vis de la propriété ψ. ψ est donc satisfaite par un programme P si et seulement si elle est satisfaite par toute trace infinie de P:

$$\begin{aligned} P \models \Box\varphi \iff & \forall(\sigma_0,\ldots,\sigma_n,\ldots) \text{ satisfaisant } (\sigma_0,\ldots,\sigma_n,\ldots) \vdash P, \\ & \forall n \geq 0, (\sigma_0,\ldots,\sigma_n) \models \varphi \\ \iff & \forall(\sigma_0,\ldots,\sigma_n,\ldots) \text{ satisfaisant } (\sigma_0,\ldots,\sigma_n,\ldots) \vdash P, \\ & \forall n \geq 0, (\sigma_0,\ldots,\sigma_n) \vdash \mathtt{E}_\varphi \mid \mathit{true} \end{aligned}$$

Dans un premier temps, nous considérerons les opérateurs définis comme exprimant des propriétés sur les traces finies. Chaque opérateur *op* sera défini formellement en utilisant le cadre de la logique temporelle standard [LPZ85,MP88]. A chaque opérateur *op* sera associé un nœud LUSTRE `Op` tel que

$$(\sigma_0,\ldots,\sigma_n) \models op(\varphi_1,\ldots,\varphi_k) \Longleftrightarrow (\sigma_0,\ldots,\sigma_n) \vdash \mathtt{Op}(\mathtt{E}_{\varphi_1},\ldots,\mathtt{E}_{\varphi_k}) \mid \mathit{true} \qquad (2.1)$$

Nous considérerons ensuite les opérateurs comme définissant des propriétés de sûreté sur les traces infinies. A chaque opérateur *op* sera associé une implémentation sous la forme d'un nœud LUSTRE `Op` tel que

$$(\sigma_0,\ldots,\sigma_n,\ldots) \models op(\psi_1,\ldots,\psi_k) \Longleftrightarrow \forall n \geq 0, (\sigma_0,\ldots,\sigma_n) \vdash \mathtt{Op}(\mathtt{E}_{\psi_1},\ldots,\mathtt{E}_{\psi_k}) \mid \mathit{true} \qquad (2.2)$$

Les équivalences (2.1) et (2.2) peuvent être établies à partir de la sémantique des traces du langage.

2.4 Opérateurs temporels

Pour l'instant, nous ne disposons que des opérateurs standards de LUSTRE pour construire les expressions booléenne exprimant les propriétés de sûreté à vérifier. Il serait intéressant de disposer d'opérateurs temporels plus généraux pour les exprimer. Nous montrons ici à l'aide de plusieurs

exemples que de tels opérateurs peuvent être définis. La sémantique formelle des opérateurs exhibés est donnée par rapport aux traces d'exécution. Leur implémentation sous forme de nœuds LUSTRE est aussi fournie, ce qui constitue une bonne illustration du langage.

2.4.1 Propriétés sur les traces finies

A tout identificateur de variable booléenne **x** apparaissant dans un programme, on associe un prédicat de base noté x, tel que $(\sigma_0, \ldots, \sigma_n) \models x$ si et seulement si $\sigma_n(\mathbf{x}) = vrai$. Les opérateurs booléens seront interprétés classiquement:

$$\begin{array}{lll} \Sigma \models \neg\varphi & \text{ssi} & \Sigma \not\models \varphi \\ \Sigma \models \varphi_1 \vee \varphi_2 & \text{ssi} & \Sigma \models \varphi_1 \quad \text{ou} \quad \Sigma \models \varphi_2 \\ \Sigma \models \varphi_1 \wedge \varphi_2 & \text{ssi} & \Sigma \models \varphi_1 \quad \text{et} \quad \Sigma \models \varphi_2 \end{array}$$

L'opérateur "After": Si φ est une propriété, "*after* φ" est *vraie* si et seulement si φ a été *vraie* au moins une fois dans le passé strict. Formellement:

$$(\sigma_0, \ldots, \sigma_n) \models \textit{after}\ \varphi \quad \text{ssi} \quad \exists i, 0 \leq i < n, \text{ satisfaisant } (\sigma_0, \ldots, \sigma_i) \models \varphi$$

Cet opérateur est implémenté par un nœud LUSTRE `After`, prenant en entrée l'expression E_φ, et renvoyant $\mathrm{E}_{after\ \varphi}$. A partir de cette définition, "*after* φ" est *fausse* sur toute trace constituée d'une seule mémoire. Elle est *vraie* sur une trace plus longue $\Sigma.\sigma$ si et seulement si ou bien elle est *vraie* sur le préfixe Σ, ou si φ est *vraie* sur $\Sigma.\sigma$. On obtient alors le nœud suivant:

```
node After(phi:  bool) returns (after_phi:  bool);
let
   after_phi = false -> (pre(phi) or pre(after_phi));
tel
```

L'opérateur "Once... since": Si φ_1 et φ_2 sont des propriétés, "*once* φ_1 *since* φ_2" est *vraie* si et seulement si ou bien φ_2 n'a jamais eté *vraie*, ou si φ_1 a été *vraie* au moins une fois depuis le dernier instant où φ_2 était *vraie*. Cela s'exprime formellement par:

$$\begin{array}{lll} (\sigma_0, \ldots, \sigma_n) \models \textit{once}\ \varphi_1\ \textit{since}\ \varphi_2 \text{ ssi ou bien} & & \forall i, 0 \leq i \leq n, (\sigma_0, \ldots, \sigma_i) \not\models \varphi_2 \\ \text{ou} & & \exists i, 0 \leq i \leq n, \text{ tel que } (\sigma_0, \ldots, \sigma_i) \models \varphi_2 \\ & & \text{et } \forall j, i < j \leq n, (\sigma_0, \ldots, \sigma_j) \not\models \varphi_2 \\ & & \text{et } \exists k, i \leq k \leq n, (\sigma_0, \ldots, \sigma_k) \models \varphi_1 \end{array}$$

Le nœud[1] LUSTRE `Once_since_` prend en entrée E_{φ_1} et E_{φ_2}, et retourne $\mathrm{E}_{once\ \varphi_1\ since\ \varphi_2}$. Quand φ_2 est *vraie*, "*once* φ_1 *since* φ_2" a la même valeur que φ_1. Autrement, elle est *vraie:*

- Sur toute trace constituée d'une seule mémoire.

[1] Les noms de nœuds sont déterminés selon la convention suivante: les caractères "underscore" apparaissent à l'endroit où les noms de paramètres apparaîtraient normalement; par exemple, `Once_since_(A,B)` exprime "Once A since B".

- Sur une trace $\Sigma.\sigma$ si et seulement si ou bien elle est *vraie* sur le préfixe Σ, ou si φ_1 est *vraie* sur $\Sigma.\sigma$.

On obtient le nœud suivant:

```
node Once_since_(phi1,phi2:  bool) returns (once_phi1_since_phi2:  bool);
let
   once_phi1_since_phi2 =
                  if phi2 then phi1
                  else (true -> (phi1 or pre(once_phi1_since_phi2)));
tel
```

L'opérateur "Always... since...": De façon similaire, on peut définir "*always* φ_1 *since* φ_2" comme étant *vraie* si et seulement si ou bien φ_2 n'a jamais été *vraie*, ou φ_1 a été continuellement *vraie* depuis le dernier instant où φ_2 était *vraie*:

$$(\sigma_0,\ldots,\sigma_n) \models \textit{always } \varphi_1 \textit{ since } \varphi_2 \text{ ssi ou bien } \forall i, 0 \leq i \leq n, (\sigma_0,\ldots,\sigma_i) \not\models \varphi_2$$
$$\text{ou } \exists i, 0 \leq i \leq n, \text{ tel que } (\sigma_0,\ldots,\sigma_i) \models \varphi_2$$
$$\text{et } \forall j, i < j \leq n, (\sigma_0,\ldots,\sigma_j) \not\models \varphi_2$$
$$\text{et } \forall k, i \leq k \leq n, (\sigma_0,\ldots,\sigma_k) \models \varphi_1$$

On obtient le nœud correspondant:

```
node Always_since_(phi1,phi2:bool) returns (always_phi1_since_phi2:bool);
let
   always_phi1_since_phi2 =
                  if Never(phi2) then true
                  else if phi2 then phi1
                  else (phi1 and pre(always_phi1_since_phi2));
tel
```

où le nœud **Never** est défini comme suit:

```
node Never (phi:  bool) returns (never:  bool);
let
   never = not phi -> (not phi and pre(never));
tel
```

Ce nœud implémente l'opérateur suivant:

$$(\sigma_0,\ldots,\sigma_n) \models \textit{never } \varphi \text{ ssi } \forall i, 0 \leq i \leq n, (\sigma_0,\ldots,\sigma_i) \not\models \varphi$$

2.4.2 Propriétés de sûreté

On définit maintenant des opérateurs temporels dont la sémantique est définie sur des traces d'exécution infinies. Ils expriment donc des propriétés de sûreté.

L'opérateur "Once... from... to...": "*once* φ_1 *from* φ_2 *to* φ_3" est *vraie* si et seulement si φ_1 est *vraie* au moins une fois dans tout intervalle de temps minimal débutant quand φ_2 est *vraie* et finissant quand φ_3 est *vraie*. Formellement, une trace Σ satisfait "*once* φ_1 *from* φ_2 *to* φ_3" si et seulement si tout préfixe de Σ satisfaisant φ_3 satisfait aussi "*once* φ_1 *since* φ_2":

$$(\sigma_0, \ldots, \sigma_n, \ldots) \models \textit{once } \varphi_1 \textit{ from } \varphi_2 \textit{ to } \varphi_3 \quad \text{ssi}$$
$$\forall i \geq 0, \quad (\sigma_0, \ldots, \sigma_i) \models \varphi_3 \implies (\sigma_0, \ldots, \sigma_i) \models \textit{once } \varphi_1 \textit{ since } \varphi_2$$

La sortie du programme ci-dessous est toujours égale à **true** si et seulement si ses paramètres d'entrée représentent des propriétés φ_1, φ_2 et φ_3 qui satisfont "*once* φ_1 *from* φ_2 *to* φ_3" :

```
node Once_from_to_(phi1,phi2,phi3:  bool)
        returns (once_phi1_from_phi2_to_phi3:  bool);
let
   once_phi1_from_phi2_to_phi3 = Implies(phi3, Once_since_(phi1,phi2));
tel
```

Le nœud implémentant l'implication est le suivant:

```
node Implies(X,Y: bool) returns (X_implies_Y: bool);
let
   X_implies_Y = not X or Y;
tel
```

L'opérateur "Always... from... to...": De façon similaire, on peut définir "*always* φ_1 *from* φ_2 *to* φ_3", qui est *vraie* si et seulement si φ_1 est continuellement *vraie* dans tout intervalle de temps minimal débutant quand φ_2 est *vraie* et finissant quand φ_3 est *vraie*. Par convention, si φ_3 n'est jamais *vraie* après que φ_2 ait été *vraie*, φ_1 est supposée être respectée pour toujours. En d'autres termes, à tout instant, ou bien φ_3 doit avoir été *vraie* au moins une fois depuis le dernier instant où φ_2 était *vraie*, ou φ_1 doit avoir été continuellement *vraie* depuis le dernier instant où φ_2 était *vraie*:

$$(\sigma_0, \ldots, \sigma_n, \ldots) \models \textit{always } \varphi_1 \textit{ from } \varphi_2 \textit{ to } \varphi_3 \quad \text{ssi}$$
$$\forall i \geq 0, \quad \text{ou bien} \quad (\sigma_0, \ldots, \sigma_i) \models \textit{once } \varphi_3 \textit{ since } \varphi_2$$
$$\text{ou} \quad (\sigma_0, \ldots, \sigma_i) \models \textit{always } \varphi_1 \textit{ since } \varphi_2$$

La sortie du nœud suivant est toujours **true** si et seulement si ses paramètres d'entrée représentent des propriétés φ_1, φ_2 et φ_3 satisfaisant "*always* φ_1 *from* φ_2 *to* φ_3" :

```
node Always_from_to_(phi1,phi2,phi3:  bool)
        returns (always_phi1_from_phi2_to_phi3:  bool);
let
   always_phi1_from_phi2_to_phi3 =
              Once_since(phi3,phi2) or Always_since(phi1,phi2);
tel
```

L'opérateur "Not... between... and...": "*not* φ_1 *between* φ_2 *and* φ_3" est *vraie* si et seulement si φ_1 n'est jamais *vraie* dans un intervalle de temps minimal débutant quand φ_2 est *vraie* et finissant quand φ_3 est *vraie*:

$$(\sigma_0,\ldots,\sigma_n,\ldots) \models \textit{not } \varphi_1 \textit{ between } \varphi_2 \textit{ and } \varphi_3 \text{ ssi}$$
$$\forall i \geq 0,\quad (\sigma_0,\ldots,\sigma_i) \models \varphi_3 \;\Rightarrow\; (\sigma_0,\ldots,\sigma_i) \models \textit{always } \neg\varphi_1 \textit{ since } \varphi_2$$

Ce qui conduit à définir le programme suivant:

```
node Not_between_and_(phi1,phi2,phi3:  bool)
        returns(not_phi1_between_phi2_and_phi3:  bool);
let
   not_phi1_between_phi2_and_phi3 =
              Implies(phi3, Always_since_(not phi1, phi2));
tel
```

L'opérateur "Alternating": Une trace satisfait "*alternating* φ_1 φ_2" si et seulement elle consiste en une alternance de périodes où φ_1 est *vraie* et de périodes où φ_2 est *vraie*, éventuellement séparées par des périodes où aucune d'elle n'est *vraie*. De plus, une période où φ_1 est *vraie* doit d'abord se produire (cf. Fig. 2.1):

$$\begin{aligned}(\sigma_0,\ldots,\sigma_n,\ldots) \models \textit{alternating } \varphi_1\ \varphi_2 \quad \text{ssi} \quad & \exists (j_p)_{p\geq 0},(k_p)_{p\geq 0},(m_p)_{p\geq 0},(n_p)_{p\geq 0},\\ \text{satisfaisant} \quad & j_0 = 0,\\ \text{et} \quad & \forall p \geq 0,\ j_p \leq k_p < m_p \leq n_p < j_{p+1}\\ \text{et} \quad & j_p \leq i < k_p \implies (\sigma_0,\ldots,\sigma_i) \models \neg\varphi_1 \wedge \neg\varphi_2\\ \text{et} \quad & k_p \leq i < m_p \implies (\sigma_0,\ldots,\sigma_i) \models \varphi_1 \wedge \neg\varphi_2\\ \text{et} \quad & m_p \leq i < n_p \implies (\sigma_0,\ldots,\sigma_i) \models \neg\varphi_1 \wedge \neg\varphi_2\\ \text{et} \quad & n_p \leq i < j_{p+1} \implies (\sigma_0,\ldots,\sigma_i) \models \neg\varphi_1 \wedge \varphi_2\end{aligned}$$

En d'autres termes, à chaque fois que φ_1 devient *fausse* (c'est-à-dire sur chaque front descendant), elle ne doit plus être *vraie* à nouveau avant que φ_2 devienne *vraie* puis redevienne *fausse*, et inversement. On obtient le programme suivant:

```
node Alternating(phi1,phi2:  bool) returns (alternating:  bool);
var phi1_forbidden, phi2_forbidden:  bool;
let
   alternating =  not(phi1 and phi1_forbidden) and
                  not(phi2 and phi2_forbidden);
   phi1_forbidden = false -> if Edge(not phi1) then true
                             else if Edge(not phi2) then false
                             else pre(phi1_forbidden);
   phi2_forbidden = true -> if Edge(not phi2) then true
                             else if Edge(not phi1) then false
                             else pre(phi2_forbidden);
tel
```

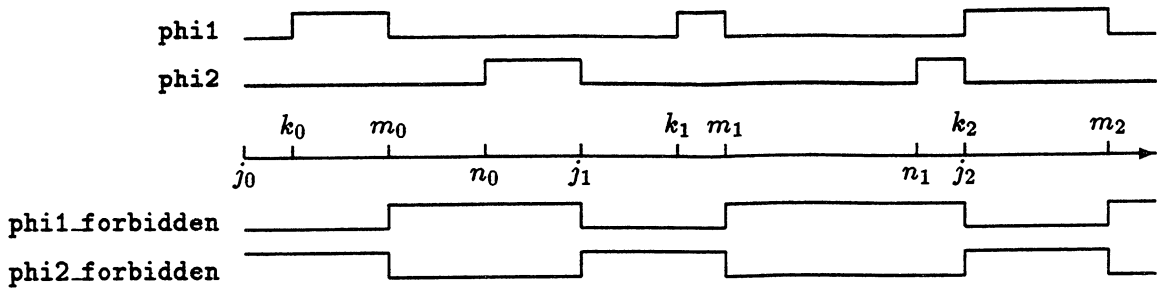


Figure 2.1: Variables booléennes alternantes

2.5 Exemple de spécification d'un système

A titre d'exemple, nous allons exprimer en LUSTRE les spécifications du système GOST décrit dans la section 1.4. Il s'agit donc d'exprimer les propriétés de sûreté qui garantissent que le système évitera *toujours* qu'un accident se produise à l'intérieur de la section.

Le premier risque d'accident concerne la collision des rames. Il faut contrôler qu'une rame n'est autorisée à pénétrer dans la section qu'à la seule condition que cette dernière soit vide. Cette propriété s'exprime en LUSTRE par l'invariance d'une variable booléenne `non_collision`, définie comme suit:

```
non_collision = Implies(autoriser_entree,section_vide);
```

A partir de l'équation définissant `autoriser_entree`, cette propriété est trivialement vérifiée. Par conséquent, en supposant que les rames ne pénètrent jamais dans la section par la voie C, il est dés lors possible de considérer que la section n'est jamais empruntée par plus d'un train à la fois.

Il ne reste plus maintenant qu'à contrôler que toute rame pénétrant dans la section par la voie A la quittera toujours par la voie C, et que les déraillements sont impossibles. Cela conduit donc à vérifier que l'aiguillage est correctement commandé. En premier lieu, il est clair que les requêtes de modification de positionnement de l'aiguillage ne doivent pas être actives en même temps. Cela s'exprime simplement par l'invariance de l'expression LUSTRE suivante:

```
exclusive_req = not(faire_AB and faire_BC);
```

D'autre part, l'aiguillage doit continuellement connecter A avec B à partir du moment où une rame est autorisée à pénétrer dans la section, et jusqu'à ce qu'elle se trouve en transit sur la voie B. Cela s'exprime par l'équation suivante, qui fait appel à l'opérateur temporel `Always_from_to`:

```
non_derail_AB = Always_from_to(connect_AB,autoriser_entree,transit_sur_B);
```

De façon similaire, l'aiguillage doit continuellement connecter B avec C à partir du moment où une rame est autorisée à quitter la section, et jusqu'à ce que cette dernière soit à nouveau vide:

```
non_derail_BC = Always_from_to(connect_BC,autoriser_sortie,section_vide);
```

On peut remarquer que si la propriété ci-dessus est vérifiée, une rame ne peut pas quitter la section par la voie A.

Finalement, la spécification globale que le système GOST doit vérifier est exprimée par l'équation LUSTRE suivante:

```
specification = non_collision and
                exclusive_req and
                non_derail_AB and
                non_derail_BC;
```

Chapitre 3

Description comportementale de l'environnement

La troisième entité s'intégrant dans la vérification formelle concerne la description comportementale de l'environnement des systèmes. Les systèmes réactifs en particulier doivent continuellement *réagir* à leur environnement. Selon la façon dont ce dernier se comporte, leur comportement sera différent. Or, l'environnement ne se comporte pas de manière anarchique: son évolution est généralement soumise à des contraintes (qui sont souvent physiques).

La simulation exhaustive exclut de pouvoir vérifier un système dans son environnement réel. Par conséquent, ce dernier doit être *modélisé*. En particulier, son comportement doit être *simulé*, de telle sorte que la vérification ne s'effectue que par rapport à des conditions réalistes de fonctionnement du système. Toutefois, il n'est pas gênant de vérifier le système pour des comportements irréalistes de l'environnement, du moment que la prise en compte de ces derniers n'empêche pas la vérification d'aboutir. Il n'est donc pas nécessaire de donner une description comportementale complète et détaillée de l'environnement, mais simplement de décrire ce qu'il est *nécessaire* de connaître pour permettre de réaliser la vérification.

Il apparaît donc nécessaire de disposer d'un formalisme de description comportementale de l'environnement des systèmes, venant s'intégrer aux programmes et aux spécifications pour effectuer la vérification. Dans ce chapitre, nous discutons de la méthode employée dans ce but en LUSTRE.

3.1 Utilisation des assertions

L'environnement d'un programme LUSTRE est modélisé par les valeurs de ses variables d'entrée, dont l'évolution constitue un comportement. Il est donc possible d'exprimer sous forme d'expressions LUSTRE sur ces variables *certaines* propriétés de ce comportement, dont seul l'opérateur humain a connaissance. Ces expressions peuvent alors être introduites dans les programmes sous forme d'hypothèses, grâce aux assertions qui apportent une solution simple et immédiate à cette nécessité. De façon informelle, on peut distinguer plusieurs types d'assertions:

- Les assertions "instantanées", dont l'expression ne contient pas l'opérateur **pre**. Elles décrivent des hypothèses sur la simultanéité entre certains événements internes ou externes

au programme.

- Les assertions "temporelles", dans lesquelles l'opérateur **pre** apparaît. Elles décrivent des hypothèses d'ordonnancement entre certains événements internes ou externes au programme.
- Les assertions portant spécifiquement sur l'environnement du programme, dans lesquelles n'apparaissent que des variables d'entrée de celui-ci. Elles décrivent des hypothèses sur le comportement intrinsèque de l'environnement.
- Les assertions portant sur le programme et son environnement. Dans ce cas, l'expression contient à la fois des variables d'entrée est des variables de sortie de celui-ci. Elles permettent de décrire des hypothèses sur les interactions entre le comportement du programme et celui de son environnement.
- Les assertions portant sur le comportement du programme. Ce sont celles qui font apparaître des variables internes de celui-ci. Ces assertions sont dangereuses car elles peuvent restreindre *arbitrairement* le comportement du programme. En pratique, il faut les utiliser avec précaution, car elles permettent d'introduire l'indéterminisme dans les programmes.

Remarquons que ces assertions ont la même sémantique dans les programmes. Un exemple d'utilisation des assertions est abordé dans la section suivante. Dans celui-ci, divers types d'assertions sont utilisées parmi ceux qui viennent d'être recensés.

3.2 Exemple

A titre d'exemple, nous reprenons le système GOST décrit dans la section 1.4. Il est indispensable d'exprimer sous forme d'assertions LUSTRE les hypothèses qui régissent les interactions entre le système et son environnement. En effet, si celles-ci ne sont pas prises en considération, il est très probable que les spécifications ne soient jamais vérifiées. Il est clair que l'environnement n'a pas un comportement aléatoire. Ainsi, dans la section de métro considérée, les rames sont supposées s'arrêter quand les feux sont au rouge. La validation serait certainement impossible sans tenir compte d'une telle information. De ce fait, il est nécessaire de pouvoir définir des hypothèses sur le comportement de l'environnement.

Nous décrivons ici certaines caractéristiques importantes de l'environnement du système GOST. A propos de l'aiguillage, les hypothèses suivantes peuvent être faites:

- L'aiguillage ne peut pas à la fois connecter A avec B et B avec C. Cela se traduit par l'assertion instantanée suivante:

```
assert not(connect_AB and connect_BC);
```

- A partir du moment où il est dans une position donnée, l'aiguillage ne change pas de position spontanément, à moins que le système émette une commande lui ordonnant d'atteindre la position opposée à celle couramment occupée:

```
assert Always_from_to(connect_AB,connect_AB,faire_BC) and
       Always_from_to(connect_BC,connect_BC,faire_AB);
```

Cette assertion décrit une interaction entre le programme et son environnement.

A propos des mouvements de rames dans la section, les hypothèses sont les suivantes:

- Initialement, il n'y a pas de rame dans la section.

```
assert section_vide -> true;
```

- Les trains respectent les feux de signalisation. Cela peut se traduire par le fait qu'à l'instant où une rame entre dans la section, alors le feu de signalisation correspondant était vert à l'instant précédent. On peut faire le même raisonnement pour une rame sortant de la section. Remarquons que les expressions ci-dessous font appel au nœud **Edge**, défini dans la section 1.2:

```
assert true -> Implies(Edge(not section_vide), pre autoriser_entree);
assert true -> Implies(Edge(sur_C), pre autoriser_sortie);
```

- Quand un train quitte A, il se retrouve sur B. Quand un train quitte B, il se retrouve soit sur A, soit sur C. Remarquons que ces hypothèses traduisent les lois physiques qui empêchent toute rame de disparaître subitement de la section sur laquelle elle se trouve.

```
assert Implies(Edge(not sur_A), sur_B);
assert Implies(Edge(not sur_B), sur_A or sur_C);
```

Ces assertions sont spécifiques au comportement de l'environnement.

3.3 Critiques

Dans le cadre de la vérification, les assertions sont utilisées pour exprimer les hypothèses sur le comportement intrinsèque de l'environnement et sur les interactions entre les programmes et leur environnement. Toutefois, ces hypothèses ne peuvent être décrites que sous la forme d'expressions invariantes. Cela est une contrainte, et il est évident que les assertions ont un pouvoir d'expression trop faible pour permettre quelquefois d'exprimer tout ce qui serait nécessaire. Elles ne fournissent donc qu'un moyen de description *partiel.*

De plus, les assertions sont délicates à utiliser, car elles doivent restreindre les comportements possibles de l'environnement de façon suffisante mais pas excessive. En effet, si elles ne sont pas assez restrictives, les spécifications risquent de ne jamais vérifiées par les programmes (l'exemple précédent en est une parfaite illustration). Toutefois, il n'est pas toujours nécessaire de décrire complètement le comportement de l'environnement, mais une description partielle de celui-ci peut suffire pour vérifier les spécifications. A l'inverse, si ces spécifications sont trop restrictives, certains comportements réalistes de l'environnement seront éliminés. Dans ce cas, la vérification formelle perd tout son sens, puisqu'il devient possible de vérifier des spécifications qui ne seront pas respectées en réalité! Le problème soulevé ici relève du principe même de la vérification formelle: la simulation exhaustive des programmes nécessite la simulation exhaustive de leur environnement. Or, cette dernière n'est possible qu'à condition de fournir une description adéquate de la manière dont il se comporte. Les assertions LUSTRE fournissent une intéressante solution à ce problème, mais elles possèdent un pouvoir d'expression limité. Les expériences menées pour la vérification en LUSTRE démontrent à ce sujet que l'aspect le plus difficile de la vérification formelle des programme concerne l'écriture des assertions.

Chapitre 4

Définition d'un principe de vérification

En LUSTRE, les trois entités mises en jeu dans toute vérification (réalisation, spécifications, assertions) peuvent être exprimées dans le même formalisme de description. Cette particularité permet de définir un principe de vérification original des programmes LUSTRE, qui est décrit ci-dessous.

4.1 Programme de vérification

Etant donné un programme LUSTRE P, nous avons vu que les spécifications S et les assertions A sur P s'expriment sous forme d'expressions LUSTRE sur les variables de P. Il est donc possible de rassembler sans difficulté P, S et A au sein d'un nouveau programme P_{verif} appelé *programme de vérification*, dont les caractéristiques sont les suivantes:

- Les entrées de P_{verif} sont identiques à celles de P.
- L'*unique* sortie de P_{verif} correspond à l'expression des spécifications S de P.

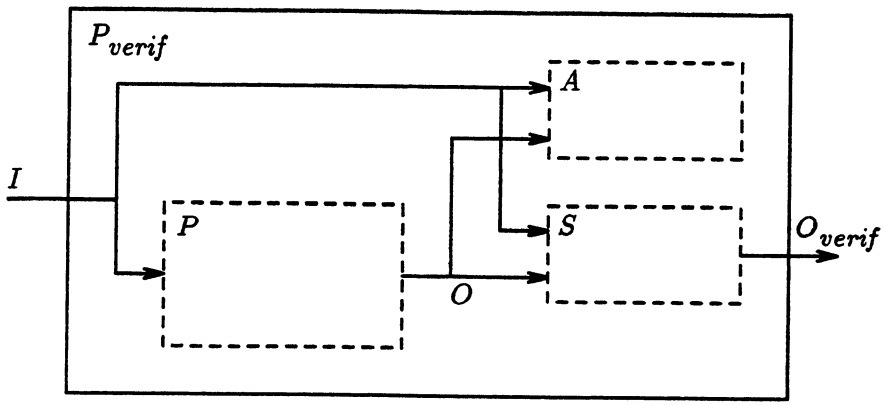


Figure 4.1: Le programme de vérification

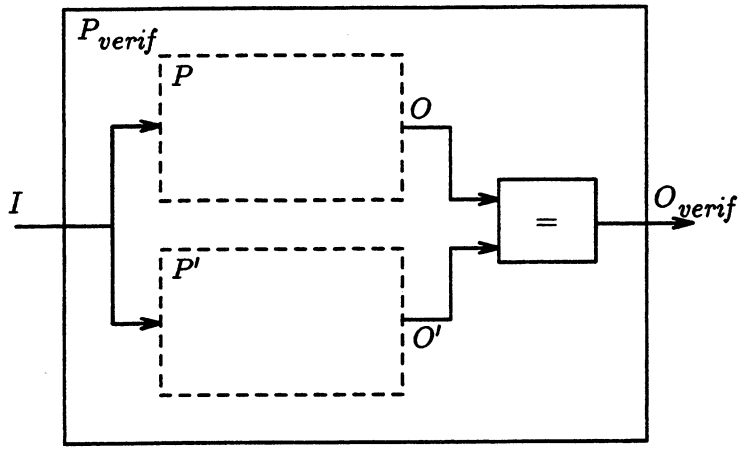


Figure 4.2: La comparaison de programmes

- Les hypothèses A sur le comportement de l'environnement sont exprimées sous forme d'assertions dans P_{verif}.

En dénotant respectivement par I les variables d'entrée et par O les variables de sortie et les variables internes de P, la construction de P_{verif} est schématisée par la figure 4.1. Dans cette figure, chaque bloc correspond à un ensemble d'équations LUSTRE, et les flèches qui les relient matérialisent les dépendances entre variables et expressions. Intuitivement, la vérification formelle de P pourrait alors consister à simuler *exhaustivement* le programme P_{verif}, afin de vérifier que sa sortie garde toujours la valeur *vraie*. Si c'est la cas, alors on peut en déduire que P vérifie ses spécifications.

4.1.1 Comparaison de programmes

Un cas particulier et néanmoins courant de la vérification consiste à comparer deux programmes implémentant les mêmes spécifications. La vérification consiste alors à contrôler que les programmes ont toujours le même comportement. Intuitivement, cela revient à les exécuter en parallèle afin d'observer que leurs sorties sont toujours identiques. Ce genre de vérification s'applique très bien aux programmes LUSTRE. En effet, étant donnés deux programmes P et P' dont on veut comparer les comportements, on peut les intégrer au sein d'un programme de vérification P_{verif}, construit selon le schéma de la figure 4.2. A partir d'un tel programme, on peut extraire un modèle d'exécution pour la vérification.

4.1.2 Exemple

Nous reprenons à nouveau comme exemple le système GOST traité dans la section 1.4. Pour cet exemple, nous avons successivement exprimé en LUSTRE:

- Un programme réalisant le système GOST.

- Les spécifications du système, qui sont des propriétés de sûreté exprimées sous la forme d'expressions booléennes sur les variables du programme.

- Les hypothèses sur le comportement de l'environnement, qui forment un ensemble d'assertions.

Ces trois entités peuvent être intégrées au sein d'un programme de vérification du système GOST:

```
node GOST_verif(sur_A, sur_B, sur_C, connect_AB, connect_BC: bool)
     returns(specification: bool);
var
   autoriser_entree,autoriser_sortie: bool;
   faire_AB, faire_BC: bool;
   non_collision, exclusive_req: bool;
   non_derail_AB, non_derail_BC: bool;
   section_vide, transit_sur_B: bool;
let
   section_vide = not(sur_A or sur_B or sur_C);
   transit_sur_B = sur_B and not(sur_A or sur_C);

   -- ASSERTIONS
   assert not(connect_AB and connect_BC);
   assert Always_from_to(connect_AB,connect_AB,faire_BC)
      and Always_from_to(connect_BC,connect_BC,faire_AB);
   assert section_vide -> true;
   assert true ->
          Implies(Edge(not section_vide),
                  pre autoriser_entree);
   assert true ->
          Implies(Edge(sur_C),
                  pre autoriser_sortie);
   assert Implies(Edge(not sur_A),sur_B);
   assert Implies(Edge(not sur_B),
                  sur_A or sur_C);

   -- APPEL DU NOEUD GOST
   (autoriser_entree,autoriser_sortie,faire_AB,faire_BC) =
        GOST(sur_A,sur_B,sur_C,connect_AB,connect_BC);

   -- SPECIFICATIONS
   non_collision =
        Implies(autoriser_entree,section_vide);
   exclusive_req =
        not(faire_AB and faire_BC);
   non_derail_AB =
         Always_from_to(connect_AB,
                        autoriser_entree,
```

```
                        transit_sur_B);
   non_derail_BC =
         Always_from_to(connect_BC,
                        autoriser_sortie,
                        section_vide);
   specification =
        non_collision and exclusive_req and
        non_derail_AB and non_derail_BC;
tel
```

Figure 4.1.2: Le programme de vérification du système GOST

4.2 Principe d'un outil de vérification

La faculté offerte par LUSTRE de pouvoir intégrer les entités de la vérification au sein d'un même programme permet de simplifier considérablement la mise en œuvre de la vérification. En effet, vérifier que les spécifications d'un programme forment un invariant de celui-ci revient alors simplement à contrôler que son unique sortie s'évalue toujours à la valeur *vraie*.

4.3 Conclusion

Dans cette partie, nous avons explicité les trois entités de la vérification en LUSTRE: le programme, ses spécifications et la description comportementale de son environnement. De celles-ci, seule la première était fixée au départ. Les deux autres ont du être définies par rapport à notre approche de la vérification. Cela a abouti sur un concept "monolangage", dans lequel LUSTRE est l'unique langage de description des trois entités. Les avantages méthodologiques et ergonomiques de ce résultat ont été mis en évidence.

D'autre part, la sémantique formelle des trois entités de la vérification a été donnée grâce à la sémantique des traces de LUSTRE, qui fournit un cadre formel simple et unificateur en terme de comportements des programmes.

Enfin, un principe de vérification original des programmes LUSTRE a été dégagé. Celui-ci doit être mis en œuvre dans un outil de vérification. Cependant, il est nécessaire de définir au préalable une procédure de décision permettant d'évaluer *formellement* si un programme vérifie ses spécifications. Ce point est abordé dans la partie suivante, dans laquelle un modèle formel dénotant les comportements d'un programme LUSTRE est défini. Puis différentes méthodes de vérification des spécifications sur le modèle sont étudiées.

Partie II

Caractérisation Formelle

Chapitre 5

Modèle d'exécution des programmes Lustre

La sémantique des traces de LUSTRE permet d'obtenir une interprétation du comportement des programmes en termes de traces d'exécution. En se basant sur une nouvelle interprétation de cette sémantique, les traces peuvent être représentées dans un *modèle d'exécution* dénotant l'ensemble des comportements du programme. Il est possible de caractériser formellement sur un tel modèle les traces exécutables, valides ou correctes d'un programme.

Dans ce chapitre, nous définissons formellement le modèle associé à tout programme LUSTRE et dénotant l'ensemble de ses comportements. Puis nous définissons un certain nombre de notations et d'opérateurs sur ce modèle, qui permettront ultérieurement de manipuler des ensembles de traces d'exécution. La sémantique des assertions et des spécifications sera alors caractérisée formellement en termes de *séquences d'exécution* sur ce modèle.

5.1 Système de transition associé à un programme Lustre

Le modèle associé à un programme LUSTRE est un système de transition, dont les différents composants sont décrits ci-dessous.

A partir de la sémantique des traces de LUSTRE, on peut définir une *relation de transition* notée $\rightarrow$ entre les traces d'exécution d'un programme. Cette relation dépend du programme à partir duquel elle est définie. Etant donné un programme P, sa définition est la suivante:

$$\forall\, \Sigma,\ \forall\, \sigma,\ \Sigma \rightarrow \Sigma.\sigma \iff \Sigma \vdash P \text{ et } \Sigma.\sigma \vdash P$$

L'ensemble des traces muni de cette relation définit un système de transition dénotant l'ensemble des comportements de P. Par définition, les traces forment les états de ce système. Celles-ci étant de longueur quelconque, les états du système et le système lui-même sont infinis, ce qui n'est pas exploitable. Par une *abstraction*, nous allons définir un système de transition *fini* caractérisant un sur-ensemble des comportements de P. Dans ce but, nous définissons ci-dessous une sémantique opérationnelle de LUSTRE.

5.1.1 Sémantique opérationnelle de Lustre

Dans la sémantique des traces de LUSTRE, la valeur des expressions dépend uniquement de la dernière mémoire constituant la trace, sauf pour l'opérateur **pre** qui peut faire intervenir ses mémoires antérieures. Nous allons définir ici une autre sémantique, où seule la mémoire précédente est nécessaire pour déduire une nouvelle mémoire compatible avec un programme. Dans ce but, nous poserons comme hypothèse que toutes les occurrences de l'opérateur **pre** dans un programme s'appliquent uniquement à des identificateurs, et jamais à des expressions plus complexes. Cette hypothèse n'est absolument pas restrictive du point de vue sémantique, car le principe de substitution de LUSTRE autorise le remplacement de toute expression du langage par une variable dont la définition correspond à cette expression. La nouvelle sémantique définit alors comment, à tout instant, une mémoire compatible avec un programme se déduit de la mémoire du programme à l'instant précédent. Le fait que pour un programme P, la mémoire σ' se déduit de la mémoire σ sera noté:

$$\sigma, \sigma' \vdash P$$

A l'instant initial, la mémoire précédente est indéfinie. Une mémoire indéfinie est dénotée par le symbole $\perp$.

La valeur de toute expression LUSTRE E sur une mémoire σ' sachant que la mémoire précédente est σ est définie par induction sur la structure de E. Le fait que E prend la valeur v sera alors noté:

$$\sigma, \sigma' \vdash \mathtt{E} \mid v$$

Les règles suivantes permettent de définir la valeur de E:

Valeur d'une constante:
Le symbole k dénotant toute constante du langage,

$$\sigma, \sigma' \vdash \mathtt{k} \mid k$$

Valeur d'une variable:
Le symbole x dénotant tout identificateur du programme,

$$\sigma, \sigma' \vdash \mathtt{x} \mid \sigma'(\mathtt{x})$$

Valeur de $\star$ (E$_1$, ..., E$_m$):
E$_1$,...,E$_m$ dénotant des expressions quelconques du langage, et $\star$ dénotant tout opérateur arithmétique, booléen ou conditionnel standard,

$$\frac{\sigma, \sigma' \vdash \mathtt{E}_1 \mid v_1 \ , \ \ldots \ , \ \sigma, \sigma' \vdash \mathtt{E}_m \mid v_m}{\sigma, \sigma' \vdash \star\,(\mathtt{E}_1, \ \ldots, \ \mathtt{E}_m) \mid \star(v_1, \ldots, v_m)}$$

Valeur de E$_1$ -> E$_2$:
E$_1$ et E$_2$ dénotant deux expressions quelconques du langage,

$$\frac{\perp, \sigma' \vdash \mathtt{E}_1 \mid v_1}{\perp, \sigma' \vdash \mathtt{E}_1 \ \texttt{->} \ \mathtt{E}_2 \mid v_1} \qquad \frac{\sigma, \sigma' \vdash \mathtt{E}_2 \mid v_2}{\sigma, \sigma' \vdash \mathtt{E}_1 \ \texttt{->} \ \mathtt{E}_2 \mid v_2}$$

Valeur de pre(x):
x dénotant tout identificateur du programme, et *nil* dénotant une valeur *indéterminée*,

$$\bot, \sigma' \vdash \texttt{pre(x)} \mid nil \qquad \sigma, \sigma' \vdash \texttt{x} \mid \sigma(\texttt{x})$$

La première règle détermine la valeur de l'expression pre(x) à l'instant initial, la seconde la détermine à n'importe quel autre instant.

Nous définissons maintenant la compatibilité d'une mémoire σ' avec un programme P sachant qu'à l'instant précédent la mémoire était σ, qui est notée:

$$\sigma, \sigma' \vdash P$$

Les règles définissant cette compatibilité sont les suivantes:

Compatibilité avec les équations:

$$\frac{\sigma, \sigma' \vdash \texttt{E} \mid v,\ \sigma'(\texttt{x}) = v}{\sigma, \sigma' \vdash \texttt{x = E}}$$

Compatibilité avec la composition:

$$\frac{\sigma, \sigma' \vdash P_1,\ \sigma, \sigma' \vdash P_2}{\sigma, \sigma' \vdash P_1; P_2}$$

Compatibilité avec les assertions:

$$\frac{\sigma, \sigma' \vdash \texttt{A} \mid true}{\sigma, \sigma' \vdash \texttt{assert A}}$$

5.1.2 Abstraction booléenne

A partir de la sémantique définie précédemment, on peut définir un nouveau système de transition sur les mémoires d'un programme P, dont la relation de transition $\rightarrow$ est définie comme suit:

$$\forall\, \sigma,\ \forall\, \sigma',\ \sigma \rightarrow \sigma' \iff \sigma, \sigma' \vdash P$$

Le système de transition obtenu reste cependant encore infini. Toutefois, on peut se ramener à un système fini par une abstraction booléenne sur les états de celui-ci. Cette abstraction est issue d'une interprétation abstraite [CC77] des programmes, qui est décrite ci-dessous.

Dans un premier temps, on va partitionner les mémoires des programmes en mémoire "booléenne" et mémoire "non booléenne". En reprenant la définition des mémoires donnée dans la section 1.3, nous allons d'abord partitionner l'ensemble I des identificateurs d'un programme et l'ensemble V des valeurs prises par ces identificateurs de la manière suivante:

$$I = B + J$$

où B est l'ensemble des identificateurs de type booléen, et J est l'ensemble des identificateurs d'un autre type.

$$V = \{0,1\} + W$$

où $\{0,1\}$ dénote l'ensemble des valeurs booléennes, et où W est l'ensemble des autres valeurs.

Toute mémoire σ est encore une fonction de I dans V, mais elle peut être maintenant définie par un nouveau couple de mémoire (σ_B, σ_J), où σ_B est une fonction de B dans $\{0,1\}$ et σ_J est une fonction de J dans W. σ_B est la mémoire booléenne du programme tandis que σ_J est sa mémoire non booléenne. L'interprétation abstraite consiste alors à définir l'abstraction booléenne α d'une mémoire comme suit:

$$\forall\ \sigma, \alpha(\sigma) = \sigma_B$$

Intuitivement, cette interprétation consiste à "projeter" le programme dans le domaine booléen. Réciproquement, la concrétisation γ d'une mémoire booléenne est définie par:

$$\forall\ \sigma_B, \gamma(\sigma_B) = \{(\sigma_B, \sigma_J)\ /\ \sigma_J \in [J \mapsto W]\}$$

Finalement, on peut définir un système de transition sur les mémoires booléennes d'un programme, dont la relation de transition $\xrightarrow{B}$ est la suivante:

$$\forall\ \sigma,\ \forall\ \sigma',\ \sigma \xrightarrow{B} \sigma' \Longleftrightarrow \exists\ \sigma \in \gamma(\sigma_B),\ \exists\ \sigma' \in \gamma(\sigma'_B)\ /\ \sigma \rightarrow \sigma'$$

Il est alors évident que l'implication ci-dessous est vérifiée:

$$\sigma \rightarrow \sigma' \Rightarrow \alpha(\sigma) \xrightarrow{B} \alpha(\sigma)$$

Ce système est une machine d'états finis car la dimension et le nombre des mémoires booléennes sont finis (le nombre d'identificateurs booléens contenus dans le programme est fini, donc le nombre de valeurs différentes des mémoires est fini aussi). Un tel système modélise tous les comportements booléens du programme.

Par la suite, le système de transition associé à tout programme LUSTRE sera indifféremment appelé *système* (de transition), *machine* (d'états finis) ou *modèle* (d'exécution).

5.1.3 Séquences d'exécution

Tous les comportements d'un programme P sont définis formellement par une trace d'exécution compatible avec celui-ci. Or, la relation de transition 5.1 est définie de telle sorte que:

$$\forall\ \Sigma = (\sigma_0, \ldots, \sigma_n, \ldots), \Sigma \vdash P \Longleftrightarrow \sigma_0 \rightarrow \sigma_1 \rightarrow \ldots \sigma_n \rightarrow \ldots$$

Par conséquent:

$$\forall\ \Sigma = (\sigma_0, \ldots, \sigma_n, \ldots),\ \sigma_0 \rightarrow \sigma_1 \rightarrow \ldots \sigma_n \rightarrow \ldots \Rightarrow \alpha(\sigma_0) \xrightarrow{B} \alpha(\sigma_1) \xrightarrow{B} \ldots \alpha(\sigma_n) \xrightarrow{B} \ldots$$

La séquence $[\alpha(\sigma_0), \ldots, \alpha(\sigma_n), \ldots]$ forme donc une *séquence d'exécution* du système de transition associé au programme.

5.2 Le modèle final

Concrètement, à partir du système de transition défini précédemment, on peut obtenir pour tout programme LUSTRE un modèle du type machine d'états finis $\mathcal{M} = (Q, E, S, q_{init}, \alpha, \sigma, \vec{\delta})$ défini ainsi:

- Q est l'ensemble des états, correspondant à la mémoire booléenne du programme.
- E est l'ensemble des entrées, correspondant aux entrées booléennes du programme.
- S est l'ensemble des sorties, correspondant aux sorties booléennes du programme.
- q_{init} est l'état initial, correspondant à l'état initial des programme.
- $\alpha \in [Q \times E] \longmapsto \{0,1\}$ est la fonction d'assertion.
- $\sigma \in [Q \times E] \longmapsto \{0,1\}^{|S|}$ est la fonction de sortie.
- $\vec{\delta} \in [Q \times E] \longmapsto Q$ est la fonction de transition.

La machine définie ci-dessus se différencie d'une machine d'états finis classique à cause de la fonction d'assertion qui entre dans sa définition, et qui en restreint les comportements.

Une *séquence d'exécution* de $\mathcal{M}$ correspond à toute séquence de transitions (q_0, e_0), (q_1, e_1), $\ldots$,(q_n, e_n), $\ldots$ telle que $\forall i \in N,\ q_{i+1} = \vec{\delta}(q_i, e_i)$.

Afin d'illustrer les calculs effectués ultérieurement sur les modèles de programmes LUSTRE, des schémas sont insérés dans la suite de l'ouvrage. Le modèle est dans ce cas représenté de manière classique par un *graphe d'états* orienté, dont les nœuds symbolisent les états et les arcs représentent les transitions.

5.3 Opérateurs sur le modèle

Le modèle obtenu permet de décrire l'ensemble des comportements d'un programme en termes de séquences d'exécution. La vérification d'un programme nécessite alors de générer et de manipuler ces séquences à l'aide d'opérateurs sur les ensembles d'états et de transitions du modèle. Nous donnons ci-dessous leur définition formelle ainsi que leurs principales propriétés. Les opérateurs définis ici sont similaires aux opérateurs standard classiquement définis sur les systèmes de transitions, mais qui opèrent plutôt sur des ensembles d'états.

5.3.1 Opérateurs sur des ensembles d'états

Nous définissons ici deux opérateurs unaires externes ayant pour opérande un ensemble d'états et pour résultat un ensemble de transitions.

Etant donné un ensemble quelconque $X \subset Q$ d'états d'un modèle, les transitions *exécutables* depuis X sont définies par l'opérateur *Exec* :

$$Exec : \begin{cases} \mathcal{P}(Q) & \longmapsto & \mathcal{P}(Q \times E) \\ X & \longrightarrow & Exec(X) = X \times E \end{cases}$$

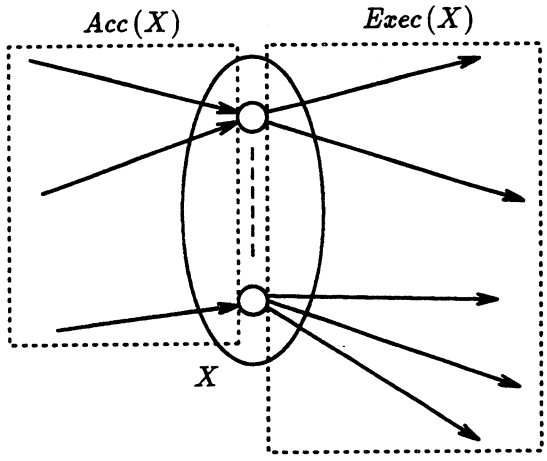


Figure 5.1: Opérateurs sur les états du modèle

L'opérateur *Exec* est monotone pour l'inclusion:

$$\forall X \subset Q,\ \forall Y \subset Q,\ X \subset Y \Rightarrow Exec(X) \subset Exec(Y)$$

Réciproquement, l'ensemble des transitions *accédant* à X est défini par l'opérateur *Acc*:

$$Acc : \begin{cases} \mathcal{P}(Q) & \longmapsto & \mathcal{P}(Q \times E) \\ X & \longrightarrow & Acc(X) = \{(q,e) \in Q \times E,\ \vec{\delta}(q,e) \in X\} \end{cases}$$

L'opérateur *Acc* possède les propriétés suivantes:

$$\forall X \subset Q,\ \forall Y \subset Q,\ X \subset Y \Rightarrow Acc(X) \subset Acc(Y)$$

$$\forall X \subset Q,\ \forall Y \subset Q,\ Acc(X \cup Y) = Acc(X) \cup Acc(Y)$$

$$\forall X \subset Q,\ \forall Y \subset Q,\ Acc(X \cap Y) = Acc(X) \cap Acc(Y)$$

La figure 5.1 schématise les ensembles de transitions caractérisées par les opérateurs *Exec* et *Acc*.

5.3.2 Opérateurs sur des ensembles de transitions

Nous définissons ici deux opérateurs ayant pour opérande un ensemble de transitions, et dont le résultat est un ensemble d'états.

Etant donné un ensemble quelconque $T \subset Q \times E$ de transitions du modèle, on définit l'ensemble $Src(T)$ des états à partir desquels il est *possible* d'exécuter une transition appartenant à T. Ces états forment des "sources existentielles" de T:

$$Src : \begin{cases} \mathcal{P}(Q \times E) & \longmapsto & \mathcal{P}(Q) \\ T & \longrightarrow & Src(T) = \{q \in Q \ /\ \exists e \in E,\ (q,e) \in T\} \end{cases}$$

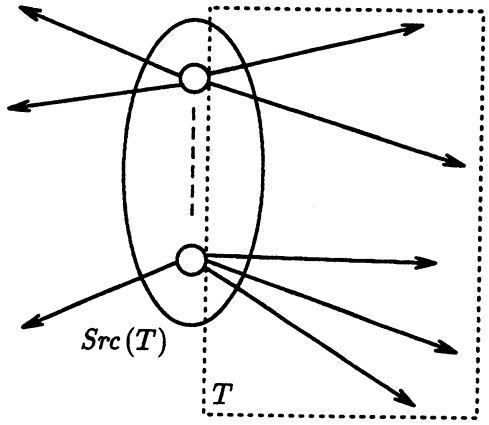


Figure 5.2: L'opérateur *Src* sur les transitions du modèle

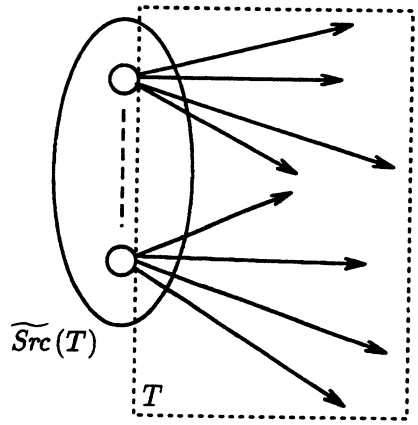


Figure 5.3: L'opérateur $\widetilde{Src}$ sur les transitions du modèle

La figure 5.3 schématise les états caractérisés par l'opérateur *Src*.

Par extension, on définit l'ensemble $\widetilde{Src}(T)$ des états à partir desquels il est *nécessaire* d'exécuter une transition appartenant à T. Ces états sont des "sources universelles" de T:

$$\widetilde{Src} : \begin{cases} \mathcal{P}(Q \times E) & \longmapsto & \mathcal{P}(Q) \\ T & \longrightarrow & \widetilde{Src}(T) = \{q \in Q \ / \ \forall e \in E, \ (q,e) \in T\} \end{cases}$$

La figure 5.3 schématise les états caractérisés par l'opérateur $\widetilde{Src}$.

5.3.3 Combinaison d'opérateurs

Les opérateurs définis précédemment forment tous des lois externes. Leur combinaison permet de définir de nouveaux opérateurs, qui sont des lois internes. Nous définissons d'abord deux

opérateurs unaires sur les ensembles de transitions, puis un opérateur unaire sur les ensembles d'états.

Etant donné un ensemble quelconque $T \subset Q \times E$ de transitions du modèle, l'ensemble des transitions *précédant* T sont toutes les transitions dont l'exécution mène à un état à partir duquel une transition de T est exécutable. A ce niveau, on distingue les transitions précédant T *existentiellement* et les transitions précédant T *universellement*. Cela permet de définir deux nouveaux opérateurs de *précondition* Pre et $\widetilde{Pre}$, obtenus par composition des opérateurs précédents:

$$Pre : \begin{cases} \mathcal{P}(Q \times E) & \longmapsto & \mathcal{P}(Q \times E) \\ T & \longrightarrow & Pre(T) = Acc(Src(T)) \end{cases}$$

De façon analogue, on a:

$$\widetilde{Pre} : \begin{cases} \mathcal{P}(Q \times E) & \longmapsto & \mathcal{P}(Q \times E) \\ T & \longrightarrow & \widetilde{Pre}(T) = Acc(\widetilde{Src}(T)) \end{cases}$$

La principale propriété de ces opérateurs est la suivante:

$$\forall T \in Q \times E,\ \widetilde{Pre}(T) = \overline{Pre(\overline{T})}$$

Nous définissons enfin l'opérateur $Post$ caractérisant les états postérieurs à un ensemble d'états quelconque $X \subset Q$ du modèle:

$$Post : \begin{cases} \mathcal{P}(Q) & \longmapsto & \mathcal{P}(Q \times E) \\ X & \longrightarrow & Post(X) = Src(Exec(X)) \end{cases}$$

On dit aussi que $Post(X)$ est l'ensemble des états accessibles à partir de X.

Chapitre 6

Caractérisation formelle sur le modèle

6.1 Evaluation des assertions

La sémantique des traces permet de caractériser la sémantique des assertions sur les traces d'exécution des programmes. Les traces infinies compatibles avec les assertions d'un programme LUSTRE forment un ensemble de traces valides. Ces traces correspondent à certaines séquences d'exécution du modèle associé au programme, qui forment par extension un ensemble de *séquences valides.* Nous donnons ici une caractérisation formelle de ces séquences.

Etant donné un modèle $\mathcal{M} = (Q, E, S, q_{init}, \alpha, \sigma, \vec{\delta})$ d'un programme P, la relation α issue des assertions de P définit l'ensemble des transitions valides de $\mathcal{M}$. On définit alors l'ensemble Q_α des états de $\mathcal{M}$ à partir desquels il est possible d'exécuter une transition valide:

$$Q_\alpha = \{q \in Q \ / \ \exists e \in E, \ (q, e) \in \alpha\}$$

Cet ensemble est la source existentielle de α: il s'exprime donc à l'aide de l'opérateur *Src* défini dans la section 5.3 par:

$$Q_\alpha = Src(\alpha)$$

Considérons alors un état $q \in Q$ tel que:

$$q \notin Q_\alpha$$

Pour un tel état, on a donc:

$$\forall e \in E, \ (q, e) \in \overline{\alpha}$$

Aucune transition n'est exécutable depuis q, donc q est un état *puits* de $\mathcal{M}$. Or $\mathcal{M}$ étant une machine réactive, les états puits de $\mathcal{M}$ sont nécessairement *inaccessibles.* α caractérise donc implicitement un ensemble d'états inaccessibles dans le modèle. Ces états correspondent formellement à l'ensemble:

$$\widetilde{Src}(\overline{\alpha})$$

Ces états étant inaccessibles, on en déduit par induction que toute transition menant à l'un d'eux est non valide. Ces transitions correspondent à l'ensemble défini par:

$$Acc(\widetilde{Src}(\overline{\alpha}))$$

Donc, de nouvelles transitions non valides apparaissent dans l'ensemble ci-dessus, ce qui donne lieu à une redéfinition de l'ensemble des transitions non valides du modèle. En appelant α_v l'ensemble des transitions valides, l'ensemble des transitions non valides est redéfini par:

$$\overline{\alpha}_v = \overline{\alpha} \cup Acc(\widetilde{Src}(\overline{\alpha}))$$

ou, en utilisant l'opérateur de précondition $\widetilde{Pre}$ défini dans la section 5.3.3:

$$\overline{\alpha}_v = \overline{\alpha} \cup \widetilde{Pre}(\overline{\alpha})$$

La mise en évidence de nouvelles transitions non valides définit implicitement un nouvel ensemble d'états puits du modèle. Par conséquent, le calcul des transitions non valides qui a été effectué à partir de α doit être itéré à α_v. Ainsi, l'ensemble des transitions non valides peut être caractérisé comme la limite de la suite $(T_n)_{n\in N}$ définie ci-dessous:

$$\begin{cases} T_0 &= \overline{\alpha} \\ T_{n+1} &= T_n \cup \widetilde{Pre}(T_n) \end{cases}$$

Cette suite est convergente car l'opérateur $\cup$ est monotone et parce que l'ensemble $Q \times E$ est fini. Sa limite correspond classiquement au plus petit point fixe de la fonction:

$$\overline{\alpha} \cup Acc(\widetilde{Src}(T))$$

En adoptant les notations du μ-calcul [Koz83], l'ensemble des assertions non valides s'exprime par:

$$\overline{\alpha}_v = \mu T.\ \overline{\alpha} \cup \widetilde{Pre}(T)$$

Par complémentation, on déduit une caractérisation des transitions valides du modèle:

$$\alpha_v = \nu V.\ \alpha \cap Pre(V)$$

6.1.1 Interprétation graphique

Les assertions définissent un ensemble de transitions non valides sur le modèle. Il est possible de donner une interprétation de cet ensemble en termes de séquences d'exécution non valides du modèle.

La sémantique des traces indique que les séquences d'exécution valides sont des séquences infinies dont chaque transition est valide. Inversement, les séquences d'exécution non valides du modèle sont donc toutes les séquences contenant une transition non valide (voir figure 6.1).

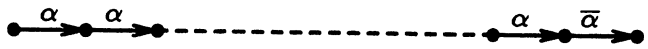


Figure 6.1: Séquence d'exécution non valide

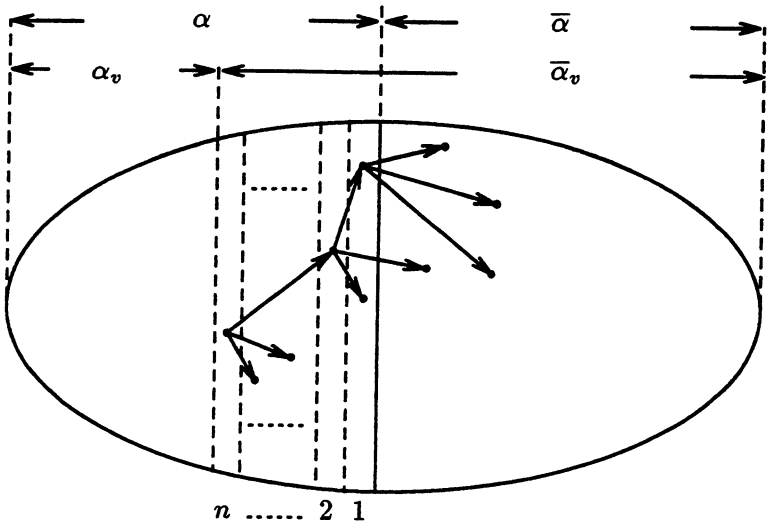


Figure 6.2: Elimination incrémentale des transitions non valides

On peut alors interpréter le point fixe consistant à calculer les transitions non valides comme étant l'élimination incrémentale des transitions dont l'exécution conduit inévitablement à exécuter une transition non valide:

- Les premières transitions de ce type sont définies à partir de α. On élimine donc toutes les transitions caractérisées par $\bar{\alpha}$.
- Ensuite, on élimine incrémentalement les transitions dont l'exécution conduit *inévitablement* à exécuter une transition éliminée.

Finalement, on obtient les états valides du modèle (cf. figure 6.2).

Considérons maintenant un état dont toutes les séquences d'exécution qui le contiennent sont non valides. Un tel état peut être schématisé par la figure 6.3. Comme cet état n'est contenu dans aucune séquence d'exécution valide, il est inaccessible bien que les assertions α sont vérifiées dans cet état. Les assertions définissent donc implicitement une restriction de l'ensemble des états accessibles du modèle.

6.1.2 Causalité des assertions

La sémantique exacte des assertions sur le modèle se heurte à l'interprétation intuitive qui leur est associée au niveau des programmes. En effet, les transitions valides du modèle forment une ensemble α_v, dont la caractérisation décrite dans la section précédente met en évidence une

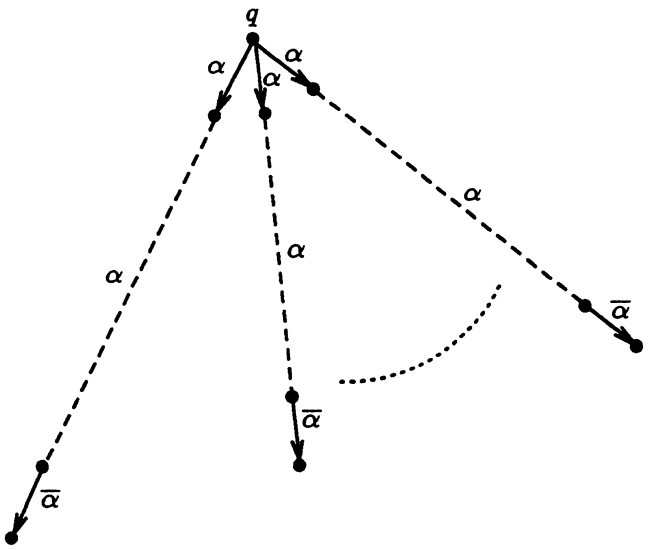


Figure 6.3: Un état non valide

dépendance de celles-ci par rapport à leur "futur". En effet, toute transition non valide est caractérisée inductivement par le fait que son exécution conduit inévitablement à exécuter *dans le futur* une transition non valide. Or, du point de vue de LUSTRE, le principe de causalité des programmes empêche que le comportement d'un programme dépende de son futur. L'interprétation intuitive des assertions consiste à considérer comme valides les assertions caractérisées par l'ensemble α. On aboutit donc d'une part à une contradiction avec le principe de causalité de LUSTRE, et d'autre part à une incompatibilité entre l'interprétation intuitive des assertions et leur sémantique exacte. C'est pourquoi lorsqu'on aboutit à de tels cas de figure, on dit que les assertions sont *non causales.* Formellement, les assertions sont causales sur un modèle si et seulement si

$$\alpha_v = \alpha$$

6.1.3 Satisfaisabilité des assertions

Les assertions caractérisent un ensemble d'états inaccessibles. Lorsque l'état initial du modèle en fait partie, le programme n'admet alors aucun comportement sous les assertions considérées. On dit alors que les assertions sont non satisfaisables. Formellement, les assertions sont satisfaisables sur un modèle si et seulement si

$$q_{init} \in Pre(\alpha_v)$$

6.2 Evaluation des spécifications

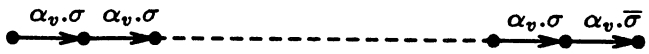


Figure 6.4: Séquence d'exécution valide mais incorrecte

Il faut maintenant évaluer les spécifications d'un programme LUSTRE sur son modèle d'exécution associé. En termes de traces d'exécution, les spécifications sont vérifiées si et seulement si toute trace d'exécution compatible avec le programme est correcte. Or, les traces compatibles sont nécessairement valides et exécutables. L'évaluation des spécifications peut alors donner lieu à deux approches:

- La première consiste à caractériser l'ensemble des séquences d'exécution valides et correctes du modèle, pour contrôler ensuite que toutes les séquences exécutables en font partie.
- La seconde consiste à caractériser toutes les séquences exécutables et valides du modèle, afin de vérifier qu'elles sont correctes.

Ces deux approches sont étudiées dans cette section. Nous nous intéressons tout d'abord à la première d'entre elles.

Nous allons donc caractériser l'ensemble des séquences à la fois correctes et valides du modèle. Ces séquences possèdent les caractéristiques suivantes:

- Elles sont infinies.
- Toutes leurs transitions sont valides.
- Toutes leurs transitions sont correctes.

Inversement, les séquences d'exécution ne vérifiant pas les caractéristiques ci-dessus sont les séquences contenant soit une transition non valide, soit une transition incorrecte. Mais, les séquences d'exécution contenant une transition non valide ne sont pas exécutables. Donc, l'évaluation des spécifications sur de telles séquences est sans intérêt. Par conséquent, il faut caractériser l'ensemble des séquences d'exécution valides mais incorrectes. Celles-ci correspondent donc à toutes les séquences finies formées de n ($n \geq 0$) transitions valides et correctes, suivies d'une transition valide mais incorrecte. Nous allons caractériser l'ensemble de ces séquences comme étant la limite d'une suite d'ensembles $(T_n)_{n \in N}$, où T_n dénote les séquences constituées d'au plus n transitions valides et correctes, suivies d'une transition valide mais incorrecte.

Etant donné un modèle $\mathcal{M} = \mathcal{M} = (Q, E, S, q_{init}, \alpha, \sigma, \vec{\delta})$, la relation σ définit l'ensemble des transitions correctes de $\mathcal{M}$. Les transitions valides du modèle sont caractérisées quant à elles par la relation α_v. Au rang $n = 0$, les séquences valides mais incorrectes forment alors un ensemble de transitions T_0 défini par:

$$T_0 = \alpha_v \cap \overline{\sigma}$$

On peut maintenant chercher l'ensemble T_1 des séquences constituées d'au plus une transition valide et correcte, suivie d'une transition appartenant à T_0. L'ensemble des états à partir desquels il est possible d'exécuter une transition de T_0 est:

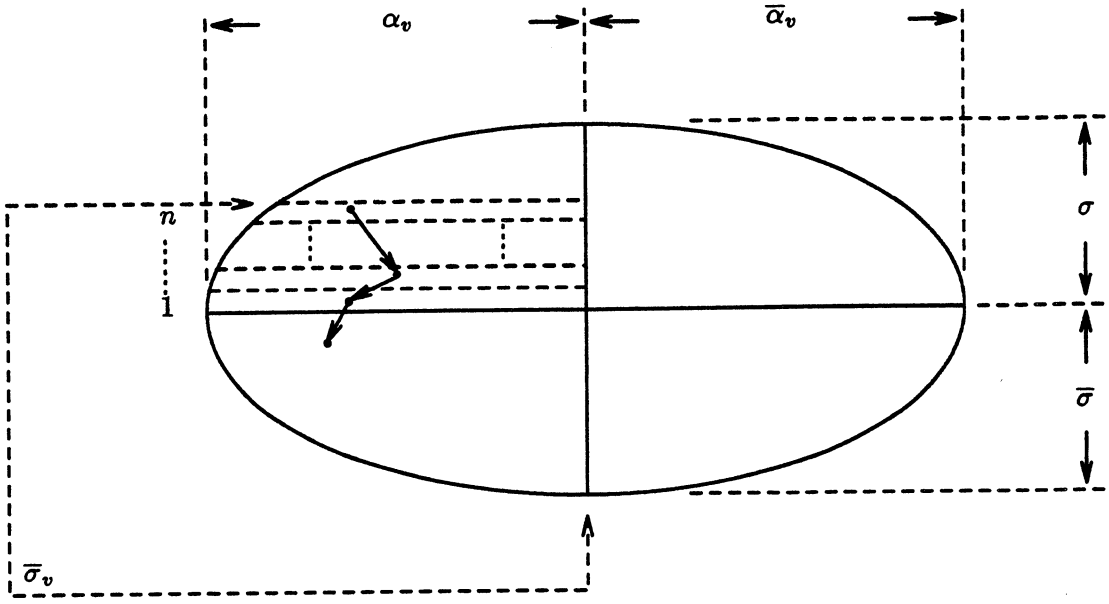


Figure 6.5: Elimination des transitions valides mais incorrectes

$Src(T_0)$

Par conséquent, les transitions dont l'exécution peut conduire à un de ces états est:

$$Acc(Src(T_0)) = Pre(T_0)$$

Ces transitions devant être de plus valides et correctes, elles appartiennent nécessairement à $\alpha_v \cap \sigma$. Ces transitions forment donc l'ensemble:

$$\alpha_v \cap \sigma \cap Pre(T_0)$$

Finalement, l'ensemble T_1 est défini par:

$$T_1 = T_0 \cup [\alpha_v \cap \sigma \cap Pre(T_0)]$$

D'où, en substituant T_0 par sa définition et en simplifiant:

$$T_1 = \alpha_v \cap (\overline{\sigma} \cup Pre(T_0))$$

T_1 définit un nouvel ensemble de transitions à partir duquel le calcul qui vient d'être effectué doit être itéré afin de générer T_2, etc... Finalement, la suite $(T_n)_{n \in N}$ est définie par:

$$\begin{cases} T_0 = \alpha_v \cap \overline{\sigma} \\ T_{n+1} = T_n \cup [\alpha_v \cap \sigma \cap Pre(T_n)] \end{cases}$$

Cette suite est convergente car l'opérateur $\cup$ est monotone et parce que l'ensemble des transitions du modèle est fini. Elle correspond classiquement au plus petit point fixe de la fonction:

$$G(T) = [\alpha_v \cap \overline{\sigma}] \cup [\alpha_v \cap Pre(T)]$$

soit encore, après simplification:

$$G(T) = \alpha_v \cap [\overline{\sigma} \cup Pre(T)]$$

En adoptant les notations du μ-calcul, l'ensemble des séquences d'exécution valides mais incorrectes est donc défini par:

$$\mu T.\ \alpha_v \cap [\overline{\sigma} \cup Pre(T)]$$

Par complémentation, on obtient alors l'ensemble σ_v des séquences d'exécution valides et correctes:

$$\sigma_v = \nu V.\ \overline{\alpha}_v \cup [\sigma \cap \widetilde{Pre}(V)]$$

On est maintenant en mesure d'exprimer la condition pour que le modèle vérifie les spécifications: il faut que toutes les transitions exécutables à partir de l'état initial q_{init} appartiennent à σ_v, soit encore que

$$q_{init} \in \widetilde{Src}(\sigma_v)$$

6.2.1 Interprétation graphique

Dans la section précédente, les séquences d'exécutions valides et correctes ont été caractérisées à partir des séquences valides mais incorrectes. Ces dernières correspondent à la figure 6.4. L'évaluation des spécifications revient à éliminer incrémentalement les transitions dont l'exécution peut mener à une transition valide mais incorrecte:

- En premier lieu, on élimine les transitions qui sont effectivement valides mais incorrectes.
- Ensuite, on élimine les transitions pouvant mener à une transition éliminée.

Lorsque toutes ces transitions ont été éliminées, on obtient l'ensemble des transitions constituant les séquences d'exécution valides et correctes du modèle par complémentation de l'ensemble des transitions éliminées. (voir figure 6.5).

Mais il est possible d'interpréter directement en termes de séquences d'exécution le point fixe défini par:

$$\sigma_v = \nu Y.\overline{\alpha}_v \cup [\sigma \cap \widetilde{Pre}(Y)]$$

Celui-ci correspond au calcul des termes d'une suite définie par:

$$\begin{cases} Y_0 &= \sigma \cup \overline{\alpha}_v \\ Y_{n+1} &= Y_n \cap [\overline{\alpha}_v \cup (\sigma \cap \widetilde{Pre}(Y_n))] \end{cases}$$

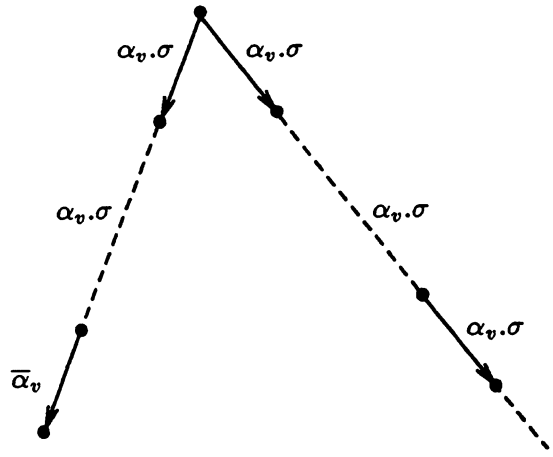


Figure 6.6: Séquences d'exécution dénotées par σ_v

Cette suite caractérise des ensembles de transitions. Pour $n = 0$, Y_0 est l'ensemble des transitions correctes ou non valides. Pour tout $n > 0$, Y_{n+1} est l'ensemble des transitions non valides ou des transitions conduisant *nécessairement* à une transition de Y_n. Par conséquent, les séquences d'exécution caractérisées par σ_v sont non seulement les séquences valides et correctes du modèle, mais aussi les séquences correctes et non valides de celui-ci (voir figure 6.6).

6.2.2 Autre caractérisation

La seconde approche pour évaluer les spécifications sur le modèle consiste à caractériser les séquences exécutables et valides du modèle, puis à vérifier que celles-ci sont correctes.

Pour tout ensemble d'états X, nous définissons l'opérateur $Acc_{\alpha_v}(X)$ calculant l'ensemble des transitions *valides* accessibles à partir de X:

$$Acc_{\alpha_v}(X) = \{(q,e) \in \alpha_v \ / \ q \in X\}$$

ou encore:

$$Acc_{\alpha_v}(X) = Acc(X) \cap \alpha_v$$

Pour tout ensemble de transitions T, nous définissons l'opérateur $Succ(T)$ calculant l'ensemble des états accessibles par exécution des transitions de T:

$$Succ(T) = \{q' \in Q \ / \ \exists (q,e) \in T, \ \vec{\delta}(q,e) = q'\}$$

Les séquences exécutables du modèle ont comme état de départ l'état q_{init}. Les transitions valides exécutables à partir de q_{init} forment l'ensemble $T_0 = Acc_{\alpha_v}(\{q_{init}\})$. L'exécution de ces transitions aboutit à un nouvel ensemble d'états accessibles, caractérisé par $Succ(T_0)$. Le calcul de l'ensemble T_1 des transitions valides et exécutables à partir de ces états peut alors être itéré. Finalement, l'ensemble des transitions valides exécutables à partir de q_{init} est la limite de la suite définie comme suit:

$$\begin{cases} T_0 &= Acc_{\alpha_v}(\{q_{init}\}) \\ T_{n+1} &= T_n \cup Acc_{\alpha_v}(Succ(T_n)) \end{cases}$$

Cette suite converge car l'opérateur $\cup$ est monotone et l'ensemble des transitions du modèle est fini. Sa limite s'exprime comme le plus petit point fixe de la fonction:

$$H(T) = Acc_{\alpha_v}(Succ(T))$$

soit, avec les notations du μ-calcul:

$$\sigma_v = \mu T.\ Acc_{\alpha_v}(\{q_{init}\}) \cup Acc_{\alpha_v}(Succ(T))$$

σ_v caractérise l'ensemble des séquences valides et exécutables du modèle. Vérifier les spécifications revient alors à contrôler que $\sigma_v \subset \sigma$.

6.3 Conclusions

On dispose maintenant d'un modèle d'exécution des programmes, et d'une caractérisation formelle de la sémantique des spécifications et des assertions sur ce modèle. L'étape suivante consiste à concevoir un outil réalisant de manière automatique la vérification formelle des programmes LUSTRE. Il s'agit donc dans un premier temps de définir une méthode d'évaluation des spécifications et des assertions sur le modèle. En outre, la méthode retenue doit être rigoureuse, de manière à pouvoir garantir avec une certitude *absolue* que les programmes sont corrects vis à vis de leurs spécifications. Le choix de la méthode doit être guidé par les deux principales particularités de la vérification en LUSTRE:

- Les comportements des programmes LUSTRE peuvent être représentés par des modèles qui sont des machines d'états finis.
- Les spécifications à vérifier sont des propriétés de sûreté.

Le premier point incite naturellement à employer des méthodes de vérification basées sur les modèles (en anglais *model-checking*), pour lesquelles les programmes LUSTRE semblent parfaitement adaptés. Intuitivement, ces méthodes consistent à *générer* un modèle représentant l'ensemble des comportements du système à vérifier, puis à *contrôler* sur ce modèle que le système est conforme à ses spécifications. Plusieurs méthodes ont été développées selon ce principe. Elles se distinguent les unes des autres par la façon de générer le modèle, et aussi par la manière d'effectuer sa vérification, l'objectif étant avant tout de réduire la taille du modèle généré, et d'abaisser le coût de la vérification. A cet égard, la restriction faite en LUSTRE à la vérification de propriétés de sûreté uniquement doit permettre de simplifier et d'optimiser les méthodes mises en œuvre.

L'autre aspect de la réalisation d'un outil de vérification formelle concerne les techniques utilisées pour la mise en œuvre les méthodes de vérification sur les modèles. En effet, le coût de ces dernières peut aussi être abaissé en utilisant des *techniques* efficaces de représentation et de manipulation des modèles. Cet aspect est fondamental car la taille de ces derniers est potentiellement exponentielle.

Dans les deux chapitres suivant, nous étudions d'abord le principe des deux principales méthodes de vérification basées sur les modèles, que nous avons appelées méthode "en avant" et méthode "en arrière". Pour chacune d'elle, nous discutons aussi de l'adéquation de deux techniques de représentation et de manipulation des modèles envisageables: la technique "énumérative" et la technique "symbolique". A l'issue de cette étude, les choix réalisés seront discutés et justifiés.

Chapitre 7

La méthode "en avant"

Une première approche de la vérification des programmes LUSTRE consiste en une application *directe* du principe de la simulation exhaustive. A partir du modèle d'exécution d'un programme de vérification, on peut vérifier les spécifications en contrôlant que la valeur de sa sortie reste invariante pour toute séquence exécutable et valide du modèle. Ces séquences d'exécution doivent donc être générées pour effectuer la vérification.

Dans la méthode en avant, la génération des séquences exécutables d'un modèle s'effectue selon le principe suivant:

- Au départ, on exécute toutes les transitions valides à partir de l'état initial. Chaque transition mène à un nouvel état, qui est un *successeur* de l'état initial, et qui est donc *accessible*.
- Ensuite, tout état accessible est visité une fois (et une seule). Les transitions valides dont il est la source sont alors exécutées, menant potentiellement à de nouveaux états accessibles.
- Finalement, lorsque tous les états accessibles ont été visités, toutes les séquences exécutables du modèle ont été simulées, ce qui achève la génération.

La figure 7.1 illustre sur un exemple les premières étapes de la simulation exhaustive d'un modèle, qui débutent la génération des séquences exécutables.

La vérification proprement dite consiste alors à contrôler que les spécifications gardent la même valeur sur toutes les séquences d'exécution générées. La méthode en avant se décompose donc en deux phases:

1. la génération des séquences exécutables du modèle.
2. l'évaluation des spécifications sur ces séquences.

Cette méthode de vérification est dite "en avant" car la génération des séquences d'exécution s'effectue en explorant les états accessibles du modèle à partir de son état initial, par exécution des transitions du modèle.

La méthode de vérification en avant a été largement étudiée dans le domaine des machines séquentielles, sans doute à cause de l'interprétation intuitive évidente sur laquelle elle est fondée. Nous étudions ici les différentes approches possibles pour la mise en œuvre de cette méthode.

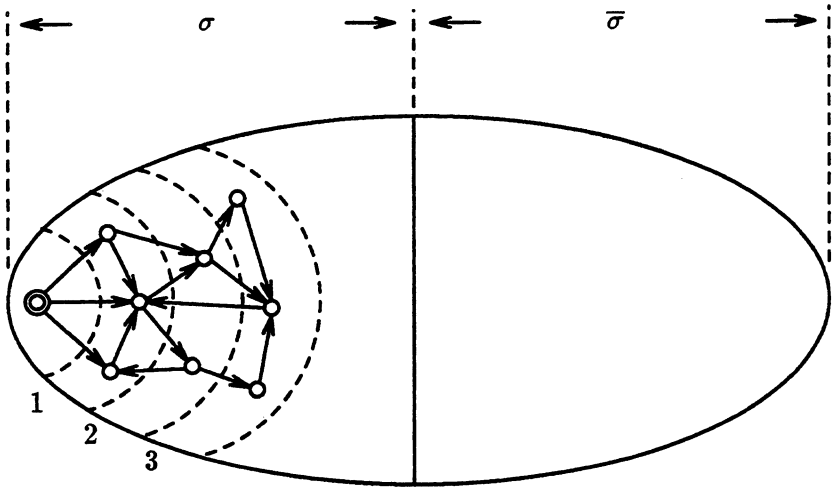


Figure 7.1: Simulation exhaustive du modèle

7.1 L'approche classique

L'approche classique de la méthode de vérification en avant consiste à traiter la génération des séquences d'exécution du modèle et la vérification de ces séquences de manière complètement indépendante. Dans un premier temps, une représentation explicite des séquences d'exécution est construite sous forme d'un système de transitions. Le modèle peut alors être représenté sous forme d'un ensemble de triplets $(q, e, q') \in Q \times E \times Q$ tels que la transition (q, e) mène à l'état q'. L'algorithme ci-dessous décrit de manière formelle la génération du modèle. Les notations d'ensembles adoptées ici sont les suivantes:

- T = Triplets du modèle
- V = états Visités
- A = états Accessibles mais non visités

De plus, nous utilisons les opérateurs *Exec* et *Post* définis dans la section 5.3. Nous rappelons que $Exec(X)$ est l'ensemble des transitions exécutables à partir des états de X, tandis que $Post(X)$ est l'ensemble des états accessibles à partir de X. L'algorithme de génération du modèle s'écrit alors:

debut
$T = \emptyset$
$V = \emptyset$
$A = \{q_{init}\}$
tantque $A \neq \emptyset$ **faire**
    debut
    $V = V \cup A$
    $T = T \cup \{(q, e, \vec{\delta}(q, e)) \in Q \times E \times Q \ / \ (q, e) \in Exec(A)\}$

$A = Post(A) \setminus V$
fin
fin

Dans le cas d'un programme de vérification, une fois que l'ensemble T représentant le modèle correspondant à ce programme est généré, la vérification des spécifications s'effectue très simplement. Il suffit de vérifier que toutes les transitions exécutables du modèle sont correctes. De manière formelle, cette vérification correspond à l'algorithme ci-dessous:

debut
pour *chaque* $(q, e, q') \in T$ **faire**
si $(q, e) \notin \sigma$
alors
retourner (Spécifications Non Satisfaites)
fin
retourner (Spécifications Satisfaites)
fin

Historiquement, l'approche classique a été la première mise en œuvre dans le cadre de la vérification sur les modèles. Elle a fait l'objet de nombreux travaux [QS82,CES86,GS90], qui ont permis d'en améliorer grandement les performances. Malheureusement, cette méthode se heurte à un problème intrinsèque: l'explosion du modèle. En pratique, ce dernier est souvent trop gros pour pouvoir être entièrement généré. Ce phénomène a deux explications. D'une part, le modèle généré est complet: toutes les informations relatives aux états et aux transitions y sont explicitement représentées, ce qui en augmente la taille. D'autre part, il n'est pas minimal, c'est-à-dire qu'il contient en général de nombreux états et transitions équivalents, donc redondants.

Afin de diminuer la taille de la représentation du modèle généré, des techniques de compactage des informations contenues dans celui-ci ont été développées, mais celles-ci restent limitées au niveau du gain qu'elles engendrent. On a aussi développé des méthodes de minimisation des modèles [Fer90], mais celles-ci ne sont exploitables que si le modèle a été complètement généré. Face à ces problèmes, une nouvelle approche de la vérification, dite "à la volée", a été développée. Celle-ci est décrite dans la section suivante.

7.2 L'approche "à la volée"

Dans le cadre de la vérification, l'alternative aux problèmes de taille du modèle rencontrés dans l'approche classique consiste à réduire au maximum le nombre d'informations nécessaires à la représentation du modèle et à la vérification des spécifications. Pour se faire, l'approche envisagée consiste à effectuer la génération et la vérification du modèle *simultanément*, d'où son nom de vérification "à la volée" [Hol87,JJ89,FM91]. Avec cette approche, il est possible de ne conserver du modèle généré que les informations indispensables à la vérification.

La vérification d'invariants sur un modèle s'adapte tout à fait à une approche à la volée. En effet, plusieurs remarques peuvent être faites à propos de l'approche classique vue précédemment:

- Les informations sur le modèle sont du type (*état de départ,entrée,état d'arrivée*), or pour la

vérification des spécifications, il est suffisant de connaître les couples (*état de départ,entrée*). Il est donc possible de réduire la quantité d'informations mémorisées.

- D'autre part, les informations sur les entrées ne servent qu'à évaluer les spécifications. Donc, si l'évaluation a lieu en même temps que la génération, ces informations n'ont pas besoin d'être mémorisées.

Finalement, les seules informations à mémoriser durant toute la vérification concernent les états accessibles du modèle. La vérification "à la volée" consiste alors à combiner la génération des états accessibles du modèle et l'évaluation des spécifications. L'algorithme ci-dessous décrit cette approche de manière formelle. Les notations d'ensembles sont les suivantes:

- V = états Visités
- A = états Accessibles mais non visités

debut
$V = \emptyset$
$A = \{q_{init}\}$
tantque $A \neq \emptyset$ **faire**
    debut
    $V = V \cup A$
    si $Exec(A) \not\subset \sigma$
    alors
        retourner (Spécifications Non Satisfaites)
    $A = Acc(A) \setminus V$
    fin
retourner (Spécifications Satisfaites)
fin

Concrètement, par rapport à la méthode de génération classique, les seules informations conservées pendant toute la durée de la vérification concernent les états accessibles du modèle, d'où une réduction évidente de la taille du modèle représenté.

L'approche "à la volée" enregistre de bien meilleures performances que la méthode classique pour la vérification. C'est donc logiquement qu'elle s'est imposée dans ce domaine. Cependant, il ne faut pas oublier que celle-ci ne modifie rien quant au modèle généré: celui-ci reste le même, seule change la manière dont il est représenté. C'est pourquoi, l'approche "à la volée" ne saurait fournir une solution aux problèmes d'explosion des modèles.

7.3 Techniques d'implémentation

Les approches de la vérification en avant que nous venons de décrire peuvent être implémentées selon différentes techniques. Les algorithmes de génération de modèles décrits précédemment sont définis en termes d'ensembles et d'opérations sur ces ensembles. Leur mise en œuvre nécessite par conséquent de définir des structures de données pour implémenter ces d'objets et leurs opérateurs associés. A ce niveau, deux grandes approches techniques se distinguent. Celles-ci sont évoquées ci-dessous.

7.3.1 Technique énumérative

La technique énumérative est basée sur une interprétation extrêmement simple de la notion d'ensemble, ces derniers étant considérés comme des unions explicites de singletons. L'intérêt de cette technique réside dans la facilité de définir une structure de données pour une telle représentation. Elle présente de plus l'avantage de permettre une implémentation simple et économique des opérateurs ensemblistes, sous la forme d'opérateurs sur des singletons. Ainsi, les opérateurs *Exec* et *Post* sont redéfinis sur les états du modèle de la manière suivante:

$$Exec : \begin{cases} Q & \longmapsto & \mathcal{P}(Q \times E) \\ q & \longrightarrow & Exec(q) = \{q\} \times E \end{cases}$$

$Exec(q)$ est l'ensemble des transitions exécutables à partir de q. De façon similaire, $Post(q)$ est l'ensemble des états successeurs de q:

$$Post : \begin{cases} Q & \longmapsto & \mathcal{P}(Q) \\ q & \longrightarrow & Post(q) = \{q' \in Q \ / \ \exists e \in E, \ \vec{\delta}(q,e) = q'\} \end{cases}$$

L'algorithme de vérification "à la volée" se réécrit alors de la manière suivante en technique énumérative:

- V = états Visités
- A = états Accessibles mais non visités

debut
$V = \emptyset$
$A = \{q_{init}\}$
tantque $A \neq \emptyset$ **faire**
  debut
  soit $q \in A$
  $V = V \cup \{q\}$
  $A = A \setminus \{q\}$
  si $Exec(q) \not\subset \sigma$
  alors
    retourner (Spécifications Non Satisfaites)
  $A = A \cup Post(q) \setminus V$
  fin
retourner (Spécifications Satisfaites)
fin

De même, l'algorithme formel décrivant l'approche classique peut être réécrit de la même manière en technique énumérative.

La technique énumérative permet donc une implémentation simple des différentes approches de la vérification en avant. Cependant, le principal problème qu'elle soulève reste lié à la manière dont les ensembles sont représentés: en effet, la taille de leur représentation est nécessairement proportionnelle à leur dimension. Par conséquent, toute explosion du modèle généré va directement se traduire par une explosion de sa représentation.

7.3.2 Technique symbolique

L'alternative aux problèmes d'explosion du modèle rencontrés en technique énumérative consiste à définir une représentation des ensembles dont la taille est indépendante de celle des ensembles eux-mêmes. En technique énumérative, le problème vient du fait que la représentation de tout ensemble consiste en une *énumération* explicite de ses éléments (d'où le nom donné à cette technique). L'idée consiste alors à rendre cette énumération *implicite*, de manière à représenter les ensembles en compréhension plutôt qu'en extension. Les techniques permettant ce type de représentation sont dites *symboliques*.

Une technique de représentation symbolique des fonctions booléennes a été récemment développée. Ils s'agit des BDDs, qui seront étudiés ultérieurement. Cette technique peut être adaptée à la représentation d'ensembles d'états et de transitions des modèles. Plusieurs implémentations de la méthode de vérification en avant ont été réalisées à partir de cette technique. Celle-ci présente en effet de nombreux avantages:

- Compacité de la représentation.
- Représentation d'ensembles quelconques.
- Implémentation efficace des opérateurs ensemblistes.

En utilisant cette représentation, l'implémentation des différentes approches de la méthode de vérification en avant est directe: elle revient à une traduction des algorithmes formels décrits précédemment.

7.4 La méthode en avant pour la vérification de Lustre

La vérification des programmes LUSTRE a déjà suscité de nombreux travaux. A l'origine, des tentatives [Rat88] ont été entreprises d'utiliser l'outil de validation de protocoles XESAR [RRSV87] pour vérifier les programmes LUSTRE. Cependant, de nombreuses incompatibilités se sont révélées entre le domaine d'utilisation de XESAR et les spécificités de LUSTRE:

- Traduction d'un programme synchrone dans le domaine asynchrone.
- Expression des spécifications.
- Prise en compte des assertions.

Ces problèmes ont mis en évidence la nécessité de concevoir des outils spécifiques à la vérification des programmes LUSTRE. A partir de ce constat, des travaux ont d'abord été menés autour de la spécification des programmes [Glo89]. A la suite de ceux-ci, des outils de vérification basés sur la méthode en avant ont été développés.

7.4.1 Prise en compte des assertions

Par rapport aux différentes approches décrites précédemment pour la vérification des modèles par la méthode en avant, la vérification des programmes LUSTRE doit en plus prendre en compte

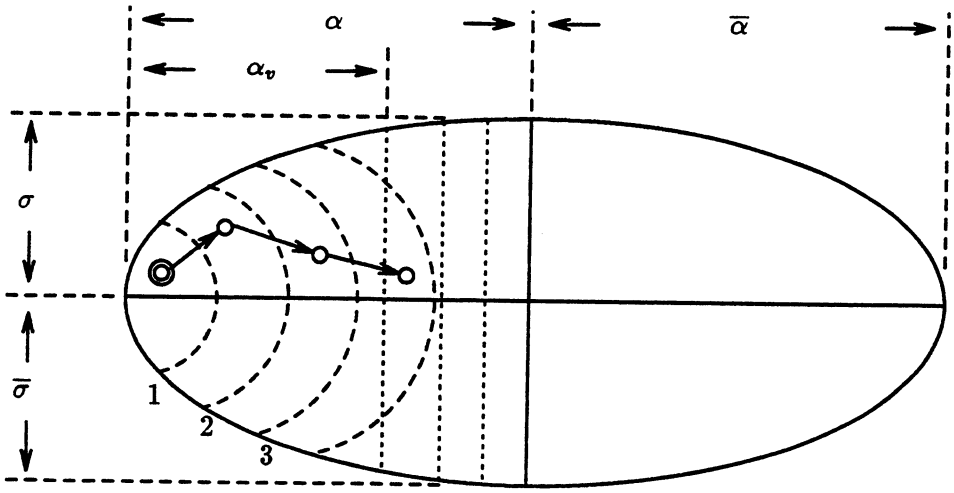


Figure 7.2: Exécution d'une transition correcte mais non valide

les assertions. Ces dernières définissent un ensemble α_v (caractérisé dans la section 6.1) de transitions *valides* du modèle, dont le calcul ne peut être effectué par la méthode en avant.

C'est pourquoi la sémantique des assertions ne peut être complètement prise en compte par les méthodes de vérification en avant. On se contente de considérer comme valides les transitions du modèle caractérisée par la fonction d'assertion α. Pratiquement, la prise en compte des assertions revient alors simplement à substituer l'opérateur *Exec* par l'opérateur $Exec_\alpha$, comme suit:

$$Exec_\alpha : \begin{cases} \mathcal{P}(Q) & \longmapsto & \mathcal{P}(Q \times E) \\ X & \longrightarrow & Exec_\alpha(X) = [X \times E] \cap \alpha \end{cases}$$

L'approximation faite sur les assertions est sans conséquence si celles-ci sont causales. Par contre, dans le cas inverse, on peut être confronté à deux types de problèmes au cours de la vérification en avant:

- on exécute une transition correcte mais non valide (Voir figure 7.2). Dans ce cas, la vérification peut se poursuivre normalement, en signalant toutefois que les assertions sont non causales.

- on exécute une transition incorrecte et non valide (Voir figure 7.3). On se retrouve dans un cas où on est obligé de conclure que le programme ne satisfait pas les spécifications, alors que c'est faux puisque la transition exécutée est en fait non valide.

Cependant, il faut remarquer que dans aucun cas on aboutit à la conclusion que les spécifications sont satisfaites alors qu'en fait elles ne le sont pas.

Les assertions non causales posent donc un problème impossible à résoudre par les méthodes de vérification en avant. Dans le cadre de l'approche classique, des systèmes de "backtracking" peuvent être développés afin d'éliminer du modèle les états dont on s'aperçoit qu'ils sont inaccessibles après les avoir explorés. Par contre, cela est impossible dans les méthodes de vérification

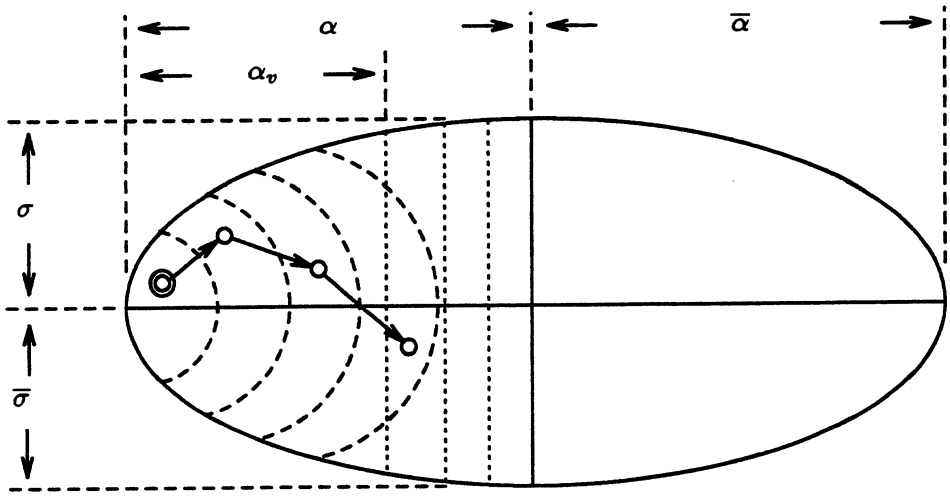


Figure 7.3: Exécution d'une transition incorrecte et non valide

à la volée, car pour diminuer la taille de la représentation du modèle, on ne garde pas trace des transitions entre les états explorés.

Malgré le problème des assertions non causales, des outils de vérification des programmes LUSTRE implémentant la méthode en avant ont été développés. Nous dressons ci-dessous un bref état de l'art des réalisations dans ce domaine.

7.4.2 Le compilateur Lustre

En LUSTRE la méthode de génération classique du modèle correspond exactement à la compilation des programmes sous forme d'automates, telle qu'elle est implémentée dans les deux versions du compilateur LUSTRE [Pla88,Ray91], où elle est utilisée pour la génération de code séquentiel. De fait, le premier outil à permettre la vérification formelle des programmes LUSTRE a été le compilateur du langage. Les programmes de vérification peuvent en effet être compilés sous forme d'automates. A partir de là, le code généré peut être contrôlé "à la main" afin de s'assurer que l'unique sortie du programme (qui correspond aux spécifications) reste toujours vraie. Toutefois, la méthode souffre de problèmes rédhibitoires:

- Elle n'est pas automatique!
- Les modèles sont générés selon l'approche classique: les problèmes d'explosion du modèle se retrouvent au niveau du compilateur.

7.4.3 Le vérificateur Lesar

La méthode de vérification énumérative à la volée a été implémentée dans le premier outil véritablement conçu pour la vérification des programmes LUSTRE: LESAR [Glo89]. L'utilisation de cet outil a permis une première approche de la vérification en LUSTRE. En particulier, il

a révélé les problèmes liés à la méthode en avant et à la technique énumérative: incapacité de traiter les assertions non causales et explosion du modèle.

Chapitre 8

La méthode "en arrière"

Le problème majeur des méthodes de vérification avant avant concerne la taille du modèle généré. En effet, ce dernier n'est en général pas minimal, et dans beaucoup de cas pratiques il atteint une taille explosive qui fait échouer sa génération. Or si le modèle ne peut pas être complètement généré, le problème de la validité des spécifications reste indécidable. La solution consiste alors à développer des méthodes de génération de modèles minimaux. Ces méthodes ne consistent pas à minimiser un modèle non minimal après qu'il ait été généré (puisque ce n'est pas toujours possible!), mais à en effectuer directement la génération. Une telle méthode a développée dans le groupe SPECTRE. Nous l'avons exploitée dans le cadre de la vérification des programmes LUSTRE. Dans un premier temps, nous décrivons le principe de cette méthode, puis nous étudions son application à la vérification.

8.1 Génération du modèle minimal

Nous décrivons de manière informelle le principe de la génération de modèle minimal appliquée aux programmes LUSTRE. Une description formelle et générale de la méthode peut être trouvée dans [BFH90a,BFH*92].

La méthode s'applique à la génération de machines d'états finis. Etant donné qu'il est possible d'associer à tout programme LUSTRE un modèle de ce type, la méthode s'applique parfaitement ici.

L'algorithme de génération d'un modèle minimal d'une machine d'états finis consiste à considérer *a priori* que tous les états de la machine sont équivalents, et à ne les distinguer qu'à partir du moment où il est prouvé que ce n'est pas le cas. Ce principe garantit la "minimalité" du modèle généré. On est alors amené à manipuler des classes d'états *équivalents*. Les critères d'équivalence entre états d'une classe sont ceux de la bisimulation [Fer90]. Les états d'une classe sont équivalents si et seulement si:

- ils sont identiques du point de vue de la valeur des sorties du modèle.
- ils sont identiques du point de vue de l'équivalence de leurs successeurs.

Une classe d'états est dite *accessible* si et seulement si elle contient l'état initial du modèle, ou s'il existe une transition d'une classe accessible vers cette classe. Une classe d'états est dite *stable* si

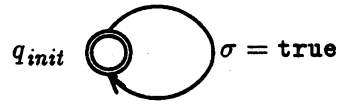


Figure 8.1: Le modèle minimal correspondant à un programme de vérification

et seulement si toutes ses sorties et tous ses successeurs ont été calculés.

L'algorithme de génération du modèle minimal fonctionne alors comme suit:

- Au départ, tous les états du modèle sont considérés comme équivalents. Ils forment donc une unique classe accessible mais non stable, puisque les sorties du modèle et les successeurs de la classe doivent être calculés.
- Ensuite, on calcule pour chaque classe accessible mais non stable les valeurs des sorties et les successeurs de cette classe. C'est à ce moment là que les états de la classe peuvent se révéler non équivalents. Dans ce cas, on effectue un "raffinage" de la classe, qui consiste à la partitionner de manière binaire. Lorsque ce raffinage est du à une différence sur la valeur d'une sortie, la classe est partagée en une sous-classe d'états dans lesquels la sortie prend la même valeur, et une autre sous-classe complémentaire de la première dans la classe d'origine. Il en est de même dans le cas où le raffinage résulte de la non-équivalence entre états successeurs. Dans ce cas, la classe est partagée en une sous-classe d'états dont les successeurs sont équivalents, et une autre sous-classe complémentaire de la première dans la classe d'origine. L'opération de raffinage crée à chaque fois de nouvelles classes qui deviendront éventuellement accessibles dans le futur.
- Finalement, lorsque toutes les classes accessibles deviennent stables, le modèle minimal est généré.

Toute l'originalité de l'algorithme est basée sur le fait que le modèle est généré en fonction de ses sorties. Le raffinage des classes dépend en priorité de celles-ci.

8.2 Application à la vérification de programmes

L'algorithme de génération de modèle minimal peut être appliqué au cas particulier d'un programme de vérification LUSTRE. Dans ce cas, la seule sortie du programme est la spécification à vérifier. Si celle-ci est effectivement un invariant du programme, alors toutes les classes accessibles du modèle sont équivalentes du point de vue de la valeur de la sortie. Par conséquent, aucun raffinage ne s'effectuera par rapport à celle-ci. Du coup, le modèle minimal associé à un tel programme est constitué d'un état émettant toujours la même valeur en sortie et dont l'unique successeur est lui-même (voir figure 8.1). On peut alors mesurer l'intérêt de cette méthode où le modèle généré ne peut pas être plus petit, alors que la méthode en avant génère des modèles constitués de dizaines d'états équivalents.

Comment alors interpréter les calculs effectués par l'algorithme de génération de modèle minimal dans ce cas particulier? Nous allons donner ici une interprétation intuitive du fonction-

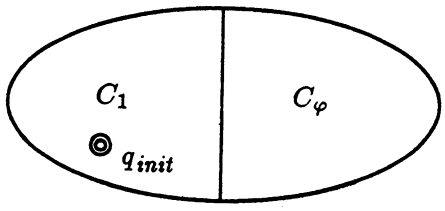


Figure 8.2: Partition initiale du modèle d'un programme de vérification

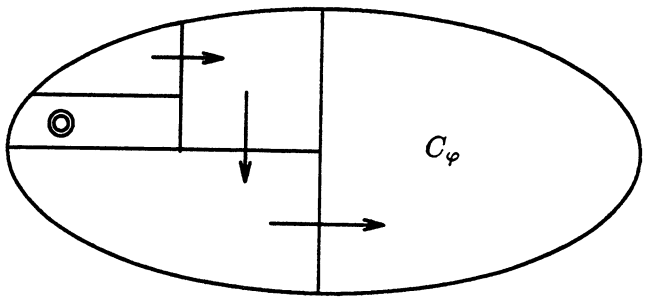


Figure 8.3: Génération du modèle minimal d'un programme de vérification

nement de l'algorithme. Initialement, la partition des états du modèle est constituée d'une classe C_1 où la valeur de la sortie du programme est *vraie*, et d'une classe C_φ où la sortie prend tout autre valeur. Cette partition correspond à la figure 8.2. Si q_{init} n'appartient pas à C_1, alors la spécification n'est pas vérifiée. Dans le cas contraire, il faut rendre C_1 stable en calculant ses successeurs. Cela peut conduire à un raffinage de C_1 en deux sous-classes d'états dont les successeurs ne sont pas équivalents. L'état initial est nécessairement contenu dans l'une de ces deux nouvelles classes, appelée C_2. Or si C_2 a des successeurs dans C_φ, il existe alors une séquence d'exécution partant de l'état initial et où les spécifications ne sont pas vérifiées. Donc, pour que les spécifications soient un invariant du programme, les successeurs de C_2 doivent nécessairement appartenir à C_2.

Le calcul des successeurs va être itéré sur C_2, et ainsi de suite jusqu'à ce que la seule classe accessible (celle qui contient l'état initial) devienne stable, c'est-à-dire que les états qu'elle contient soient équivalents du point de vue de leurs successeurs. L'algorithme consiste donc à éliminer progressivement tout état *prédécesseur* d'un état ne vérifiant pas les spécifications (voir figure 8.3). Si q_{init} fait partie de ces états, les spécifications ne sont pas vérifiées.

On s'aperçoit donc que l'application de l'algorithme de génération de modèle minimal à un programme de vérification revient au calcul des états à partir desquels il est possible d'atteindre un état ne vérifiant pas les spécifications, qui forment les *prédécesseurs* des états de la classe C_φ. C'est pourquoi cette méthode est appelée "en arrière", par opposition à la méthode en avant, où les calculs sont basés sur la recherche d'états *successeurs*.

Finalement, la méthode de vérification en arrière peut être décomposée en deux phases:

1. Génération: On élimine tout d'abord les états du modèle qui ne vérifient pas les spécifications. Puis on élimine au fur et à mesure les états prédécesseurs de tout état éliminé.

2. Vérification: On contrôle que l'état initial du modèle n'a pas été éliminé.

8.3 Techniques d'implémentation

L'algorithme de génération de modèle minimal manipule des classes d'états équivalents et nécessite la représentation et la mise en œuvre d'opérateurs sur ces ensembles. Une implémentation basée sur une approche énumérative - où les seuls ensembles manipulés sont des singletons - n'est donc pas adaptée à cette méthode. Seule les techniques symboliques peuvent être appliquées ici.

L'algorithme de génération de modèle minimal a déjà été implémenté dans la dernière version du compilateur LUSTRE [Ray91], où il sert pour la génération de code sous forme d'automates.

8.4 Choix effectués

L'étude des méthodes de vérification qui vient d'être faite doit maintenant permettre de déterminer celle qui est la mieux adaptée à la vérification des programmes LUSTRE. La méthode en avant a déjà été intensivement étudiée dans le domaine général de la vérification de machines séquentielles, mais aussi dans le cadre de la vérification des programmes LUSTRE. L'état de l'art du développement d'outils de vérification basés sur cette méthode est le suivant:

- Approche classique: elle correspond au compilateur LUSTRE V2.
- Approche "à la volée" par la technique énumérative: elle est implémentée par LESAR, qui est un outil de vérification de programmes LUSTRE.
- Approche "à la volée" par la technique symbolique: elle n'est pas implémentée pour la vérification en LUSTRE, mais elle a déjà fait l'objet de nombreux travaux similaires [SB90, Mad90,Cou91]. Cette approche est désormais bien maitrisée et plusieurs outils efficaces ont été implémentés.

Cependant, on peut formuler trois critiques à l'encontre de la méthode en avant:

- Elle ne permet pas de générer un modèle minimal, d'où les problèmes d'explosion du modèle auxquels elle est confrontée.
- La génération du modèle est uniquement basée sur le critère d'accessibilité des états. Elle s'effectue donc indépendamment des spécifications à vérifier, ce qui est étonnant pour une méthode destinée à être implémentée au sein d'un outil de vérification.
- Elle ne permet pas de prendre en compte la sémantique des assertions LUSTRE.

En ce qui concerne la méthode en arrière, ses principales caractéristiques sont les suivantes:

- Elle est basée sur la génération de modèle minimal, ce qui minimise complètement les risques d'explosion des modèles rencontrés dans la méthode en avant.
- La génération implicite du modèle est guidée par les sorties de celui-ci. Dans le cas d'un programme de vérification, l'unique sortie est équivalente aux spécifications, donc le modèle généré dépend des spécifications à vérifier. Par conséquent, cette méthode privilégie l'évaluation des spécifications, ce qui semble naturel dans le cadre de la vérification.
- Elle permet de prendre en compte la sémantique des assertions sur les modèles des programmes LUSTRE.

Il aparait enfin que la méthode en arrière n'a jamais été implémentée dans le cadre de la vérification. Cela est certainement du au fait qu'elle nécessite l'emploi de techniques symboliques pour son implémentation. La popularité de ces dernières étant assez récente, les recherches sur la mise en œuvre de la méthode en arrière n'ont encore jamais été entreprises. Compte tenu de ce constat, et des avantages présumés de cette méthode, nous avons naturellement choisi de l'implémenter. La partie suivante décrit comment cette implémentation a été réalisée. Mais auparavant, le chapitre suivant décrit le principe d'une méthode de diagnostic complémentaire de la méthode de vérification en arrière, et qui en est indissociable.

Chapitre 9

Diagnostic

La vérification d'un programme consiste à déterminer s'il se comporte conformément à ses spécifications. Nous avons défini des méthodes formelles permettant de contrôler et de garantir cette conformité au niveau des programmes LUSTRE. Cependant, le résultat de toute vérification est binaire: si le programme est conforme à ses spécifications, son comportement est garanti correct par rapport à celles-ci, et il n'y a plus rien à faire. Dans le cas contraire, le résultat de la vérification démontre simplement *l'existence* d'erreurs dans le programme, provoquant sa non conformité vis à vis de ses spécifications. De telles erreurs doivent évidemment être localisées et corrigées par l'utilisateur.

A ce stade, on peut dresser un parallèle entre la mise au point classique des programmes et leur mise au point dans le cadre de la vérification formelle. L'approche classique consiste à alterner l'exécution des programmes dans un environnement de simulation avec une phase de correction à l'aide d'outils de débogage. De façon similaire, la vérification formelle réalise la simulation exhaustive des programmes. En cas d'échec de celle-ci, un outil d'assistance à la correction d'erreurs s'avère tout aussi indispensable qu'un outil de débogage dans l'approche classique. En effet, les programmes vérifiés sont souvent complexes, et les erreurs détectées ont trois origines possibles: elles peuvent non seulement provenir du programme, mais aussi des spécifications ou des assertions. A ce stade, seul l'opérateur humain est capable de déterminer leur origine précise. Toutefois, celui-ci pourrait être assisté dans sa tâche par un outil de *diagnostic* permettant de mettre en évidence les raisons pour lesquelles les programmes ne satisfont pas leurs spécifications.

Nous abordons donc ici la définition d'une méthode de diagnostic associée à la vérification des programmes LUSTRE par la méthode en arrière, et son implémentation sous forme d'un outil de diagnostic.

9.1 Principe du diagnostic en Lustre

Au sens premier du terme, le diagnostic élaboré à partir d'un programme ne vérifiant pas ses spécifications consisterait à indiquer précisément les erreurs à l'origine de ce résultat. Toutefois, il est clair que l'abstraction nécessaire à la localisation des erreurs ne peut être obtenu que par un opérateur humain. L'objectif visé dans le cadre de la définition d'un diagnostic associé à la vérification formelle est par conséquent bien plus modeste.

En pratique, la vérification est effectuée sur un modèle du programme, en contrôlant que toutes les séquences exécutables du modèle sont correctes. S'il existe des séquences exécutables mais incorrectes, celles-ci correspondent alors à des comportements indésirables du programme. Le diagnostic consiste dans ce cas à construire puis à traduire ces séquences dans le langage *source* du programme, afin que l'utilisateur puisse les interpréter comme des comportements incorrects du programme.

Sur le modèle d'un programme LUSTRE, une séquence d'exécution est constituée d'un certain nombre de transitions (*état,entrée*) encodées sur un ensemble de variables d'état et de variables d'entrée. Celles-ci correspondent à des expressions dans le programme source. La traduction d'une séquence d'exécution en LUSTRE consiste alors à donner la valeur de ces expressions pour chacune des transitions qui la constituent, autrement dit à fournir une *trace* d'exécution du programme.

Le diagnostic en LUSTRE nécessite donc deux opérations: il faut tout d'abord élaborer les séquences exécutables mais incorrectes du modèle, puis les traduire sous forme de traces d'exécution incorrectes du programme.

9.2 Elaboration des séquences diagnostiques

De la manière dont il vient d'être défini, le diagnostic est un processus complémentaire de la vérification. En effet, la vérification d'un modèle consiste à contrôler que toutes les séquences exécutables de celui-ci sont correctes. Elle échoue donc dés qu'une séquence exécutable mais incorrecte est détectée, auquel cas celle-ci peut être exhibée en guise de diagnostic. C'est pourquoi au niveau de son implémentation, le diagnostic peut couplé avec le processus de vérification, en apportant quelques modifications à ce dernier. La méthode de diagnostic est donc dépendante de la méthode de vérification, ce qui implique qu'à chaque méthode de vérification correspond une méthode de diagnostic qui lui est propre. Nous décrivons ici le diagnostic élaboré pour la vérification en arrière.

La méthode de vérification en arrière consiste à générer l'ensemble des séquences d'exécution correctes d'un modèle, qui sont caractérisées (voir section 12.4) par le point fixe suivant:

$$\sigma_v = \nu T.\ \overline{\alpha_v} \cup [\sigma \cap \widetilde{Pre}(T)]$$

Or, celui-ci est le complémentaire du point fixe définissant l'ensemble des séquences d'exécution incorrectes. En pratique, σ_v est déterminé itérativement par calcul des termes de la suite $(T_n)_{y \in N}$ (voir section 12.4):

$$\begin{cases} T_0 &= \overline{\alpha}_v \cup \sigma \\ T_{n+1} &= T_n \cap [\overline{\alpha}_v \cup \sigma \cap \widetilde{Pre}(T_n)] \end{cases}$$

Dans l'algorithme de vérification, la condition d'arrêt au calcul des termes de cette suite correspond soit à sa convergence, qui indique que les spécifications sont vérifiées par le modèle, soit au fait que l'état initial du modèle est incorrect. Dans ce dernier cas, il existe au moins une séquence exécutable du modèle qui est incorrecte. Mais en général, il y en a *plusieurs*. On peut alors envisager différentes sortes de diagnostic, en fonction du nombre et du genre des séquences incorrectes exhibées. Toutefois, notre objectif ici se limitera à montrer comment il est possible

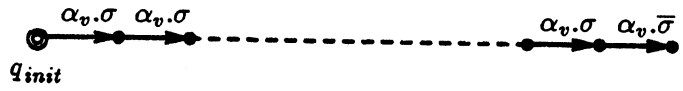


Figure 9.1: Séquence exécutable et valide mais incorrecte

d'en exhiber une. D'autre part, le choix d'une séquence incorrecte peut être fonction de différents critères que nous n'étudierons pas dans ce qui suit.

Le principe du diagnostic décrit ci-dessous est de construire *incrémentalement* une séquence exécutable mais incorrecte *de longueur minimale.* Une telle séquence est du type de la figure 9.1. Sa construction consiste, en partant de l'état initial, à calculer successivement les transitions qui la composent. Or, celles-ci font toutes partie des transitions éliminées pendant la phase de vérification, étant donné qu'elles peuvent appartenir à une séquence valide mais incorrecte du modèle (voir section 6.2.1). Par conséquent, elles sont caractérisées par les termes de la suite $(Z_n)_{n\in N}$, définie par:

$$\forall\, n \in N, Z_n = \overline{T}_n$$

Cette suite définit les ensembles de transitions successivement éliminées au cours de la vérification, à partir desquels une séquence diagnostique va être générée (voir figure 9.2). Par conséquent, il suffit de mémoriser les termes T_n calculés pendant la vérification pour pouvoir fournir un diagnostic en cas d'échec.

Le diagnostic consiste, à partir de l'état initial, en une itération au cours de laquelle à chaque étape, une transition (q_i, e_i) est *choisie* dans l'ensemble Z_{k-i}, selon les relations suivantes:

$$\begin{cases} q_0 = q_{init} \\ \forall\, i < k, q_{i+1} = \vec{\delta}(q_i, e_i) \end{cases}$$

On obtient ainsi une séquence exécutable mais incorrecte. L'algorithme informel lié au diagnostic associée à la méthode de vérification en arrière est le suivant:

debut
$q = q_{init}$
pour *i allant de k à* 0 **faire**
    debut
    choisir e telle que $(q, e) \in Z_i$
    afficher (q, e)
    $q = \vec{\delta}(q, e)$
    fin
fin

Dans cet algorithme, la séquence générée est construite heuristiquement: elle dépend du choix de l'entrée e réalisé à chaque cycle, qui détermine la transition exécutée. Selon les critères utilisés pour effectuer ce choix, il est possible d'exhiber diverses séquences diagnostiques.

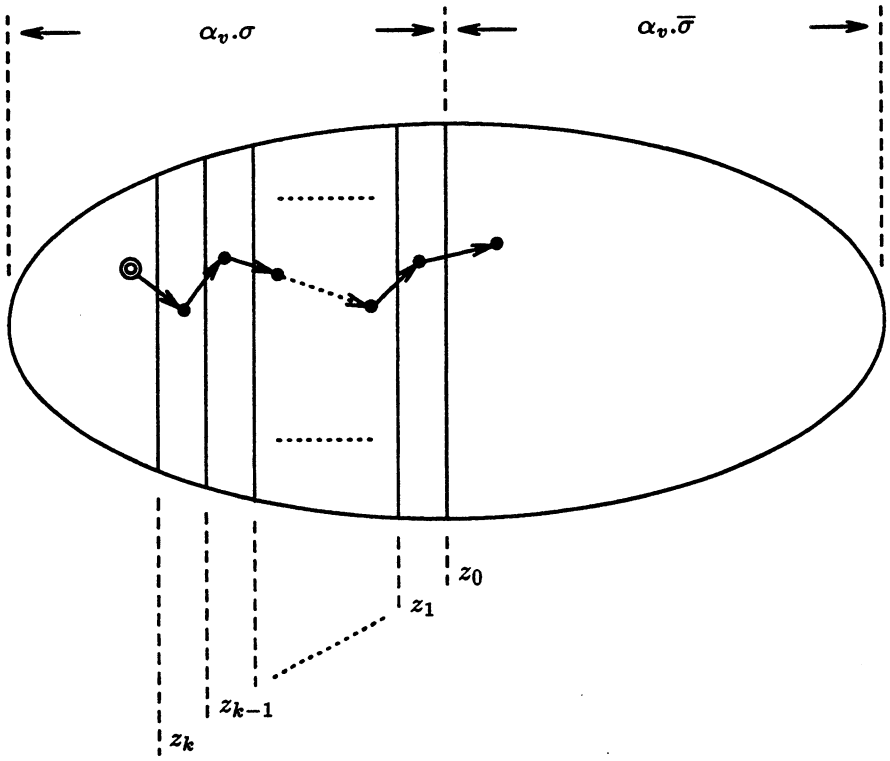


Figure 9.2: Construction d'une séquence diagnostique

9.2.1 Critique du diagnostic présenté

Le diagnostic associé à la méthode en arrière a la propriété d'exhiber une séquence d'exécution incorrecte de longueur *minimale*, ce qui présente un avantage pour l'utilisateur, dont le travail d'interprétation du programme est réduit au minimum.

Cependant, le diagnostic généré actuellement doit faire face à deux problèmes. Tout d'abord au niveau méthodologique, le langage source dans lequel il est exprimé n'est pas LUSTRE mais le format intermédiaire EC. En effet, le modèle sur lequel s'effectue la vérification des programmes LUSTRE est déterminé à partir d'une traduction de ceux-ci au format EC. Il est donc impossible d'élaborer un diagnostic directement dans le langage source.

D'autre part, au niveau de son implémentation, le diagnostic présente un surcoût pour la méthode en arrière, car celui-ci nécessite la mémorisation de chaque terme de la suite d'ensembles calculés pendant la phase de vérification. Dans certains cas, ce surcoût peut être un obstacle au bon déroulement de la vérification.

9.2.2 Ebauche d'un véritable outil de diagnostic

Il est possible d'envisager un principe de diagnostic beaucoup plus général que celui décrit précédemment. Ainsi, le diagnostic présenté consiste à générer selon un choix heuristique fixé *une seule* séquence d'exécution ne vérifiant pas les spécifications. L'utilisateur dispose donc de

la description d'un unique comportement incorrect du programme. Pourtant, il existe en général *plusieurs* comportements de ce genre, dont certains sont plus explicites ou plus compréhensibles que d'autres. La mise en évidence de plusieurs comportements incorrects permet alors une localisation plus aisée des sources d'erreur dans les programmes. Il serait donc intéressant de concevoir un outil de diagnostic interactif, offrant à l'utilisateur la possibilité d'obtenir plusieurs séquences d'exécution incorrectes générées automatiquement par l'outil, ou d'être assisté par ce dernier dans la construction "manuelle" de ces séquences. Un tel outil couplé à un outil de vérification en arrière bénéficierait en outre de toute la puissance du calcul symbolique pour implémenter les fonctionnalités décrites ci-dessus.

Partie III

Implémentation

Chapitre 10

Obtention du modèle d'exécution

En pratique, l'obtention du modèle d'exécution d'un programme LUSTRE requiert plusieurs phases de traitement, qui sont effectuées par le compilateur du langage. La compilation des programmes LUSTRE est divisée de manière tout à fait classique en une phase de vérification statique suivie d'une phase de génération de code. Certaines vérifications sont standard: l'analyse syntaxique des programmes et la vérification de types en font partie. D'autres sont plus spécifiquement liées à l'aspect "flot de données" du langage (inter-blocages, horloges). A l'issue de ces vérifications, le programme correct est traduit dans un format intermédiaire appelé EC (de l'anglais *Expanded Code*). Deux outils permettent la vérification statique des programmes LUSTRE et leur réécriture au format EC:

- L'un est la partie "haute" du compilateur LUSTRE-V2 [Pla88].
- L'autre, récemment développé par F. Rocheteau dans le cadre du projet POLLUX [Roc92], est un outil indépendant dédié à cette seule tâche.

L'obtention du modèle d'exécution associé à un programme s'effectue alors à partir du programme réécrit au format EC. En pratique, cette tâche est réalisée par la partie "haute" du générateur de code développé par P. Raymond [Ray91]. En effet, à partir du modèle d'un programme il est possible de générer du code exécutable synthétisant un automate de contrôle. Nous avons donc réutilisé le module de définition de modèle développé dans cet outil, dont le fonctionnement est basé sur une phase de "normalisation" du programme et une phase "d'identification" du contrôle. Ces phases sont brièvement décrites ci-dessous.

10.1 Normalisation du programme

La première phase menant à l'obtention d'un modèle consiste en une transformation syntaxique du programme, dont le but essentiel est de le réécrire à l'aide d'un nombre limité de constructions syntaxiques classiques et d'opérateurs simples. A l'issue de la normalisation, on obtient une représentation interne des programmes. Classiquement, celle-ci est un arbre abstrait [ASU86]. En LUSTRE, ce concept peut être poussé plus loin, compte tenu du *principe de substitution* du langage: toute équation du type `x = e` autorise la substitution de toute référence

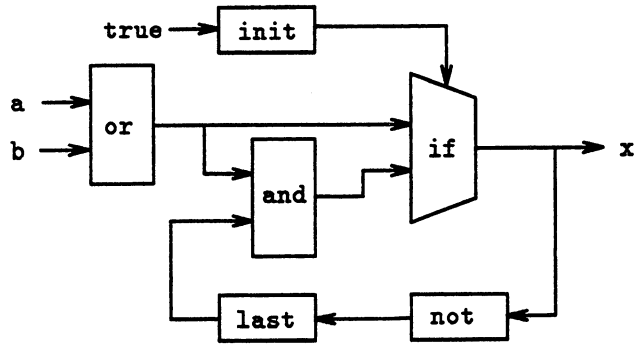


Figure 10.1: Un réseau d'opérateurs normalisé

à l'identificateur `x` par l'expression `e` dans le programme, et réciproquement. Après normalisation, les programmes peuvent alors être représentés de manière interne sous forme de *réseaux d'opérateurs*, éventuellement cycliques. La figure 10.1 donne un exemple de réseau d'opérateurs normalisé correspondant à l'équation

```
x = (a or b) and (true -> not pre x)
```

Dans ce réseau, `init` et `last` sont des opérateurs issus de la normalisation des programmes:

- `init` sert à caractériser l'état initial du programme. Il implémente l'opérateur `->`.
- `last` sert dans l'exemple traité à implémenter l'opérateur `pre`.

10.2 Identification du contrôle

Une fois le programme normalisé et représenté par un réseau d'opérateurs, un modèle d'exécution est obtenu par analyse du réseau. L'abstraction booléenne sur ce modèle consiste alors à ne prendre en considération que les expressions booléennes du réseau, dont font partie toutes les expressions synthétisant le *contrôle* du programme. A partir de ces expressions, les *éléments de contrôle* du programme sont identifiés. Ce sont:

- **Mémoires booléennes:** Elles correspondent à toutes les expressions du type `init(`*exp*`)` et `last(`*exp*`)`, où *exp* est une expression booléenne. Ces mémoires booléennes sont appelées *variables d'état.* Dans le réseau de la figure 10.1, `init(true)` et `last(x)` sont des variables d'état.
- **Expressions non évaluables:** Ce sont toutes les expressions booléennes dont l'évaluation ne peut être que dynamique. Il s'agit des entrées booléennes et des comparaisons entre expressions non booléennes (=, <>, <, >, etc...).

En pratique, l'identification des éléments de contrôle est effectuée par un parcours du réseau d'opérateurs représentant le programme normalisé. A partir de là, on peut déduire directement un modèle d'exécution du programme.

10.3 Encodage du modèle

La mémoire booléenne du programme correspond aux états du modèle. Chaque valeur possible de son contenu coïncide avec un état. Tout état peut alors être symbolisé par le vecteur de valeurs des variables d'état, de sorte que si n variables d'état ont été identifiées dans le programme, celles-ci définissent un ensemble Q constitué des 2^n états possibles du modèle. Parmi ces états, il existe un état initial q_{init}, qui est unique car les programmes LUSTRE sont déterministes.

Les expressions booléennes non évaluables du programme correspondent aux *variables d'entrée* du modèle. Une entrée du modèle peut alors être symbolisée par le vecteur de valeurs des variables d'entrée, de sorte que si m variables d'entrée ont été identifiées dans le programme, celles-ci définissent un ensemble E constitué des 2^m entrées possibles du modèle.

A partir des variables d'état et des variables d'entrée, toute expression booléenne du programme est une *fonction booléenne* de ces variables. Ainsi, on obtient la représentation des trois fonctions du modèle:

- La fonction d'assertion α, qui est une fonction booléenne sur $Q \times E$.
- La fonction de sortie σ, qui est une fonction booléenne sur $Q \times E$ dans le cas d'un programme de vérification.
- La fonction de transition $\vec{\delta}$, qui est une fonction booléenne *vectorielle* de $Q \times E$ dans Q. A chaque variable d'état du modèle est associée une fonction de transition *partielle*, qui est une fonction booléenne sur $Q \times E$. Cette fonction correspond à l'expression booléenne se trouvant en paramètre des opérateurs `init` et `last`. A chaque instant, un état et une entrée du modèle étant donnés, l'évaluation de la fonction de transition partielle d'une variable d'état donne la valeur que celle-ci prendra à l'instant suivant. Par conséquent, l'évaluation des fonctions de transition partielles donne l'état successeur de l'état courant pour une entrée donnée. La fonction de transition du modèle correspond alors au vecteur des fonctions de transition partielles.

Finalement, le modèle obtenu est donc une machine d'états finis (automate de Mealy), encodée sur des domaines booléens. Pour un programme de vérification, le modèle $\mathcal{M} = (Q, E, S, q_{init}, \alpha, \sigma, \vec{\delta})$ est défini comme suit:

- $Q = \{0,1\}^n$ est l'ensemble des états (vecteurs booléens de dimension n)
- $q_{init} \in Q$ est l'état initial
- $E = \{0,1\}^m$ est l'ensemble des entrées (vecteurs booléens de dimension m)
- $\alpha : Q \times E \mapsto \{0,1\}$ est la fonction d'assertion
- $\sigma : Q \times E \mapsto \{0,1\}$ est la fonction de sortie
- $\vec{\delta} : Q \times E \mapsto Q$ est la fonction de transition

Par rapport à une machine classique, le modèle défini ici se distingue par la fonction d'assertion qui entre dans sa définition.

Chapitre 11

Représentation symbolique du modèle d'exécution

11.1 Représentation des fonctions booléennes

Les choix effectués pour implémenter la vérification formelle des programmes LUSTRE nécessitent l'emploi d'une technique efficace pour représenter et manipuler symboliquement les fonctions booléennes. Plus précisément, celle-ci doit permettre une représentation compacte des fonctions et une mise en œuvre économique des opérateurs de décision et de calcul booléen impliqués dans les algorithmes de vérification.

Le problème de la représentation des fonctions booléennes est intrinsèquement difficile, puisqu'il est exponentiel par rapport au nombre de variables qui déterminent leur domaine de définition. Néanmoins, la nécessité de disposer d'une telle forme de représentation a depuis longtemps suscité d'importants efforts de recherche, donnant naissance à diverses propositions [Cou91,Sto36]. Mais curieusement, ce n'est que récemment qu'une nouvelle forme de représentation est apparue comme une solution réellement innovatrice et intéressante dans ce domaine: il s'agit des **B**inary **D**ecision **D**iagrams, ou **BDD**s. La popularité des BDDs s'est fortement accrue depuis qu'ils ont acquis une très solide réputation en tant que technique de représentation et de manipulation des fonctions booléennes, de par leurs performances et leur souplesse d'utilisation. C'est la raison pour laquelle ils sont maintenant largement utilisés dans de nombreux domaines, tel que la synthèse et la preuve de circuits [CB90,BL90,HS91], la vérification de protocoles [KLM91,EE91] ou la compilation de programmes [Ray91,HRR91]. En particulier, on les retrouve dans l'implémentation de la plupart des outils de vérification symbolique développés récemment. Constatant les qualités qui leurs sont unanimement reconnues, nous avons adopté les BDDs pour la représentation des fonctions booléennes dans notre implémentation de la vérification des programmes LUSTRE. C'est pourquoi nous rappellons ici les principes de base de cette représentation.

11.1.1 Expansion de Shannon

Les BDDs résultent du *théorème d'expansion de Shannon* [Sha38] des fonctions booléennes. Etant donné un ensemble de variables $X = \{x_1, \ldots, x_n\}$ où $n \in N^*$ est quelconque, et une fonction

booléenne $f(x_1, \ldots, x_n)$ de $\{0,1\}^n$ dans $\{0,1\}$, l'expansion de Shannon de f par rapport à la variable x_i $(1 \leq i \leq n)$ est le couple de fonctions $(f_{\bar{x}_i}, f_{x_i})$ défini par:

$$\begin{aligned} f_{\bar{x}_i} &= f(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n) \\ f_{x_i} &= f(x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_n) \end{aligned}$$

La terminologie associée à cette expansion est la suivante:

- x_i est la *variable d'expansion.*
- $f_{\bar{x}_i}$ est le *cofacteur* de f par rapport à $\bar{x}_i$.
- f_{x_i} est le *cofacteur* de f par rapport à x_i.

Les cofacteurs $f_{\bar{x}_i}$ et f_{x_i} ne dépendant plus de la variable x_i, ce sont donc des fonctions de $\{0,1\}^{n-1}$ dans $\{0,1\}$.

De façon formelle, nous définissons l'expansion de Shannon des fonctions booléennes de $\{0,1\}^n$ dans $\{0,1\}$ par rapport à une variable x comme étant une fonction $\Lambda_{n,x}$, définie comme suit:

$$\Lambda_{n,x} : \begin{cases} (\{0,1\}^n \rightarrow \{0,1\}) & \longmapsto & (\{0,1\}^{n-1} \rightarrow \{0,1\})^2 \\ f & \longrightarrow & (f_0, f_1) \ / \ f = \neg x \wedge f_0 \vee x \wedge f_1 \end{cases}$$

Le théorème de Shannon ci-dessous assure l'unicité du couple (f_0, f_1):

Théorème 1 *Etant donnée une fonction booléenne $f(x_1, \ldots, x_n)$ de $\{0,1\}^n$ dans $\{0,1\}$ et la variable x_i $(1 \leq i \leq n)$, il existe un couple unique de fonctions booléennes (f_0, f_1) de $\{0,1\}^{n-1}$ dans $\{0,1\}$ tel que:*

$$f = \neg x_i \wedge f_0 \vee x_i \wedge f_1$$

On déduit de ce résultat que f_0 et f_1 sont nécessairement les cofacteurs de f par rapport à $\bar{x}_i$ et par rapport à x_i, respectivement.

Finalement, l'expansion de Shannon de toute fonction booléenne f définie sur $\{x_1, \ldots, x_n\}$ par rapport à n'importe quelle variable x_i $(1 \leq i \leq n)$ correspond de façon unique à la réécriture de f sous la forme:

$$f = \neg x_i \wedge f_{\bar{x}_i} \vee x_i \wedge f_{x_i}$$

L'unicité de l'expansion de Shannon est une propriété fondamentale car elle caractérise de manière unique toute fonction booléenne. Ainsi, si deux fonctions f et g de $\{0,1\}^n$ dans $\{0,1\}$ admettent des expansions de Shannon identiques par rapport à une de leurs variables x_i $(1 \leq i \leq n)$, alors f est égale à g. Les fonctions $\Lambda_{n,x}$ sont donc des bijections de $(\{0,1\}^n \rightarrow \{0,1\})$ dans $(\{0,1\}^{n-1} \rightarrow \{0,1\})^2$.

11.1.2 Expansion complète

D'après le théorème de Shannon, toute fonction booléenne f de $\{0,1\}^n$ dans $\{0,1\}$ peut être indifféremment expansée selon chacune des n variables $x_1,\ldots,x_n$ de son domaine de définition. Si f est expansée par rapport à x_i, les cofacteurs $f_{\bar{x}_i}$ et f_{x_i} peuvent à leur tour être expansés par rapport à l'une des $n-1$ variables de leur domaine de définition $\{x_1,\ldots,x_n\}\setminus\{x_i\}$, et ainsi de suite jusqu'à ce que le domaine de définition des fonctions à expanser soit vide. En fin de compte, on obtient une expansion "complète" de f. Il existe *plusieurs* expansions de ce type, dont le nombre est donné par le théorème suivant:

Théorème 2 *Le nombre d'expansions de Shannon complètes d'une fonction booléenne f de $\{0,1\}^n$ dans $\{0,1\}$ est*

$$S_n = \Pi_{i=0}^{n-1}(n-i)^{2i}$$

Preuve: Par récurrence sur n.
Au rang $n=0$, les seules fonctions définissables sont les fonctions constantes 0 et 1, qui peuvent être considérées comme complètement expansées.
Au rang $n-1$, on suppose que toute fonction de $\{0,1\}^{n-1}$ dans $\{0,1\}$ admet S_{n-1} expansions complètes. Au rang n, toute fonction de $\{0,1\}^n$ dans $\{0,1\}$ peut être expansée selon l'une des n variables de son domaine. Pour chaque variable, les cofacteurs associés admettent alors chacun S_{n-1} expansions possibles, donc il existe en tout $S_n = n.S_{n-1}^2$ expansions complètes de f.

D'après ce résultat, toute fonction booléenne admet au moins une expansion de Shannon complète. On peut alors définir une *expansion canonique* associant à toute fonction booléenne f une *unique* expansion complète de f. Nous étudions ci-dessous comment une telle fonction peut être obtenue.

11.1.3 Expansion de Shannon canonique

On considère l'ensemble des fonctions booléennes de $\{0,1\}^n$ dans $\{0,1\}$, définies sur les variables $X_n = \{x_1,\ldots,x_n\}$ $(n \in N^*)$. Nous avons vu que toute expansion de Shannon Λ_{n,x_i} de ces fonctions par rapport à la variable x_i est canonique. Donc si on choisit une unique variable $x_{\pi(n)}$ pour expanser toutes les fonctions de $\{0,1\}^n$ dans $\{0,1\}$, l'expansion complète de ces fonctions sera canonique si et seulement si l'expansion complète de leurs cofacteurs est elle-même canonique. Or ces derniers étant des fonctions de $\{0,1\}^{n-1}$ dans $\{0,1\}$ définies sur les variables $X_{n-1} = X_n \setminus \{x_{\pi(n)}\}$, on peut leur appliquer le même argument de choix d'une variable d'expansion unique.

Finalement, une expansion canonique complète des fonctions de $\{0,1\}^n$ dans $\{0,1\}$ sur les variables $\{x_1,\ldots,x_n\}$ peut être obtenue en associant pour tout i $(1 \le i \le n)$ une variable unique $x_{\pi(i)}$ à l'expansion de Shannon des fonctions de $\{0,1\}^i$ dans $\{0,1\}$ et en définissant une représentation canonique des constantes booléennes 0 et 1.
La définition de π est celle d'une permutation:

$$\pi : \begin{cases} \{0,\ldots,n\} & \longmapsto & \{0,\ldots,n\} \\ i & \longrightarrow & \pi(i) \ / \ \forall\,(i,j),\ i \neq j \Rightarrow \pi(i) \neq \pi(j) \end{cases}$$

Toute permutation π définit alors un *ordre d'expansion* des variables $\prec_\pi$:

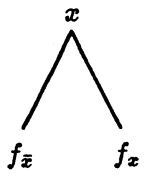


Figure 11.1: Représentation graphique de l'expansion de Shannon

$$\forall\, i \neq j,\ x_i \prec_\pi x_j \iff \pi(i) < \pi(j)$$

Par la suite, toute relation d'ordre sur les variables sera simplement notée $\prec$. Comme il existe $n!$ permutations des n premiers entiers, il existe $n!$ ordres d'expansion des variables définissant une expansion canonique complète des fonctions booléennes de $\{0,1\}^n$ dans $\{0,1\}$. Par conséquent, celle-ci est canonique *modulo* l'ordre d'expansion des variables.

11.1.4 Arbre de Shannon

Il est possible d'associer à toute expansion de Shannon d'une fonction booléenne f par rapport à une variable x une représentation graphique appelée *arbre de Shannon* de f. En effet, on peut associer à cette expansion un arbre à un nœud et deux feuilles, comme indiqué par la figure 11.1. Le nœud de l'arbre dénote alors la variable x et, *par convention*, la feuille droite (*resp.* gauche) de l'arbre est le cofacteur de f par rapport à x (*resp.* $\bar{x}$). Globalement, cet arbre dénote l'expansion de Shannon de f par rapport à x, c'est à dire l'expression $\neg x \wedge f_{\bar{x}_i} \vee x \wedge f_{x_i}$. De plus, il dénote f de façon unique car le couple $(f_{\bar{x}_i}, f_{x_i})$ est unique. Intuitivement, un tel arbre s'interprète donc de la manière suivante: "**non** x_i **et** $f_{\bar{x}_i}$ **ou** x_i **et** f_{x_i}" ou, plus simplement: "**si** x_i **alors** f_{x_i} **sinon** $f_{\bar{x}_i}$".

En ce qui concerne les fonctions constantes, la représentation qui leur est associée est la valeur qu'elles délivrent, soit 0 et 1.

La représentation de l'expansion de Shannon sous forme d'arbre de Shannon peut alors être itérée aux expansions de Shannon des cofacteurs $f_{\bar{x}_i}$ et f_{x_i} de f, et ainsi de suite jusqu'aux constantes 0 et 1. En fin de compte, la représentation obtenue est un arbre de Shannon *complet* dénotant l'expansion complète de f. Si de plus l'expansion de f est canonique, son arbre de Shannon l'est aussi. Un exemple d'arbre de Shannon canonique complet d'une fonction booléenne est donné dans la figure 11.2.

Les arbres de Shannon peuvent aussi être représentés de manière textuelle. Nous définissons ci-dessous une syntaxe contenant les arbres de Shannon des fonctions définies sur $\{x_1, \ldots, x_n\}$. Le vocabulaire du langage est composé des symboles $x_1, \ldots, x_n$ dénotant les variables du domaine, des constantes 0 et 1 et du symbole Λ dénotant l'expansion de Shannon. Les règles syntaxiques sont les suivantes:

$$\begin{aligned} < \textit{arbre} > &::= \Lambda(< \textit{var} >, < \textit{arbre} >, < \textit{arbre} >) \\ &\quad\ |\ \ 0 \mid 1 \\ < \textit{var} > &::= x_1 \mid \ldots \mid x_n \end{aligned}$$

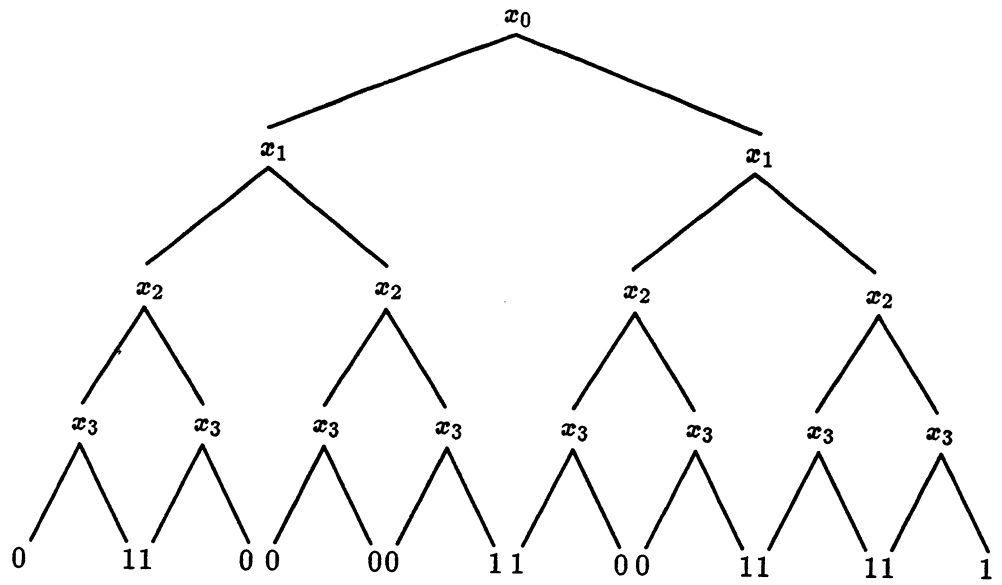


Figure 11.2: Arbre de Shannon de "**si** x_1 **alors** $(x_0 \vee x_2 \wedge x_3)$ **sinon** $(x_0 = (x_2 = x_3))$"

Les structures $\Lambda(\ldots)$ correspondent aux nœuds de l'arbre, tandis que les constantes 0 et 1 correspondent aux feuilles. La syntaxe ci-dessus sera utilisée dans la suite afin de permettre une manipulation formelle plus aisée des arbres de Shannon.

L'expansion complète de toute fonction f de $\{0,1\}^n$ dans $\{0,1\}$ s'obtient à partir de $2^n - 1$ expansions de Shannon de cette fonction et de ses cofacteurs successifs. L'arbre de Shannon représentant l'expansion complète de f est par conséquent constitué de $2^n - 1$ nœuds et 2^n feuilles. Si on assimile sa taille au nombre de nœuds qu'il possède, cet arbre est de taille *exponentielle*. Les arbres de Shannon ne constituent donc pas une représentation efficace des fonctions booléennes. Nous allons voir maintenant comment on peut les transformer pour les rendre plus compacts.

11.2 Les BDDs

Les BDDs (de l'anglais *Binary Decision Diagrams*) sont une structure de données introduite à l'origine par Akers [Ake78] pour la représentation des fonctions booléennes. Ils ont été popularisés par les travaux de Bryant [Bry86], qui a défini des algorithmes efficaces implémentant les principaux opérateurs booléens de manière symbolique sur ce type de représentation.

Les BDDs sont basés sur l'expansion de Shannon canonique complète des fonctions booléennes. Etant donné un ensemble de variables et un ordre d'expansion sur ces variables, toute fonction f sur ces variables admet une expansion canonique complète. De même, f admet une représentation unique sous forme d'arbre de Shannon complet. Le BDD de f est alors simplement obtenu par deux opérations de réduction sur cet arbre. Nous détaillons ces deux opérations ci-dessous.

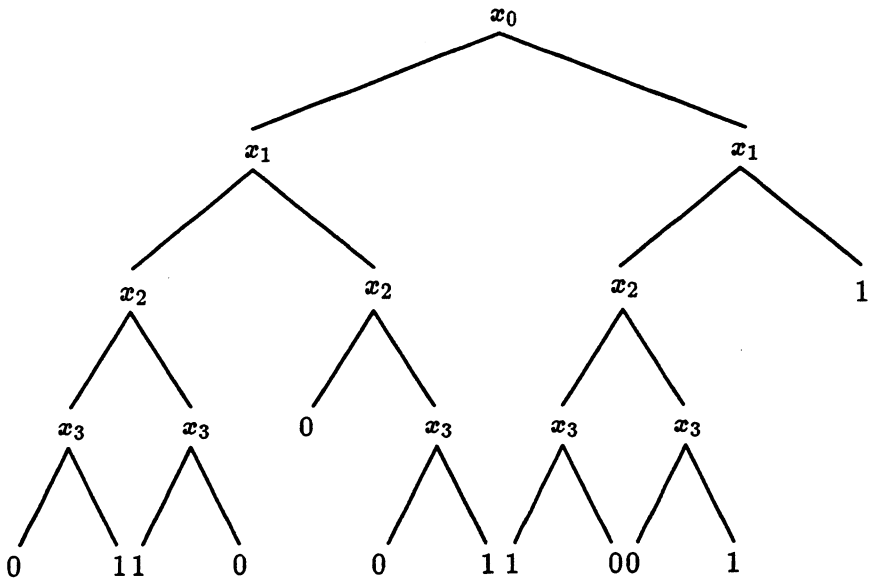


Figure 11.3: Arbre réduit de Shannon de "**si** x_1 **alors** $(x_0 \vee x_2 \wedge x_3)$ **sinon** $(x_0 = (x_2 = x_3))$"

11.2.1 Elimination des nœuds redondants

Quelquefois, l'expansion de Shannon $(f_{\bar{x}_i}, f_{x_i})$ d'une fonction f par rapport à une variable x_i peut être telle que $f_{\bar{x}_i} = f_{x_i}$, ce qui traduit le fait que f est indépendante de x_i. Dans ce cas, on a $f_{\bar{x}_i} = f_{x_i} = f$. L'expansion de Shannon de f par rapport à x_i est f: elle est donc inutile. Au niveau de l'arbre de Shannon complet de f, cette expansion correspondra à un nœud du type $\Lambda(x_i, F, F)$, où F est l'arbre de Shannon complet de f. Ce nœud peut donc être remplacé par F. Par conséquent, tout nœud du type $\Lambda(x_i, F, F)$ peut être éliminé en appliquant la règle de réécriture suivante aux arbres de Shannon complets:

$$\Lambda(x_i, F, F) \rightarrow F$$

Cette transformation constitue la première opération de réduction des arbres de Shannon complets. Son application permet de transformer tout arbre complet en arbre *réduit*. La figure 11.3 représente l'arbre réduit obtenu à partir de l'exemple de la figure 11.2.

11.2.2 Partage des sous-arbres isomorphes

Tout arbre réduit de Shannon peut à son tour être compacté par *partage* de ses sous-arbres isomorphes. Cette opération peut être formalisée de la manière suivante: étant donné un arbre réduit de Shannon quelconque, supposons qu'on dispose d'une fonction *Index* parcourant l'arbre en associant à chacun de ses nœuds ou feuilles un indice unique. Lorsque tous les nœuds sont indicés, le partage des sous-arbres isomorphes s'effectue par application des règles de réécriture ci-dessous:

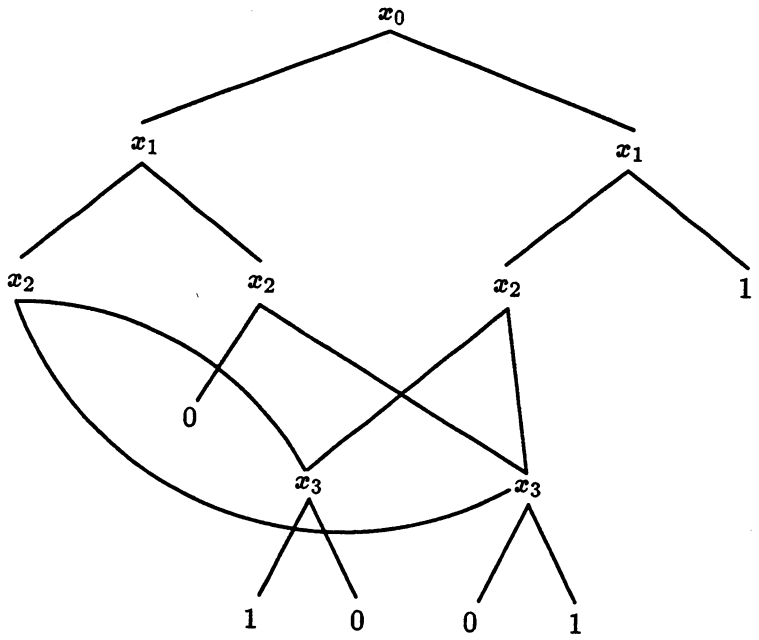


Figure 11.4: BDD de "**si** x_1 **alors** $(x_0 \vee x_2 \wedge x_3)$ **sinon** $(x_0 = (x_2 = x_3))$"

$$\begin{array}{lll} F = F' = 0, & Index(F) < Index(F'), & F' \rightarrow F \\ F = F' = 1, & Index(F) < Index(F'), & F' \rightarrow F \\ L = L', H = H', & Index(\Lambda(x, L, H)) < Index(\Lambda(x, L', H')), & \Lambda(x, L', H') \rightarrow \Lambda(x, L, H) \end{array}$$

L'opérateur "=" utilisé dans les règles désigne le prédicat d'égalité syntaxique entre arbres de Shannon complets.

A l'issue de cette deuxième opération de réduction, tout arbre réduit de Shannon est transformé en un graphe orienté acyclique (ou DAG [ASU86]: *Directed Acyclic Graph*), qui est un BDD. La figure 11.4 représente le BDD obtenu à partir de l'exemple de la figure 11.3 par partage des sous-arbres isomorphes. Cependant, pour la clarté de la figure, les feuilles du BDD ne sont pas partagées. Il en sera de même pour les figures ultérieures.

Toute fonction booléenne possède donc une représentation canonique sous forme de BDD, *modulo* un ordre fixé d'expansion des variables. De plus, les deux opérations de réduction décrites précédemment assurent manifestement une réduction de la taille des BDDs par rapport aux arbres de Shannon complets. Cependant, la question reste de savoir le facteur de cette réduction. La réponse est donnée par le théorème [Cou91] suivant:

Théorème 3 *Soit f une fonction quelconque de $\{0,1\}^n$ dans $\{0,1\}$. Si $|f|$ dénote la taille du BDD de f, c'est-à-dire le nombre de nœuds qui le constituent, on a alors*

$$|f| = O(2^n/n)$$

Ce résultat démontre que dans le pire cas, la taille du BDD dénotant f peut être exponentielle. Les BDDs ne peuvent être considérés comme une forme compacte de représentation. Néanmoins,

il s'avère en pratique que moyennant certaines précautions qui seront discutées ultérieurement, la plupart des fonctions booléennes peuvent être représentées par des BDDs de taille non exponentielle.

11.2.3 Opérateurs booléens sur les BDDs

La technique de représentation des fonctions booléennes recherchée doit non seulement être compacte, mais aussi permettre d'implémenter efficacement le calcul booléen. A ce sujet, les BDDs répondent de nouveau à ces besoins, puisque les principales opérations booléennes peuvent être implémentées sous forme d'opérateurs "symboliques" sur les BDDs. Ces opérateurs sont dits symboliques car leur résultat est généré par manipulation des représentations *syntaxiques* de leurs opérandes, sans évaluation de ces dernières.

Nous donnons ci-dessous la liste des principaux opérateurs booléens qui peuvent être implémentés sur les BDDs:

- **Opérateurs de base:**
  Les opérateurs booléens de base: négation, somme, produit sont implémentés sous forme d'opérateurs sur les BDDs [Bry86], respectivement unaire pour la négation et binaires pour la somme et le produit. A partir de ces opérateurs, on peut bien sûr implémenter tous les autres opérateurs du calcul booléen: somme exclusive, implication, équivalence, "si alors sinon", composition fonctionnelle, etc...

- **Quantification par rapport à une variable:**
  Etant donnée une fonction $f(x_1, \ldots, x_n)$ et x_i une variable quelconque de f, la quantification existentielle et la quantification universelle de f par rapport à x_i sont notées respectivement $\exists x_i\ f$ et $\forall x_i\ f$. Leur définition formelle est la suivante:

  $$\begin{aligned} \exists x_i\ f &= f_{\bar{x}_i} \vee f_{x_i} \\ \forall x_i\ f &= f_{\bar{x}_i} \wedge f_{x_i} \end{aligned}$$

  Ces opérations s'implémentent facilement sur les BDDs [McG89].

- **Quantification par rapport à un ensemble de variables:**
  Les opérateurs de quantification ci-dessus peuvent être généralisés à des ensembles de variables quelconques. Etant donnée une fonction $f(x_1, \ldots, x_n)$ et $X = \{x_{i_1}, \ldots, x_{i_m}\} \subset \{x_1, \ldots, x_n\}$, la quantification existentielle et la quantification universelle de f par rapport à X sont notées respectivement $\exists X\ f$ et $\forall X\ f$. Ces opérations sont définies formellement comme suit:

  $$\begin{aligned} \exists X\ f &= \exists x_{i_1}(\ldots(\exists x_{i_m}\ f)\ldots) \\ \forall X\ f &= \forall x_{i_1}(\ldots(\forall x_{i_m}\ f)\ldots) \end{aligned}$$

- **Calcul de prédicats:**
  Le calcul de prédicats unaires tels que le test de satisfaisabilité, le test de tautologie et l'évaluation de fonction, ainsi que le calcul de prédicats binaires tels que le test d'implication ou le test d'équivalence de deux fonctions peuvent aussi être implémentés sous forme d'opérateurs sur les BDDs.

- **Opérateurs particuliers:**
  D'autres opérateurs booléens très intéressants peuvent être implémentés sur les BDDs. Citons entre autres le calcul du domaine de définition exact d'une fonction, ou la simplification de fonctions [Cou91,Ray91], qui seront décrits ultérieurement.

Le gros intérêt des BDDs réside en outre dans leur grande souplesse d'utilisation. En effet, il est facile de définir des opérateurs spécifiques permettant d'effectuer des manipulations "non standard" sur les fonctions booléennes. Citons à titre d'exemple tout genre de renommage de variables, ou la mise sous forme de produits d'une fonction, ou encore le calcul des impliquants premiers d'une fonction.

11.2.4 Exemple d'opérateur

Nous donnons ci-dessous une définition de l'opérateur $and\,(f,g)$ implémentant la conjonction booléenne sur deux BDDs f et g. Celui-ci est défini par l'ensemble de règles ci-dessous:

Règles terminales:
L'une d'elles s'applique à chaque fois qu'un des opérandes est une constante.

$$and(f,1) = f$$
$$and(1,g) = g$$
$$and(f,0) = 0$$
$$and(0,g) = 0$$

Règles d'inférence:
Elles s'appliquent dans les autres cas.

Soit $f = \neg x \wedge f_0 \vee x \wedge f_1$
Soit $g = \neg y \wedge g_0 \vee y \wedge g_1$

si $x \prec y$, $and(f,g) = reduction\,(\neg x \wedge and(f_0,g) \vee x \wedge and(f_1,g))$
si $y \prec x$, $and(f,g) = reduction\,(\neg y \wedge and(f,g_0) \vee y \wedge and(f,g_1))$
si $x = y$, $and(f,g) = reduction\,(\neg x \wedge and(f_0,g_0) \vee x \wedge and(f_1,g_1))$

Le fonctionnement effectif de l'opérateur est le suivant: il commence par "descendre" dans les BDDs f et g par application des règles d'inférence. Lorsqu'on atteint une feuille, une des règles terminales s'applique, et le résultat de l'opération est généré au fur et à mesure en "remontant" dans les BDDs de f et g.
Les règles d'inférence intègrent trois principes fondamentaux:

- Elles utilisent une propriété importante de l'expansion de Shannon, dite propriété d'orthogonalité, qui est la suivante:

  Si $f = \neg x \wedge f_0 \vee x \wedge f_1$
  et $g = \neg x \wedge g_0 \vee x \wedge g_1$
  $f \wedge g = \neg x \wedge (f_0 \wedge g_0) \vee x \wedge (f_1 \wedge g_1)$

Ainsi, si f et g sont expansées par rapport à la même variable x, leur produit est obtenu à partir du produit de leurs cofacteurs respectifs par rapport à x.

- Elles préservent l'ordre des variables dans le résultat généré.
- Elles maintiennent la canonicité du résultat généré par appel à une fonction de simplification et de partage des BDDs. Cette fonction est appelée *reduction*.

Pratiquement tous les opérateurs implémentés sur les BDDs sont basés sur ces deux principes: propriété d'orthogonalité et préservation de l'ordre. La plupart des opérateurs booléens vérifient la propriété d'orthogonalité, c'est pourquoi leur implémentation sous forme d'opérateurs sur les BDDs est si facile.

11.2.5 Implémentation efficace des opérateurs

La plupart des opérateurs sur les BDDs génèrent des résultats intermédiaires au cours de leur exécution. Ainsi, pour l'opérateur de produit booléen décrit précédemment, chaque application d'une règle d'inférence génère un résultat intermédiaire. L'idée à la base de l'implémentation efficace des opérateurs consiste alors à gérer ces résultats intermédiaires pour pouvoir les réutiliser ultérieurement, sans avoir besoin de refaire l'opération correspondante. Ainsi, le coût des opérateurs serait abaissé au niveau du temps de calcul. Cependant, la gestion de résultats intermédiaires est elle-même coûteuse, particulièrement du point de vue de la mémoire puisque la représentation des résultats doit être conservée. Cette gestion doit donc s'effectuer selon une stratégie de compromis entre le coût effectif des opérateurs et le coût de gestion des résultats intermédiaires.

Avant de décrire les différentes stratégies adoptées à ce niveau, essayons de caractériser les phénomènes induits par l'exécution d'un opérateur BDD quelconque. Considérons donc, pour un ordre des variables fixé, un opérateur op n-aire et les BDDs $f_1, \ldots, f_n, g$ tels que $g = op(f_1, \ldots, f_n)$. Au début de l'exécution, la mémoire contient nécessairement la représentation de $f_1, \ldots, f_n$. A la fin de l'exécution, la mémoire contient en plus la représentation de g. Au cours de l'exécution, la mémoire contient la représentation des $f_1, \ldots, f_n$ et d'un certain nombre de résultats intermédiaires générés par l'opérateur. Les stratégies de gestion de ces résultats sont extrêmement diverses, mais elles sont comprises entre les deux stratégies ci-dessous:

- Conservation seulement des résultats intermédiaires *indispensables* pour la poursuite de l'opération. Cette stratégie est donc *minimale* du point de vue des résultats conservés, donc de la taille $M_{\min}$ de la mémoire nécessaire à l'exécution de l'opérateur. Par contre, tous les résultats qui auraient pu être réutilisés mais qui ont été détruits vont devoir être regénérés à chaque fois, d'où un temps de calcul *maximal* $T_{\max}$.
- Conservation de *tous* les résultats intermédiaires générés par l'opérateur. Cette stratégie est donc *maximale* du point de vue des résultats conservés, donc de la taille $M_{\max}$ de la mémoire nécessaire à l'exécution de l'opérateur. En revanche, tout résultat généré est réutilisable autant de fois que nécessaire sans avoir à le regénérer, d'où un temps de calcul *minimal* $T_{\min}$.

Si M est la taille de la mémoire disponible au début de l'exécution de l'opérateur, on peut alors distinguer quatre cas qui vont dicter la stratégie de gestion des résultats intermédiaires à adopter:

- $M < M_{\min}$: l'exécution est impossible!
- $M = M_{\min}$: on est obligé d'adopter la stratégie "minimale", donc le temps d'exécution T et égal à $T_{\max}$.
- $M_{\min} < M < M_{\max}$: la stratégie adoptée ne pourra conserver qu'une partie des résultats intermédiaires. Donc, certaines opérations seront nécessairement refaites, d'où un temps d'exécution T vérifiant $T_{\min} < T < T_{\max}$.
- $M_{\max} \leq M$: on peut adopter la stratégie "maximale" et obtenir un temps d'exécution égal à $T_{\min}$.

En conséquence, le coût des opérateurs BDD dépend de la mémoire disponible au moment de leur exécution et de la stratégie utilisée pour gérer les résultats intermédiaires. Malheureusement, on ne peut savoir *a priori* la quantité de mémoire nécessaire à l'exécution des opérateurs, donc il est très difficile de changer de stratégie en fonction de la mémoire disponible. C'est la raison pour laquelle on adopte toujours la même stratégie pour chaque opérateur.

En pratique, les résultats intermédiaires des opérateurs sont gérés par des mécanismes de "mémoires caches". Un cache est une zone mémoire où sont conservés un certain nombre de résultats intermédiaires. Le contenu du cache est renouvelé au fur et à mesure que les résultats sont générés, les plus récents remplaçant les plus anciens. Ce principe s'applique parfaitement aux BDDs. En effet, le cache de chaque opérateur va fonctionner de telle sorte qu'à chaque nouvelle exécution d'un opérateur, il va progressivement se remplir des résultats intermédiaires générés par cet opérateur, qui ont une importante probabilité d'être réutilisés.
Les principaux paramètres d'un cache concernent le nombre maximum de résultats intermédiaires conservés, et la stratégie de remplacement de ces résultats. Ces paramètres doivent être choisis de façon à offrir le meilleur compromis entre l'accélération des opérateurs et le coût mémoire occasionnés par leur utilisation. On distingue deux cas:

- Opérateurs unaires: si possible, on évite de refaire plusieurs fois les mêmes opérations sur les sous-graphes partagés à l'intérieur du BDD opérande en se servant d'un mécanisme de marquage du graphe parcouru et d'un cache au niveau de chaque nœud de BDD. On arrive ainsi à linéariser le coût de la plupart des opérateurs unaires par rapport à la taille de leur opérande.
- Opérateurs n-aires: chaque opérateur possède son propre cache implémenté de manière tout à fait classique par une table de hash-code. Chaque entrée de la table mémorise les opérandes et le résultat de l'opération effectuée. En fonction de la fréquence d'utilisation de l'opérateur et de la taille et de la stratégie de gestion de la table, on peut atteindre des taux de collision très intéressants. Cependant, le taux de collision s'effondre rapidement à mesure que l'arité de l'opérateur considéré augmente. C'est pourquoi en pratique, ces caches ne sont pas utilisés au delà des opérateurs ternaires.

11.2.6 Coût des opérateurs

Nous venons de voir que le coût de certains opérateurs sur les BDDs n'est pas uniquement lié à la complexité théorique des algorithmes qui les implémentent, mais qu'il est aussi fonction de la mémoire disponible à l'exécution. Un bon compromis entre la diminution du temps de calcul et l'exploitation de la mémoire disponible consiste à associer aux opérateurs des caches de résultats intermédiaires. Lorsque chaque opérateur dispose d'un cache adapté (genre, taille et stratégie de gestion du cache), le coût des opérateurs s'établit ainsi [Cou91]:

Opération	Coût
$f = g$ (Prédicat d'égalité)	$O(1)$
$f \neq 0$ (Satisfaisabilité)	$O(1)$
$f = 1$ (Tautologie)	$O(1)$
$f \Rightarrow g$ (Prédicat d'implication)	$O(\|f\| \times \|g\|)$
Evaluation de $f(x_1, \ldots, x_n)$	$O(n)$
$\neg f$	$O(\|f\|)$
$f \vee g$	$O(\|f\| \times \|g\|)$
$f \wedge g$	$O(\|f\| \times \|g\|)$
si f alors g sinon h	$O(\|f\| \times \|g\| \times \|h\|)$
$\exists x\ f$	$O(\|f\|^2)$
$\forall x\ f$	$O(\|f\|^2)$
$\exists X\ f$	$O(2^{\sqrt{\|f\|}})$
$\forall X\ f$	$O(2^{\sqrt{\|f\|}})$
Taille de f	$O(\|f\|)$
Domaine de f	$O(\|f\|)$

En conclusion, le coût des opérateurs est donc à la fois fonction du temps de calcul et de la mémoire nécessaire pour générer le résultat. Or, le temps de calcul n'est pas borné tandis que la mémoire l'est, donc il vaut mieux diminuer le coût des opérateurs en mémoire plutôt qu'en temps de calcul. C'est pourquoi en pratique, on préférera toujours implémenter des opérateurs lents mais consommant peu de mémoire plutôt que des opérateurs rapides mais risquant de saturer celle-ci.

11.2.7 Importance de l'ordre

La canonicité des BDDs et le principe des opérateurs symboliques qui leur sont associés imposent le choix d'un ordre d'expansion *unique*, c'est à dire que l'ordre des variables dans les BDDs doit être le même pour toute fonction représentée. Nous étudions ici les conséquences de cette contrainte sur l'utilisation pratique des BDDs.

L'ordre des variables détermine une unique expansion de Shannon canonique associée à chaque fonction booléenne. Si on change d'ordre, l'expansion de Shannon obtenue va être différente. Il en est de même au niveau des arbres de Shannon et des BDDs. Suivant l'ordre choisi, la structure des BDDs représentant les fonctions va être modifiée, donc leur taille va varier aussi. Cette dernière remarque constitue une caractéristique fondamentale des BDDs: l'ordre des variables influence totalement la taille des BDDs. La figure 11.5 illustre ce phénomène sur un exemple

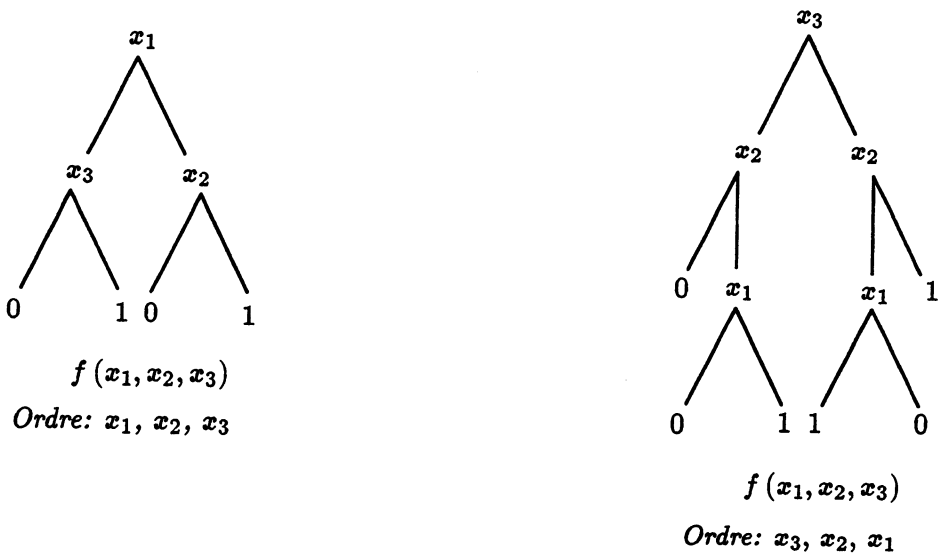


Figure 11.5: Influence de l'ordre dans les BDDs

simple, où la fonction à représenter est $f(x_1, x_2, x_3) = (\neg x_1 \wedge x_3) \vee (x_1 \wedge x_2)$. Cette figure exhibe deux représentations possibles de f sous forme de BDDs, utilisant des ordres différents.

Sachant que les fonctions booléennes doivent être représentées de façon la plus compacte possible, le choix de l'ordre des variables devient fondamental, et ce d'autant plus que le coût de la plupart des opérateurs BDDs est en général proportionnel à la taille de leurs opérandes (voir section 11.2.6). Donc l'ordre conditionne aussi les performances des BDDs en termes de temps d'exécution des opérateurs symboliques.

Par conséquent, l'ordre des variables ne peut être choisi aléatoirement: il doit permettre de minimiser le coût des BDDs en mémoire et en temps de calcul. D'un point de vue très pragmatique, le meilleur ordre est celui qui permet une représentation *suffisamment* compacte des BDDs pour rester dans les limites de la mémoire disponible, tout en minimisant le temps de calcul des opérations effectuées. Cependant, ce dernier paramètre est impossible à maîtriser. Par conséquent, le critère de choix d'un ordre ne peut être basé que sur la taille des BDDs générés. Compte tenu de l'influence de la taille des BDDs sur le coût des opérateurs, tout ordre déterminé selon ce critère possède quand même de grandes chances de minimiser ce coût.

Il faut donc choisir parmi tous les ordres possibles un ordre minimisant la taille des BDDs générés. Or, le nombre d'ordres possibles sur n variables est $n!$, donc le choix devient rapidement très vaste dés que n augmente. Cela oblige à définir des moyens pour effectuer ce choix, sachant que:

- On ne peut pas se permettre un choix aléatoire, compte tenu de l'importance de l'ordre.
- Tester tous les ordres est impossible pour des valeurs réalistes de n.
- Le critère de comparaison entre les ordres est la taille des BDDs générés. Le "meilleur" ordre est celui qui minimise cette taille.

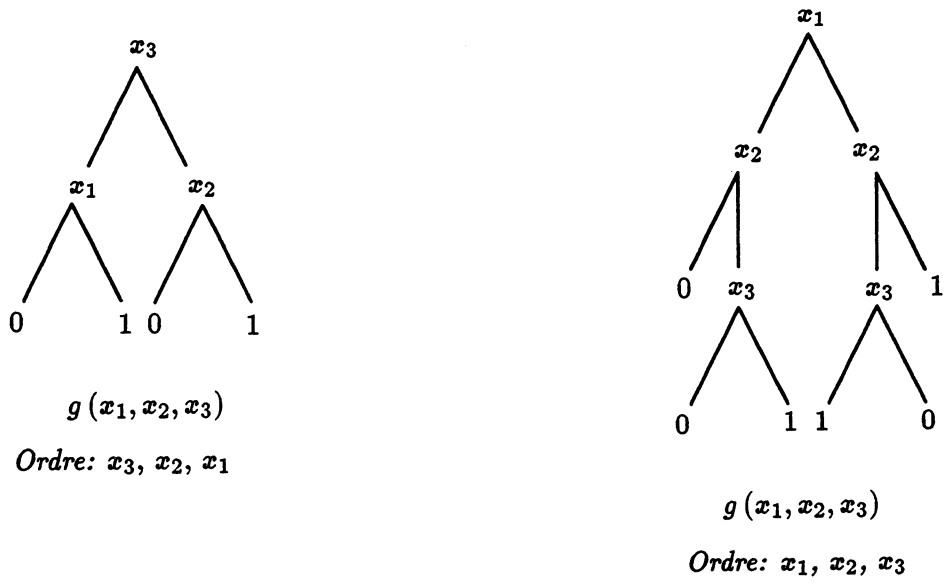


Figure 11.6: Incompatibilité entre ordres locaux

A ce niveau, le critère de choix d'un ordre est encore assez vague car il y a au moins deux interprétations possibles de ce que sont "les BDDs à minimiser":

- La première, que nous qualifierons de "locale", consisterait à considérer la minimisation d'un seul BDD.
- La seconde, dite "globale", serait basée sur la minimisation *d'un ensemble* de BDDs pris comme une seule entité. Le but ne serait plus de minimiser un BDD particulier mais la représentation globale correspondant à tous ces BDDs.

L'approche "locale" de l'ordonnancement de variables consiste, étant donnée une fonction booléenne f, à déterminer un ordre minimisant la taille du BDD de f. A ce niveau, il est nécessaire de rappeler que certaines fonctions booléennes n'admettent pas de représentation sub-exponentielle avec les BDDs, et ce quel que soit l'ordre choisi [Bry86]. Ce résultat met en évidence toute la difficulté de la recherche d'un "bon" ordre: le choix d'un ordre est indispensable pour utiliser les BDDs mais il n'existe parfois aucune solution satisfaisante à ce problème en pratique! Néanmoins, la taille de tout BDD étant bornée inférieurement, il existe des ordres qui en minimisent la taille. Ceux-ci sont alors *optimaux* et ils constituent le *meilleur* choix possibles dans l'approche "locale". Cependant, la recherche d'un ordre optimal est un problème *NP-difficile* [Bry88,Ber89a,Ber89b], à cause de la combinatoire intrinsèque du choix. Un algorithme a été proposé [SJF90], déterminant un ordre optimal pour la représentation d'une fonction booléenne donnée sous forme de BDD, mais celui-ci est d'un coût prohibitif car de complexité $O(n^2 3^n)$. C'est pourquoi on lui préfère des méthodes de choix heuristiques. Le résultat obtenu par ces méthodes est en général moins bon en termes d'optimalité, mais le coût du choix reste raisonnable.

L'approche "locale" du choix d'un ordre a été étudiée de façon intensive et de nombreuses

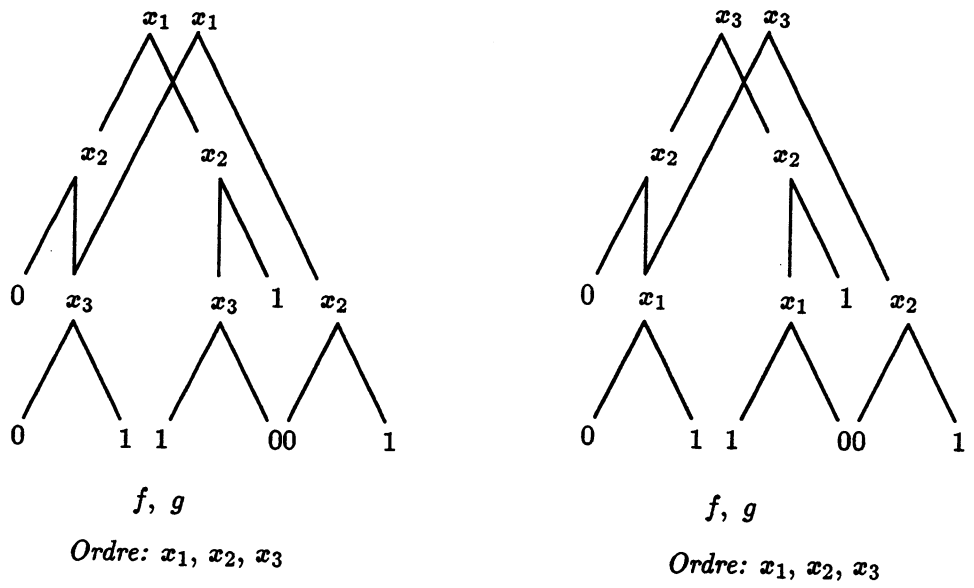


Figure 11.7: Représentation globale avec ordres locaux

méthodes heuristiques ont été proposées [MF88,Ber89a,Ber89b,SM88,NI91]. L'expérience montre qu'aucune d'elles n'est réellement meilleure que les autres [Cou91]. Une méthode peut s'avérer bien meilleure que les autres dans un cas précis et très mauvaise dans d'autres circonstances. A partir de ce constat, une nouvelle technique a été développée [DER90], rassemblant toutes les méthodes précédentes au sein d'une seule et même procédure de choix, dans le but discriminer les ordres proposés par chacune d'elles et de choisir le meilleur.

Les méthodes locales de choix d'un ordre permettent donc de déterminer un "bon" ordre pour la représentation d'une fonction booléenne donnée sous forme de BDD. Cependant, l'intérêt évident des BDDs n'est pas de permettre la représentation d'une seule fonction, mais de plusieurs. Si on adopte une approche locale pour déterminer un ordre, l'ordre ainsi trouvé pourra être optimal pour le BDD de la fonction ayant servi au choix, mais très mauvais pour les BDDs des autres fonctions. Ainsi, dans l'exemple illustré par la figure 11.6, si on utilise un ordre optimal de la fonction $f(x_1, x_2, x_3) = (\neg x_1 \wedge x_3) \vee (x_1 \wedge x_2)$ pour la représentation de la fonction $g(x_1, x_2, x_3) = (\neg x_3 \wedge x_1) \vee (x_3 \wedge x_2)$, le BDD obtenu pour g est deux fois plus gros que si on l'avait généré à partir d'un ordre optimal de g. Le défaut majeur de l'approche locale est donc de privilégier de manière généralement arbitraire la représentation d'une fonction particulière, sans tenir compte des autres fonctions devant être représentées.

Cette critique conduit à envisager une approche "globale" du choix d'un ordre, consistant à déterminer un ordre "moyen" pour la représentation d'un groupe de fonctions données. L'idée n'est alors plus de chercher à minimiser la représentation d'une fonction particulière, mais de minimiser la représentation globale du groupe de fonctions. Cette idée est illustrée par l'exemple de la figure 11.7 et de la figure 11.8. En utilisant l'ordre "local" optimal de f ou celui de g, la représentation globale de f et g est composée de sept nœuds. Par contre, en utilisant un ordre judicieusement choisi, la représentation globale de f et g tombe à six nœuds (cf. figure 11.8).

Curieusement, il n'existe pas de travaux sur cette approche, qui semble pourtant plus réaliste que l'approche locale. Nous avons donc développé une méthode basée sur cette approche, qui sera décrite ultérieurement.

En conclusion, le choix d'un "bon" ordre est un problème crucial dans l'utilisation des BDDs. La recherche d'un tel ordre est toujours difficile, et quelquefois sans solution acceptable! Il existe plusieurs méthodes pour effectuer cette recherche, mais elles sont toutes basées sur le même principe. Or, il semble que d'autres approches restent à explorer dans ce domaine.

11.2.8 Les TDGs

D'autres représentations des fonctions booléennes dérivées des BDDs ont été proposées [SM90, Bil87]. Nous ne nous intéressons ici qu'à l'une d'entre elles, qui est la seule à apporter une réelle amélioration par rapport aux BDDs: ce sont les TDGs [Bil87] (de l'anglais *Typed Decision Graphs*).

Les TDGs sont issus d'une propriété intéressante de l'expansion de Shannon concernant la négation des fonctions booléennes. Etant donnée une fonction f et x une variable de f, on a en effet (orthogonalité de la négation booléenne par rapport à l'expansion de Shannon):

$$\begin{aligned} f &= \neg x \wedge f_{\bar{x}} \vee x \wedge f_x \\ (\neg f) &= \neg x \wedge (\neg f_{\bar{x}}) \vee x \wedge (\neg f_x) \end{aligned}$$

L'expansion de Shannon de la fonction $(\neg f)$ par rapport à la variable x s'obtient donc à partir de celle de f par négation des cofacteurs de f. Cette règle s'applique aux cofacteurs de f, et ainsi de suite jusqu'aux fonctions constantes 0 et 1. Finalement, l'expansion complète de Shannon de $(\neg f)$ est identique à celle de f sauf pour les fonctions constantes 0 et 1, dont l'expansion correspond à une négation de leur valeur. Au niveau des arbres de Shannon, ces résultats signifient que l'arbre de $(\neg f)$ est identique à celui de f, sauf pour les feuilles 0 et 1, qui sont permutées. Par conséquent, l'arbre de $(\neg f)$ ne se distingue de l'arbre de f qu'au niveau de l'interprétation des feuilles. L'arbre de f muni d'une interprétation "positive" des feuilles dénote la fonction f. Cette interprétation, notée "+", est la suivante:

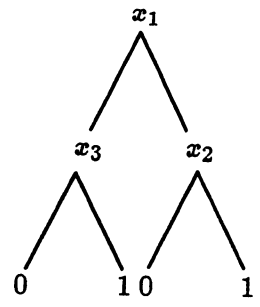
$$+0 = 0 \quad +1 = 1$$

L'arbre de f muni d'une interprétation "négative" des feuilles dénote la fonction $(\neg f)$. Cette interprétation, notée "−", est la suivante:

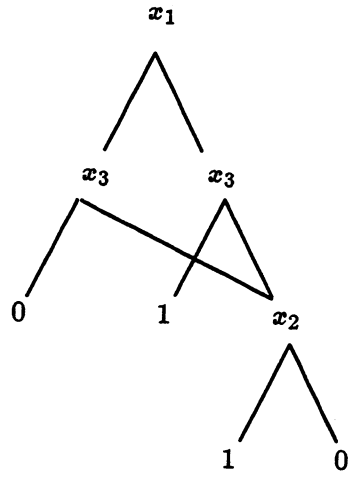
$$-0 = 1 \quad -1 = 0$$

L'idée introduite par Akers consiste alors à transformer les arbres de Shannon en arbres *typés* de Shannon en leur ajoutant une indication sur l'interprétation donnée aux feuilles. Ainsi, l'arbre typé de Shannon d'une fonction f s'obtient en associant le signe + à son arbre de Shannon. De même, celui de $(\neg f)$ s'obtient en associant le signe − à l'arbre de Shannon de f. Un arbre typé de Shannon est alors un couple (s, T), où $s \in \{+, -\}$ et T est un arbre de Shannon.

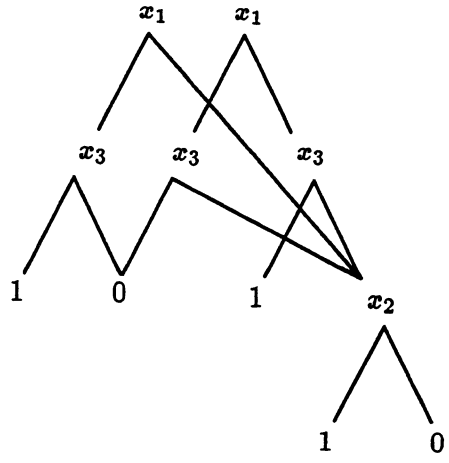
L'interprétation des arbres typés de Shannon se déduit donc de celle des arbres de Shannon de la façon suivante, si T est un arbre de Shannon quelconque:



f, *Ordre:* x_1, x_3, x_2



g, *Ordre:* x_1, x_3, x_2



f, *g*, *Ordre:* x_1, x_3, x_2

Figure 11.8: Représentation globale avec ordre global

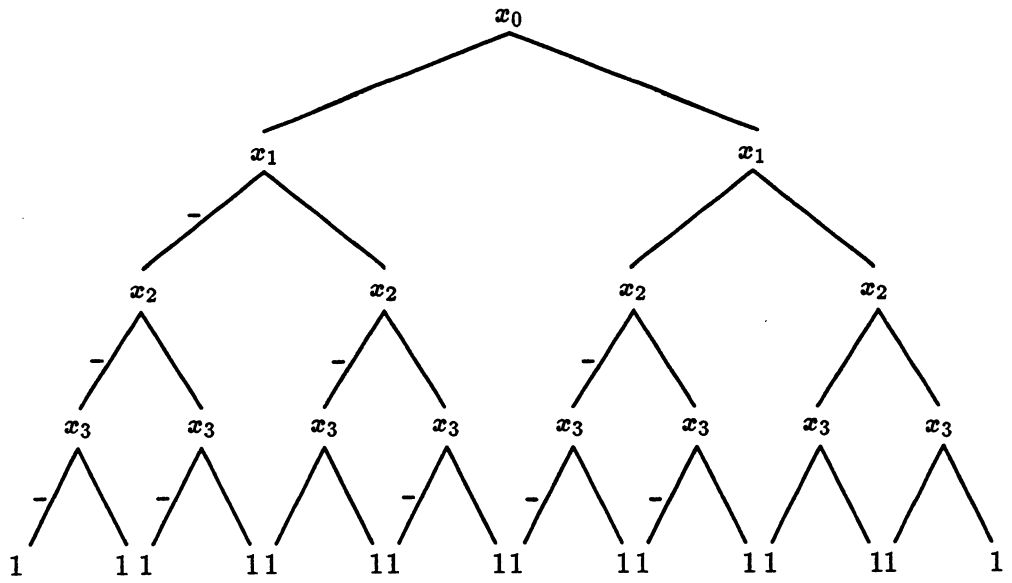


Figure 11.9: Arbre typé de Shannon de "**si** x_1 **alors** $(x_0 \vee x_2 \wedge x_3)$ **sinon** $(x_0 = (x_2 = x_3))$"

$$\begin{aligned}(+,T) &= T\\ (-,T) &= \neg T\end{aligned}$$

Cependant, l'introduction des signes détruit la canonicité des arbres typés de Shannon. L'exemple le plus simple de ce phénomène se situe au niveau des fonctions constantes. Ainsi, la fonction 0 possède deux représentations sous forme d'arbre typé de Shannon: $(+,0)$ et $(-,1)$. Afin de rétablir la canonicité dans les arbres typés de Shannon, on définit les règles de typage des arbres suivantes [JPB88]:

$$\begin{aligned}(+,0) &\rightarrow (-,1)\\ (-,0) &\rightarrow (+,1)\\ (+,\Lambda(x,(+,L),(-,H))) &\rightarrow (-,\Lambda(x,(-,L),(+,H)))\\ (+,\Lambda(x,(-,L),(-,H))) &\rightarrow (-,\Lambda(x,(+,L),(+,H)))\\ (-,\Lambda(x,(+,L),(-,H))) &\rightarrow (+,\Lambda(x,(-,L),(+,H)))\\ (-,\Lambda(x,(-,L),(-,H))) &\rightarrow (+,\Lambda(x,(+,L),(+,H)))\end{aligned}$$

Ces règles garantissent la canonicité des arbres typés de Shannon. Pratiquement, elles déterminent de manière unique le signe associé à chaque arbre de Shannon. D'un point de vue sémantique, elles interdisent, pour toute fonction f et toute variable x, d'associer une interprétation négative au cofacteur f_x. Graphiquement, on représente les arbres typés de Shannon en ajoutant le signe associé à chaque arbre de Shannon sur l'arc qui y conduit. Pour éviter d'alourdir la représentation, seuls les signes négatifs sont portés sur celle-ci. La figure 11.9 représente l'arbre typé de Shannon correspondant à notre exemple. On peut noter le gain de deux nœuds par rapport au BDD correspondant.

Les TDGs s'obtiennent à partir des arbres typés de Shannon en appliquant les mêmes

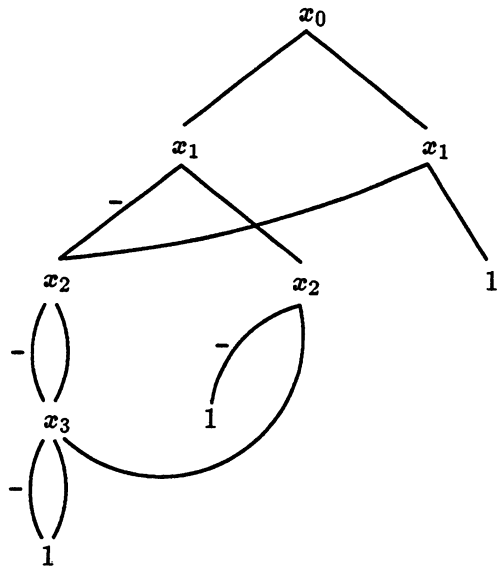


Figure 11.10: TDG de "**si** x_1 **alors** $(x_0 \vee x_2 \wedge x_3)$ **sinon** $(x_0 = (x_2 = x_3))$"

opérations de réduction permettant d'obtenir les BDDs à partir des arbres de Shannon. La figure 11.10 illustre le TDG obtenu à partir de l'arbre typé de Shannon de la figure 11.9.

Le gain essentiel des TDGs par rapport aux BDDs se situe tout d'abord au niveau de la représentation des fonctions booléennes, qui est plus compacte. Ainsi, un ordre des variables étant donné, le TDG d'une fonction est toujours de taille inférieure à celle de son BDD. Cependant, le facteur de réduction n'étant lié qu'au gain obtenu sur la représentation des fonctions et de leur négation, celui-ci ne peut être supérieur à 2. Les opérateurs sur les TDGs fonctionnent selon le même principe que les opérateurs sur les BDDs. Leur coût est donc tributaire de la taille des opérandes, mais comme les TDGs sont plus compacts que les BDDs, ils permettent un gain sur le temps de calcul de ces opérateurs. Le gain le plus spectaculaire concerne la négation booléenne, qui s'effectue en $O(1)$ pour toute fonction f avec les TDGs tandis qu'elle est en $O(|f|)$ avec les BDDs.

Les TDGs sont reconnus pour être une représentation plus efficace que les BDDs. Ce sont eux qui ont été effectivement implémentés pour la vérification des programmes LUSTRE, sous la forme d'une librairie optimisée [KSB90]. Néanmoins, nous continuerons à utiliser les BDDs dans la suite de l'exposé, car ceux-ci sont plus simples à manipuler formellement (pas de signe).

11.3 Encodage du modèle d'exécution par les BDDs

Nous avons vu qu'il est possible d'associer à tout programme LUSTRE un modèle d'exécution dénoté par un ensemble de variables d'état et de variables d'entrée et un ensemble de fonctions booléennes sur ces variables. On peut alors représenter ce modèle par des BDDs, afin d'en obtenir une représentation compacte et de bénéficier de tous les opérateurs sur les BDDs pour le

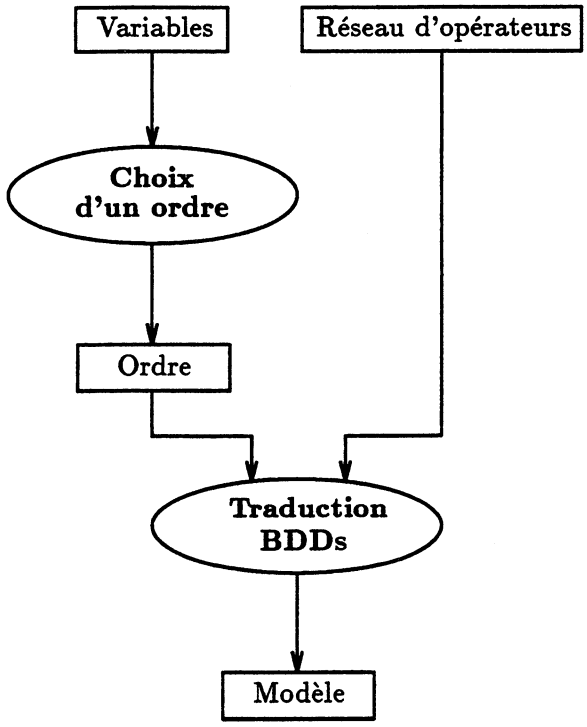


Figure 11.11: Traduction du modèle sous forme de BDDs

manipuler. Nous décrivons ici comment s'effectue la traduction du modèle sous forme de BDDs.

11.3.1 Génération compositionnelle

Nous rappelons que le modèle de tout programme est déterminé par le compilateur LUSTRE. Le modèle obtenu est représenté par le compilateur sous forme d'un ensemble de variables (les éléments de contrôle) et d'un réseau d'opérateurs dénotant un ensemble de fonctions booléennes sur ces variables, qui correspondent aux fonctions d'assertion, de sortie et de transition. La traduction du modèle sous forme de BDDs nécessite alors la mise en œuvre de deux tâches:

- Le choix d'un ordre sur les variables.
- La traduction du réseau d'opérateurs à l'aide des opérateurs sur les BDDs.

Le choix de l'ordre est un point qui sera discuté ultérieurement. On supposera toutefois qu'un ordre π a été fixé pour pouvoir traduire le réseau d'opérateurs sous forme de BDDs. L'obtention du BDD de toute fonction dénotée par un réseau d'opérateurs s'effectue alors par un parcours de ce réseau, au cours duquel le BDD de la fonction est généré de manière compositionnelle. Tous les opérateurs booléens LUSTRE se traduisent directement par des opérateurs sur les BDDs, ou par des combinaisons simples de ces opérateurs. La fonction *get_bdd* définie ci-dessous réalise la traduction de tout réseau d'opérateurs sous forme de BDD (l'opérateur *ite* utilisé dans les règles implémente l'opération booléenne "**si** ... **alors** ... **sinon** ..." sur les BDDs):

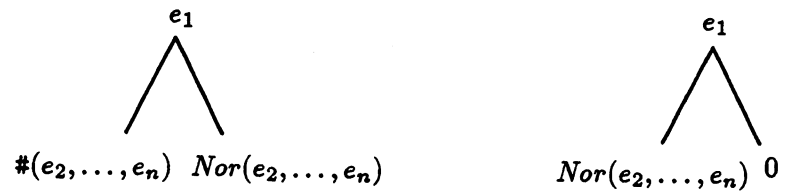


Figure 11.12: Représentation graphique de $\#(e_1,\ldots,e_n)$ et $Nor(e_1,\ldots,e_n)$

$$
\begin{array}{lcl}
get\_bdd(\texttt{false}) & = & 0 \\
get\_bdd(\texttt{true}) & = & 1 \\
get\_bdd(v_i) & = & ite(\pi(i),1,0) \\
get\_bdd(e_j) & = & ite(\pi(j),1,0) \\
get\_bdd(\texttt{not e}) & = & not(get\_bdd(\texttt{e}) \\
get\_bdd(\texttt{e1 or e2}) & = & ite(get\_bdd(\texttt{e1}),1,get\_bdd(\texttt{e2})) \\
get\_bdd(\texttt{e1 and e2}) & = & ite(get\_bdd(\texttt{e1}),get\_bdd(\texttt{e2}),0) \\
get\_bdd(\texttt{e1 = e2}) & = & ite(get\_bdd(\texttt{e1}),get\_bdd(\texttt{e2}),not(get\_bdd(\texttt{e2})) \\
get\_bdd(\texttt{e1 <> e2}) & = & ite(get\_bdd(\texttt{e1}),not(get\_bdd(\texttt{e2})),get\_bdd(\texttt{e2})) \\
get\_bdd(\texttt{if e1 then e2 else e3}) & = & ite(get\_bdd(\texttt{e1}),get\_bdd(\texttt{e2}),get\_bdd(\texttt{e3}))
\end{array}
$$

11.3.2 Cas de l'opérateur

L'opérateur # est un opérateur LUSTRE booléen n-aire qui n'admet pas de traduction simple sous forme de combinaison d'opérateurs BDDs. Il a en effet une sémantique particulière qui peut être intuitivement définie de la manière suivante: étant données n expressions booléennes $e_1,\ldots,e_n$, la valeur de l'expression booléenne $\#(e_1,\ldots,e_n)$ s'évalue à chaque instant à la valeur **true** si et seulement si au plus une expression e_i s'évalue à **true**. Donc, par induction:

- Si e_1 s'évalue à **true**, $\#(e_1,\ldots,e_n)$ s'évalue à **true** si et seulement si $e_2,\ldots,e_n$ s'évaluent toutes à **false**.
- Si e_1 s'évalue à **false**, la valeur de $\#(e_1,\ldots,e_n)$ est égale à celle de $\#(e_2,\ldots,e_n)$.

Cette définition inductive peut être traduite en termes d'opérateurs de la manière suivante:

$$
Nor : \left\{ \begin{array}{lcl} Nor(e) & = & \neg e \\ Nor(e_1,\ldots,e_n) & = & \neg e_1 \wedge Nor(e_2,\ldots,e_n) \end{array} \right.
$$

$$
\# : \left\{ \begin{array}{lcl} \#(e) & = & e \\ \#(e_1,\ldots,e_n) & = & e_1 \wedge Nor(e_2,\ldots,e_n) \vee \neg e_1 \wedge \#(e_2,\ldots,e_n) \end{array} \right.
$$

Par analogie avec les arbres de Shannon, les expressions $\#(e_1,\ldots,e_n)$ et $Nor(e_1,\ldots,e_n)$ peuvent être représentées sous forme d'arbres, comme indiqué dans la figure 11.12. En itérant cette représentation, on obtient le réseau d'opérateurs de la figure 11.13. Intuitivement, la sémantique

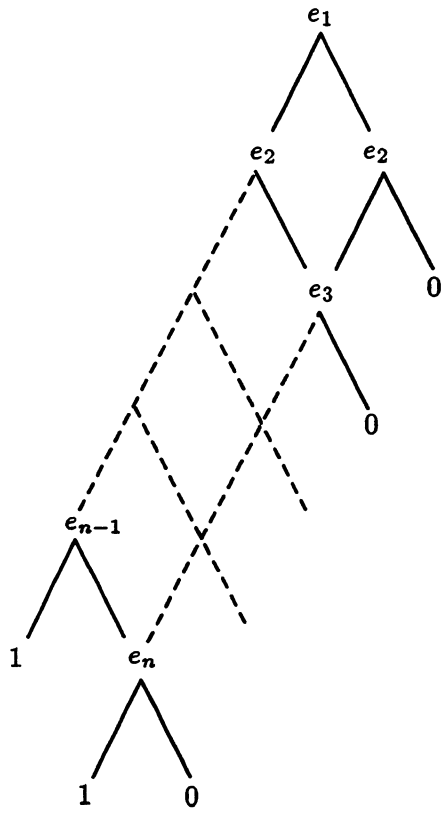


Figure 11.13: Réseau correspondant à $\#(e_1, \ldots, e_n)$

de l'opérateur # associe à chaque nœud du réseau l'interprétation "**si** e_i **alors** ... **sinon** ...". Le BDD correspondant à $\#(e_1, \ldots, e_n)$ peut donc être généré en simulant le parcours de ce réseau par la fonction *get_bdd*. Il faut remarquer que le réseau obtenu est linéaire par rapport au nombre d'opérandes de l'opérateur #. Cette propriété assure que le BDD de $\#(e_1, \ldots, e_n)$ peut toujours être généré en théorie.

Chapitre 12

Implémentation standard

Nous disposons maintenant d'une caractérisation formelle de la vérification des programmes LUSTRE sur leur modèle d'exécution, d'une méthode pour effectuer la vérification et d'une technique de représentation et de manipulation symbolique du modèle. Il reste par conséquent à réaliser un outil de vérification basé sur ces données.

La technique d'implémentation sélectionnée étant les BDDs, il faut tout d'abord définir un ordre des variables à l'intérieur de ces derniers. Il est ensuite nécessaire de disposer de tous les opérateurs permettant la manipulation du modèle, afin de pouvoir en définitive concevoir des algorithmes de vérification basés sur les BDDs. Dans ce chapitre, une première approche de ces problèmes est fournie.

12.1 Ordonnancement des variables

L'utilisation des BDDs pour implémenter la vérification des programmes LUSTRE nécessite de choisir un ordre des variables du modèle permettant une représentation compacte de celui-ci. Dans un premier temps, nous avons utilisé des méthodes préexistantes pour effectuer ce choix.

Nous avons tout d'abord utilisé l'ordre donné par le compilateur LUSTRE lorsqu'il détermine les variables d'état et les variables d'entrée du modèle associé à un programme. Cet ordre ne dépend que de la manière dont a été écrit le programme, donc il peut être considéré comme *aléatoire*. Sa seule particularité consiste dans le fait que toutes les variables d'état précèdent toutes les variables d'entrée dans l'ordre. Les différentes méthodes utilisées ultérieurement pour déterminer un ordre des variables "astucieux" pourront être comparées à cet ordre aléatoire afin d'évaluer leur efficacité. Dans un second temps, nous avons utilisé une méthode proposée par [NI91]. Celle-ci consiste en fait à *construire* un ordre, d'où son nom "d'ordonnancement" des variables. Nous décrivons ci-dessous le principe de cette méthode.

Nous avons déjà évoqué le problème que représente le choix d'un ordre des variables, ainsi que les conséquences de ce choix. Plusieurs heuristiques ont été proposées pour l'effectuer, mais celles-ci ne diffèrent les unes des autres que par les critères utilisés pour le choix d'un ordre. Autrement, leur principe de base est identique:

- Ce sont des méthodes d'ordonnancement "local": elles génèrent un ordre efficace des variables pour la représentation sous forme de BDD *d'une seule* fonction booléenne donnée.

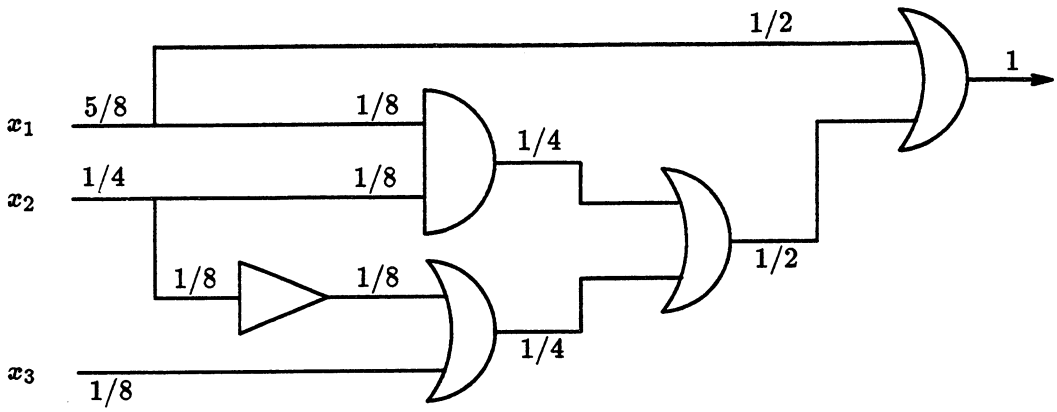


Figure 12.1: Pondération d'un réseau d'opérateurs

- Elles sont basées sur une analyse topologique de la fonction à représenter: l'ordre généré dépend uniquement de la représentation syntaxique de la fonction.

Devant cette relative uniformité parmi les méthodes existantes, notre choix s'est porté sur une méthode reconnue pour donner de bons résultats en général, et s'adaptant bien au calcul d'un ordre à partir de réseaux d'opérateurs booléens, car les fonctions du modèle nous sont fournies sous cette forme. La méthode proposée par [NI91] répondait convenablement à ces deux critères, c'est pourquoi nous l'avons choisie. En voici une description informelle.

Soit une fonction booléenne décrite par un réseau d'opérateurs booléens combinant un ensemble de variables et les constantes 0 et 1 entre elles. On suppose que le réseau n'est constitué que des opérateurs $\neg$, $\vee$, $\wedge$, ces deux derniers étant binaires. Cette hypothèse simplificatrice n'enlève rien à la généralité de la méthode, car on peut exprimer n'importe quelle fonction booléenne à partir de ces opérateurs. Les différentes entités du réseau sont connectées entre elles par des fils. Par la suite, tout fil portant le résultat d'un opérateur est appelé "sortie" de cet opérateur. Tout fil portant un opérande sera quant à lui appelé "entrée" de l'opérateur. L'ordonnancement est basé sur la "pondération" des variables du réseau. La pondération s'effectue par un parcours du réseau au cours duquel on associe une valeur réelle -appelée "poids"- à chaque fil du réseau. Les règles de pondération sont les suivantes:

- La sortie du réseau porte le poids 1.
- Si un opérateur $\neg$ porte un poids p sur sa sortie, on propage ce poids p sur son entrée.
- Si la sortie d'un opérateur $\vee$ ou $\wedge$ porte un poids p, on propage un poids $p/2$ sur chacune de ses entrées.
- Tout fil connecté à n fils de poids respectifs $p_1, \ldots, p_n$ se voit attribuer un poids $\Sigma_{i=1}^{n} p_i$.

La figure 12.1 donne un exemple d'application de ces règles. Lorsque la pondération est terminée, chaque variable du réseau possède un poids porté par le fil qui lui correspond. Remarquons que ce poids est compris entre 0 et 1. La variable dont le poids est le plus élevé est considérée comme

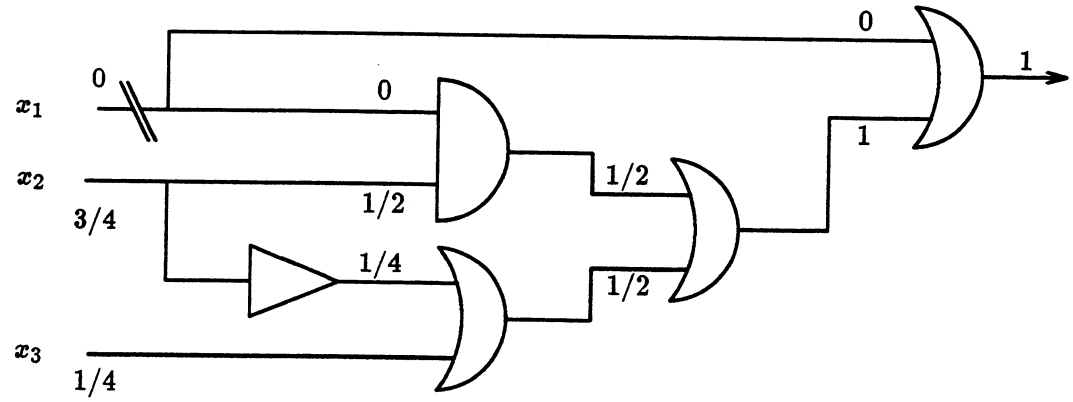


Figure 12.2: Pondération effective d'un réseau

la plus influente sur la sortie du réseau. C'est donc la plus discriminante sur le résultat de la fonction. Elle est donc placée en tête de l'ordre. Dans certains cas, plusieurs variables peuvent avoir le même poids maximal. Le choix de l'une d'elles comme variable la plus discriminante se pose alors. Diverses stratégies peuvent être adoptées, mais en général on se contente d'un choix aléatoire.

Pour ordonner les autres variables, la méthode va être itérée en ayant préalablement "coupé" dans le réseau le fil de la variable la plus influente. La raison de cette opération est intuitivement la suivante: le réseau initial dénotant une fonction f quelconque, nous appelons x la variable la plus influente du réseau. L'expansion de Shannon de f par rapport à x donne les cofacteurs (f_0, f_1), qui sont deux fonctions indépendantes de x. Or, si on coupe le fil de x dans le réseau, le nouveau réseau obtenu dénote une fonction indépendante de x, qui correspond en fait à $f_0 \vee f_1$. Par conséquent, en pondérant ce nouveau réseau, on déterminera la variable la plus discriminante sur $f_0 \vee f_1$. L'approximation heuristique consiste alors à considérer que cette variable risque d'être aussi la plus influente sur f_0 et sur f_1 considérées séparément. C'est pourquoi elle sera placée en seconde position dans l'ordre.

Le procédé de pondération est donc itéré jusqu'à ce que les fils de toutes les variables du réseau initial aient été coupés. Cela nécessite de prendre en compte les fils coupés à chaque nouvelle pondération. On appelle "réseau inerte" un réseau dont la sortie n'est influencée par aucune variable. Les réseaux dont tous les fils de variables ont été coupés et les constantes 0 et 1 sont des réseaux inertes. La pondération de ces réseaux doit associer un poids *nul* à chaque variable.

Le mécanisme de pondération est donc amélioré en associant des poids nuls aux parties inertes des réseaux parcourus. La figure 12.2 reprend l'exemple de la figure 12.1, où le réseau correspondant à la variable x_1 est inerte.

Finalement, la méthode d'ordonnancement topologique que nous venons de décrire est synthétisée par l'algorithme informel suivant:

tantque *le réseau n'est pas inerte* **faire**
debut

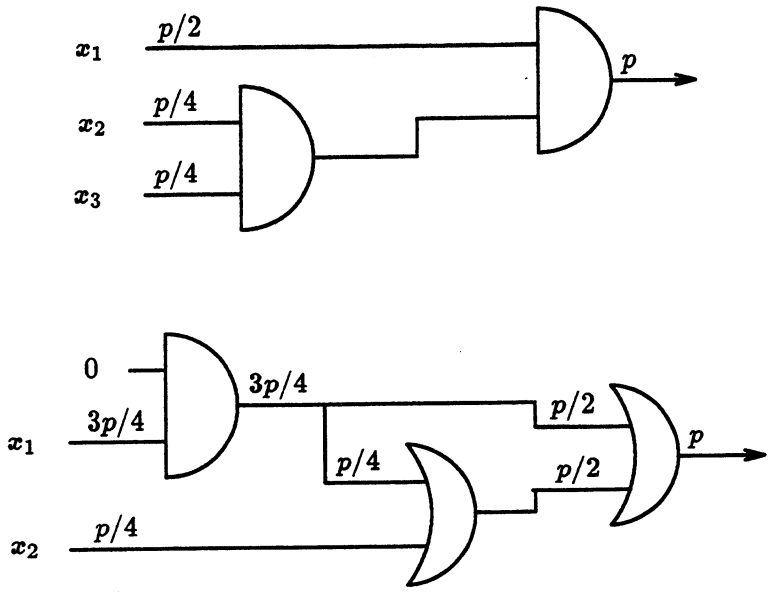


Figure 12.3: Les limites de la pondération topologiques

pondérer les variables connectées au réseau;
choisir la variable la plus discriminante;
placer cette variable dans l'ordre;
couper le fil de cette variable;
fin

Dans cette méthode, l'ordonnancement des variables est entièrement basé sur le mécanisme de pondération. Par conséquent, la qualité de l'ordonnancement dépend de la précision de ce mécanisme. Or, celui-ci a des limites qui peuvent facilement être mises en évidence sur des exemples simples. Ainsi, sur le premier réseau de la figure 12.3, on s'aperçoit que la pondération ne tient pas compte de la symétrie entre les opérandes. Sur le second réseau, la variable considérée comme la plus importante n'a en réalité aucune influence sur le résultat de la sortie. Le problème de la pondération est qu'elle ne s'effectue que par rapport à la topologie du réseau, sans aucune prise en compte de sa sémantique. Nous verrons ultérieurement comment on peut remédier partiellement à cette lacune.

12.2 Opérateurs de précondition

L'étude formelle de l'évaluation des assertions et des spécifications d'un programme sur son modèle d'exécution a révélé la nécessité de définir des opérateurs de précondition (voir section 5.3.3) sur les états du modèle. Ces opérateurs doivent par conséquent être implémentés sur les BDDs.

Les deux opérateurs de précondition Pre et $\widetilde{Pre}$ que nous avons défini sont respectivement

issus de la composition de l'opérateur *Acc* avec l'opérateur *Src*, et de la composition de l'opérateur *Acc* avec l'opérateur $\widetilde{Src}$. Les opérateurs *Src* et $\widetilde{Src}$ sont faciles à implémenter sur les BDDs car ils correspondent à une quantification par rapport aux entrées du modèle. Ce dernier étant encodé dans le domaine fonctionnel, si on suppose que les transitions sont encodées sur les variables d'état $x_1, \ldots, x_n$ et sur les variables d'entrée $e_1, \ldots, e_m$, alors tout ensemble de transition $T \subset Q \times E$ est dénoté par sa fonction caractéristique χ_T et on a, en utilisant les opérateurs de quantification sur les BDDs définis dans la section 11.2.3:

$$\chi_{Src(T)} = \exists\{e_1, \ldots, e_m\}\ \chi_T$$

et:

$$\chi_{\widetilde{Src}(T)} = \forall\{e_1, \ldots, e_m\}\ \chi_T$$

Donc les opérateurs *Src* et $\widetilde{Src}$ sont directement implémentables à l'aide des opérateurs standards sur les BDDs.

Il reste à implémenter l'opérateur *Acc* s'appliquant à tout ensemble d'états X d'un modèle $\mathcal{M} = (Q, E, S, q_{init}, \alpha, \sigma, \vec{\delta})$, et déterminant les transitions conduisant aux états de X:

$$Acc(X) = \{(q, e) \in Q \times E,\ \vec{\delta}(q, e) \in X\}$$

Le modèle étant encodé par des fonctions booléennes, on peut traduire cet opérateur ensembliste dans le domaine fonctionnel. L'ensemble X étant dénoté par sa fonction caractéristique χ_X, on obtient:

$$Acc(X) = \{(q, e) \in Q \times E,\ \chi_X(\vec{\delta}(q, e))\}$$

d'où on déduit la fonction caractéristique de $Acc(X)$:

$$\chi_{Acc(X)} = \chi_X(\vec{\delta}(q, e))$$

Dans le domaine booléen, l'opérateur *Acc* est donc défini par:

$$Acc : \begin{cases} (\{0,1\}^n \mapsto \{0,1\}) & \longmapsto & (\{0,1\}^{n+m} \mapsto \{0,1\}) \\ \chi_X(x_1, \ldots, x_n) & \longrightarrow & \chi_X(\delta_1, \ldots, \delta_n) \end{cases}$$

Nous étudions ci-dessous les différentes manières d'implémenter l'opérateur *Acc* avec les BDDs.

12.2.1 La méthode de la relation de transition

L'implémentation de l'opérateur *Acc* peut s'effectuer à partir de la relation de transition du modèle [SB90,EE91]. Etant donné un modèle $\mathcal{M}$, sa relation de transition de $\mathcal{T}$ caractérise l'ensemble des triplets (q, e, q') tels que (q, e) est une transition de q vers q'. Formellement, cette relation s'exprime par la fonction booléenne suivante:

$$\mathcal{T} : \begin{cases} Q \times E \times Q & \longmapsto & \{0,1\} \\ (q, e, s') & \longrightarrow & \mathcal{T}(q, e, q')\ /\ \mathcal{T}(q, e, q') = 1 \Longleftrightarrow \vec{\delta}(q, e) = q' \end{cases}$$

Cette relation peut être complètement transposée dans le domaine fonctionnel: les états q et q' étant encodés respectivement par les variables $x_1, \ldots, x_n$ et $y_1, \ldots, y_n$, et l'entrée e étant encodée par les variables $i_1, \ldots, i_m$, on obtient:

$$\mathcal{T}(x_1, \ldots, x_n, i_1, \ldots, i_m, y_1, \ldots, y_n) = \bigwedge_{k=1}^{n} (y_k = \delta_k(x_1, \ldots, x_n, i_1, \ldots, i_m))$$

$\mathcal{T}$ s'exprime donc facilement à l'aide des opérateurs booléens de base: négation, somme et produit. Le calcul de la relation de transition d'un modèle ne requiert donc pas d'opérateur spécialisé. Par conséquent, il est très facile de l'implémenter à l'aide des opérateurs BDD de base.

Une fois obtenue la relation de transition $\mathcal{T}$ du modèle, l'opérateurs *Acc* s'exprime alors facilement à partir de celle-ci. En effet, les prédecesseurs de X sont tous les états q pour lesquels il existe un triplet (q, e, q') de $\mathcal{T}$ tel que $q' \in X$, ce qui s'exprime simplement par:

$$Acc(\chi_X(x_1, \ldots, x_n)) = \exists\{y_1, \ldots, y_n\}\ \chi_X(y_1, \ldots, y_n) \wedge \mathcal{T}(x_1, \ldots, x_n, i_1, \ldots, i_m, y_1, \ldots, y_n)$$

Les seuls opérateurs nécessaires pour implémenter l'opérateur *Acc* sont donc le produit booléen et la quantification existentielle sur un ensemble de variables, qui sont des opérateurs de base sur les BDDs. Par conséquent, à partir de $\mathcal{T}$le calcul de prédécesseurs d'un ensemble d'états peut facilement être implémenté sur les BDDs.

12.2.2 La méthode directe

La seconde façon d'implémenter l'opérateur *Acc* consiste en une traduction directe de sa définition:

$$Acc(\chi_X(x_1, \ldots, x_n)) = \chi_X(\delta_1, \ldots, \delta_n) = \chi_X \circ \vec{\delta}$$

Le calcul des prédécesseurs d'un ensemble d'états peut donc être effectué par *composition fonctionnelle* entre la fonction caractéristique de cet ensemble et la fonction de transition du modèle. La composition fonctionnelle est une opération particulière, qu'il faut alors implémenter sous forme d'opérateur BDD.

Définition 3 *Etant donnée une fonction $g(x_1, \ldots, x_n)$ et un vecteur fonctionnel $\vec{f} = [f_1, \ldots, f_n]$, la composition fonctionnelle de g par $\vec{f}$ est la fonction*

$$g(x_1, \ldots, x_n) \circ \vec{f} = g(f_1, \ldots, f_n)$$

Sur des formes de représentation non canoniques, cette opération correspond simplement à substituer chaque occurrence des variables x_i dans g par les fonctions f_i. Cependant, la même opération effectuée sur des formes canoniques est beaucoup plus complexe, car le résultat de la substitution doit lui-même être mis sous forme canonique. Ainsi, la composition fonctionnelle est une opération *NP-difficile* [Cou91] sur les BDDs. Compte tenu de ce résultat théorique inquiétant, il est indispensable d'implémenter la composition fonctionnelle sur les BDDs de la manière la plus efficace possible. Nous présentons dans un premier temps l'implémentation classique de cet opérateur.

Etant donnée une fonction $g(f_1, \ldots, f_n)$ et un vecteur fonctionnel $\vec{f} = [f_1, \ldots, f_n]$, la composition fonctionnelle vérifie la propriété d'orthogonalité de l'expansion de Shannon, donc:

$$\forall\, 1 \leq i \leq n,\ g \circ \vec{f} = \neg f_i \wedge g\,(f_1, \ldots, 0, \ldots, f_n) \vee f_i \wedge g\,(f_1, \ldots, 1, \ldots, f_n)$$

Or d'après l'expansion de Shannon de g par rapport à x_i, les cofacteurs $g_{\bar{x}_i}$ et g_{x_i} sont définis de la manière suivante:

$$g_{\bar{x}_i} = g\,(x_1, \ldots, 0, \ldots, x_n)$$
$$g_{x_i} = g\,(x_1, \ldots, 1, \ldots, x_n)$$

On en déduit donc que

$$\begin{array}{lrcl} \forall\, 1 \leq i \leq n,\ g \circ \vec{f} & = & \neg f_i & \wedge \quad g_{\bar{x}_i}\,(f_1, \ldots, f_{i-1}, f_{i+1}, \ldots, f_n) \\ & \vee & f_i & \wedge \quad g_{x_i}\,(f_1, \ldots, f_{i-1}, f_{i+1}, \ldots, f_n) \end{array}$$

Cette règle donne une définition inductive de la composition fonctionnelle. En effet, elle s'applique au calcul de $g_{\bar{x}_i} \circ \vec{f}'$ et $g_{x_i} \circ \vec{f}'$, où $\vec{f}' = [f_1, \ldots, f_{i-1}, f_{i+1}, \ldots, f_n]$, et ainsi de suite jusqu'aux fonctions constantes 0 et 1, pour lesquelles on a:

$$\forall\, \vec{f},\ g = 0 \ \Rightarrow\ g \circ \vec{f} = 0$$
$$\forall\, \vec{f},\ g = 1 \ \Rightarrow\ g \circ \vec{f} = 1$$

A partir de ces règles, on obtient directement un opérateur de calcul de la composition fonctionnelle sur les BDDs, appelé *compose*:

Règles terminales:

$$compose\,(0, \vec{f}) = 0$$
$$compose\,(1, \vec{f}) = 1$$

Règle d'inférence:

soit $g = \neg x_i \wedge g_{\bar{x}_i} \vee x_i \wedge g_{x_i}$ **et** $\vec{f} = [f_1, \ldots, f_n]$
$$compose\,(g, \vec{f}) = ite\,(f_i, compose\,(g_{x_i}, \vec{f}'), compose\,(g_{\bar{x}_i}, \vec{f}'))$$

L'opérateur *ite* implémente l'opérateur "if then else" sur les BDDs. C'est lui qui cumule les deux aspects de la composition fonctionnelle sur les BDDs: substitution des variables x_i par les fonctions f_i et canonisation du résultat. Par ailleurs, dans l'opérateur *compose*, le vecteur fonctionnel passé en paramètre n'a pas besoin d'être modifié au cours des applications successives de la règle d'inférence. C'est pourquoi on peut définir un opérateur interne *compose_int* (g) *unaire* qui sera appelé par l'opérateur *compose* pour effectuer la composition fonctionnelle. Ce nouvel opérateur est défini comme suit:

Règles terminales:

$$compose\_int(0) = 0$$
$$compose\_int(1) = 1$$

Règle d'inférence:

$$compose\_int(\neg x_i \wedge g_{\bar{x}_i} \vee x_i \wedge g_{x_i}) = ite(f_i, compose\_int(g_{x_i}), compose\_int(g_{\bar{x}_i}))$$

Dans cette définition, $\vec{f} = [f_1, \ldots, f_n]$ correspond au second paramètre de l'opérateur *compose*. Cette modification de l'algorithme originel permet une implémentation optimisée de la composition fonctionnelle. En effet, on peut associer un cache de résultats intermédiaires à l'opérateur unaire *compose_int*, ce qui permet de réduire le nombre d'opérations effectuées durant ce calcul. Ainsi, le nombre d'appels à l'opérateur *ite* sera égal à la taille du BDD de g. Dans de telles conditions, la complexité de la composition fonctionnelle est en $O(|g|^2 \times \Pi_{k=1}^{n} |f_k|)$ [Cou91]. La composition fonctionnelle est donc une opération très coûteuse sur les BDDs.

12.2.3 Choix d'une implémentation sur les BDDs

Nous venons de décrire deux méthodes pour implémenter l'opérateur *Acc* sur les BDDs, ceci afin de disposer des opérateurs de précondition *Pre* et $\widetilde{Pre}$. Il faut maintenant déterminer laquelle est la plus appropriée à la vérification. En ce qui concerne la méthode de la relation de transition, une fois que cette dernière est générée, le calcul de n'importe quelle précondition est aisé, car il ne nécessite que deux opérations standards sur les BDDs. Mais le gros problème de cette méthode réside dans la génération de la relation de transition. En effet, le calcul effectif de cette relation sur les BDDs est déjà très coûteux en temps de calcul, puisque sa complexité est en $O(2^n \times \prod_{k=1}^{n} |\delta_k|)$ [Cou91]. Mais le coût de la représentation de cette relation est encore plus problématique. En effet, sa taille est en général explosive [HJT90] dés que le modèle atteint une dimension réaliste (nombre de variables d'état supérieur à 30). Cela s'explique au niveau des BDDs par le fait que la relation de transition nécessite $2n + m$ variables pour sa représentation, où n désigne le nombre de variables d'état et m le nombre de variables d'entrée, alors que toutes les autres fonctions sont représentées sur au plus $n+m$ variables. Sachant que les BDDs sont une représentation potentiellement exponentielle par rapport au nombre de variables, la difficulté que constitue la représentation de la relation de transition s'explique aisément. Plus intuitivement, le BDD de la relation de transition peut s'interpréter comme la représentation *en compréhension* du modèle généré par la méthode classique en avant. Or nous avons vu que cette méthode est souvent impossible à mettre en œuvre car le modèle peut devenir très gros. Il n'est donc pas surprenant de constater le même phénomène pour la génération de la relation de transition en technique symbolique.

Les arguments que nous venons d'évoquer contre la méthode de la relation de transition nous forcent donc à implémenter les opérateurs de précondition par la méthode directe.

12.3 Evaluation des assertions

Nous disposons à présent d'une représentation symbolique du modèle, et d'une implémentation de tous les opérateurs nécessaires à l'évaluation des assertions et des spécifications sur celui-ci. Il reste donc à définir et à implémenter les algorithmes permettant cette évaluation. Dans cette section, nous traitons l'évaluation des assertions.

12.3.1 Principe de l'évaluation

Nous avons vu qu'étant donné un modèle $\mathcal{M} = (Q, E, S, q_{init}, \alpha, \sigma, \vec{\delta})$ où α est la fonction d'assertion sur $\mathcal{M}$, l'ensemble des transitions valides de $\mathcal{M}$ est caractérisé par le point fixe suivant:

$$\alpha_v = \nu Y.\ \alpha \cap Acc(Src(Y))$$

D'autre part, nous avons établis les correspondances suivantes entre le domaine ensembliste et le domaine fonctionnel booléen:

- α définit une fonction booléenne de $Q \times E$ dans $\{0,1\}$.
- Pour tout ensemble d'états $X \subset Q$ du modèle, dénoté par sa fonction caractéristique χ_X, $Acc(X)$ correspond à la fonction $\chi_X \circ \vec{\delta}$ définie de $Q \times E$ dans $\{0,1\}$.

De plus, l'opérateur ensembliste $\cap$ correspondant au produit booléen entre fonctions caractéristiques, on obtient finalement une définition de point fixe dans le domaine booléen de l'ensemble des transitions valides du modèle:

$$\alpha_v = \nu y.\ [\alpha \wedge Src(y) \circ \vec{\delta}]$$

α_v est la fonction caractéristique de l'ensemble des transitions valides. Ce point fixe est alors classiquement la limite de la suite de fonctions définie ci-dessous:

$$\begin{cases} \alpha_0 &= \alpha \\ \alpha_{n+1} &= \alpha_n \wedge Src(\alpha_n) \circ \vec{\delta} \end{cases}$$

Le calcul effectif du point fixe revient donc au calcul des termes successifs de cette suite, jusqu'à convergence:

$$\begin{aligned} \alpha_0 &= \alpha \\ \alpha_1 &= \alpha_0 \wedge Src(\alpha_0) \circ \vec{\delta} \\ \vdots &= \vdots \\ \alpha_{k+1} &= \alpha_k \wedge Src(\alpha_k) \circ \vec{\delta} \end{aligned}$$

Au rang k, on a $\alpha_{k+1} = \alpha_k = \alpha_v$.

Le cas où les assertions sont non satisfaisables peut être pris en compte au fur et à mesure du calcul des termes de la suite. En effet, nous avons vu qu'il correspond à

$$\forall e \in E, (q_{init}, e) \notin \alpha_v$$

soit, dans le domaine fonctionnel:

$$[\widetilde{Src}(\neg\alpha_v)](q_{init})$$

ou encore:

$$\neg[Src(\alpha_v)](q_{init})$$

Compte tenu de la décroissance monotone de la suite calculée ($\forall i \in N, \alpha_{i+1} \Rightarrow \alpha_i$), il est possible de tester la satisfaisabilité des assertions à chaque calcul d'un nouveau terme. En tenant compte de toutes ces remarques, on peut maintenant définir un algorithme d'évaluation des assertions.

12.3.2 Algorithme d'évaluation

D'un point de vue global, le calcul du point fixe est implémenté sous la forme d'une boucle de calcul des termes successifs de la suite définie précédemment, dont la condition d'arrêt est la convergence de la suite, et dans laquelle la satisfaisabilité des assertions est testée à chaque cycle. Les notations adoptées sont les suivantes:

- α correspond à la fonction d'assertion du modèle.
- α_v est la fonction caractéristique des transitions valides.
- α_r est la fonction de référence à laquelle on va comparer α_v pour déterminer la convergence.
- α'_v est la fonction caractéristique des états valides.

L'algorithme ci-dessous traduit alors l'évaluation des assertions dans le domaine fonctionnel. Il est *directement implémentable* sur les BDDs.

debut
$\alpha_v = \alpha$
$\alpha_r = 1$
tantque $(\alpha_r \neq \alpha_v)$ **faire**
  debut
  $\alpha'_v = Src(\alpha_v)$
  si $\alpha'_v\ (q_{init})$
  alors
  debut
  $\alpha_r = \alpha_v$
  $\alpha_v = \alpha_v \wedge \alpha'_v \circ \vec{\delta}$
  fin
sinon
    retourner (`Assertions Non Satisfaisables`)
  fin
si $(\alpha_v \neq \alpha)$
alors
  afficher (`WARNING: Assertions Non Causales`)
fin

12.4 Evaluation des spécifications

Il reste maintenant à définir et à implémenter un algorithme permettant l'évaluation des spécifications. Ce problème est abordé dans les sections suivantes.

12.4.1 Principe de l'évaluation

Nous avons vu qu'étant donné un modèle $\mathcal{M} = (Q, E, S, q_{init}, \alpha, \sigma, \vec{\delta})$, où σ est la fonction exprimant les spécifications sur $\mathcal{M}$, l'ensemble des séquences d'exécution de $\mathcal{M}$ devant vérifier σ est caractérisé par le point fixe suivant:

$$\sigma_v = \nu Y.\ \overline{\alpha}_v \cup [\sigma \cap Acc(\widetilde{Src}(Y))]$$

D'autre part, nous avons établi les correspondances suivantes entre le domaine ensembliste et le domaine fonctionnel booléen:

- α_v est l'ensemble des transitions valides du modèle. Sa fonction caractéristique α_v est une fonction booléenne de $Q \times E$ dans $\{0,1\}$, calculée selon l'algorithme décrit précédemment.
- σ définit une fonction booléenne de $Q \times E$ dans $\{0,1\}$.
- Pour tout ensemble d'état $X \subset Q$ du modèle, dénoté par sa fonction caractéristique χ_X, $Acc(X)$ correspond à la fonction $\chi_X \circ \vec{\delta}$ définie de $Q \times E$ dans $\{0,1\}$.

De plus, les opérateurs ensemblistes $\cup$ et $\cap$ se traduisent respectivement par les opérateurs de somme et de produit booléen entre fonctions caractéristiques. On obtient donc une définition de point fixe dans le domaine booléen de l'ensemble des transitions du modèle vérifiant les spécifications:

$$\sigma_v = \nu y.\ [\neg\alpha_v \vee [\sigma \wedge \widetilde{Src}(y) \circ \vec{\delta}]]$$

Ce point fixe s'écrit encore:

$$\sigma_v = \nu y.\ [\alpha_v \Rightarrow [\sigma \wedge \widetilde{Src}(y) \circ \vec{\delta}]]$$

Celui-ci est alors classiquement la limite de la suite de fonctions définie ci-dessous:

$$\begin{cases} y_0 &= \alpha_v \Rightarrow \sigma \\ y_{n+1} &= y_n \wedge [\alpha_v \Rightarrow [\sigma \wedge \widetilde{Src}(y_n) \circ \vec{\delta}]] \end{cases}$$

Le calcul effectif du point fixe revient donc au calcul des termes successifs de cette suite, jusqu'à convergence:

$$\begin{aligned} y_0 &= \alpha_v \Rightarrow \sigma \\ y_1 &= y_0 \wedge [\alpha_v \Rightarrow [\sigma \wedge \widetilde{Src}(y_0) \circ \vec{\delta}]] \\ \vdots &= \vdots \\ y_{k+1} &= y_k \wedge [\alpha_v \Rightarrow [\sigma \wedge \widetilde{Src}(y_k) \circ \vec{\delta}]] \end{aligned}$$

Au rang k, on a $y_{k+1} = y_k$.

Le test de satisfaction des spécifications par le modèle peut être effectué au fur et à mesure du calcul des termes de la suite. En effet, nous avons vu que lorsque les spécifications ne sont pas vérifiées, cela correspond à

$$\exists e \in E, (q_{init}, e) \notin \sigma_v$$

soit, dans le domaine fonctionnel:

$$[Src(\neg\sigma_v)](q_{init})$$

ou encore:

$$\neg[\widetilde{Src}(\sigma_v)](q_{init})$$

Compte tenu de la décroissance monotone de la suite calculée ($\forall i \in N, y_{i+1} \Rightarrow y_i$), il est possible de tester que les spécifications sont satisfaites à chaque calcul d'un nouveau terme.

En tenant compte de toutes ces remarques, on peut maintenant définir un algorithme d'évaluation des spécifications, qui est décrit dans la section suivante.

12.4.2 Algorithme d'évaluation

D'un point de vue global, le calcul du point fixe est implémenté sous la forme d'une boucle de calcul des termes successifs de la suite définie précédemment, dont la condition d'arrêt est la convergence de la suite, et dans laquelle la satisfaction des spécifications est testée à chaque cycle. Les notations adoptées sont les suivantes:

- α_v est la fonction caractéristique des transitions valides.
- σ est la fonction de sortie du modèle (dénotant les spécifications).
- σ_v est la fonction caractéristique des transitions vérifiant les spécifications.
- σ_r est la fonction de référence à laquelle σ_v va être comparée pour déterminer la convergence de la suite.
- σ'_v est la fonction caractéristique des états corrects par rapport aux spécifications.

L'algorithme ci-dessous traduit alors l'évaluation des spécifications dans le domaine fonctionnel booléen. Il est directement implémentable sur les BDDs.

debut
$\sigma_v = (\alpha_v \Rightarrow \sigma)$
$\sigma_r = 1$
tantque $(\sigma_r \neq \sigma_v)$ **faire**
    debut
    $\sigma'_v = \widetilde{Src}(\sigma_v)$

si $\sigma'_v\,(q_{init})$
alors
debut
$\sigma_r = \sigma_v$
$\sigma_v = \sigma_v \wedge (\alpha_v \Rightarrow (\sigma \wedge \sigma'_v \circ \vec{\delta}))$
fin
sinon
retourner (Spécifications Non Satisfaites)
fin
retourner (Spécifications Satisfaites)
fin

Chapitre 13

Implémentation optimisée

Dans le chapitre précédent, certaines carences de l'implémentation "naturelle" des algorithmes de vérification se sont dégagées: ordre inadéquat des variables dans les BDDs, coût exponentiel de certains opérateurs. Dans ce chapitre, plusieurs améliorations sont apportées pour abaisser le coût de la vérification en mémoire et aussi en temps de calcul.

13.1 Ordonnancement topologique des variables

L'importance de l'ordre des variables dans les BDDs a déjà été plusieurs fois évoqué précédemment. On sait que selon cet ordre, la taille des BDDs et le coût de leurs opérateurs peuvent varier dans de grandes proportions. Or le choix d'un "bon" ordre est un problème très difficile. L'étude des différentes méthodes proposées dans ce domaine révèle qu'elles consistent toutes à rechercher un ordre "local", destiné à réduire la représentation *d'une seule* fonction. Or l'utilisation effective des BDDs conduit toujours à représenter et manipuler un grand nombre de fonctions, donc il n'y a aucune raison de privilégier la représentation d'une fonction particulière par rapport aux autres. Cette remarque conduit immédiatement à définir une méthode de choix d'un ordre "global", déterminé à partir de plusieurs fonctions à représenter et non plus à partir d'une seule. Cette idée est mise en œuvre dans cette section, sous la forme du calcul d'un ordre global par combinaison des ordres locaux obtenus pour chacune des fonctions. La méthode nécessite donc l'emploi d'une méthode de choix d'un ordre local. Nous avons à nouveau choisi la méthode d'ordonnancement topologique décrite dans la section 12.1, à laquelle nous apportons tout d'abord des améliorations. Puis nous présentons la méthode de calcul d'un ordre global, qui a été spécifiquement développée pour la vérification "en arrière".

13.1.1 Optimisation de l'ordonnancement local

Dans la méthode d'ordonnancement topologique de la section 12.1, nous avons vu que l'ordre calculé dépend totalement du mécanisme de pondération des variables. Il est donc logique d'espérer obtenir de meilleurs ordres en optimisant cette phase. C'est pourquoi nous proposons ci-dessous deux améliorations à ce mécanisme.

La pondération des variables s'effectue par analyse topologique d'un réseau d'opérateurs booléens dénotant la fonction à représenter. Il est possible dans certains cas de modifier la

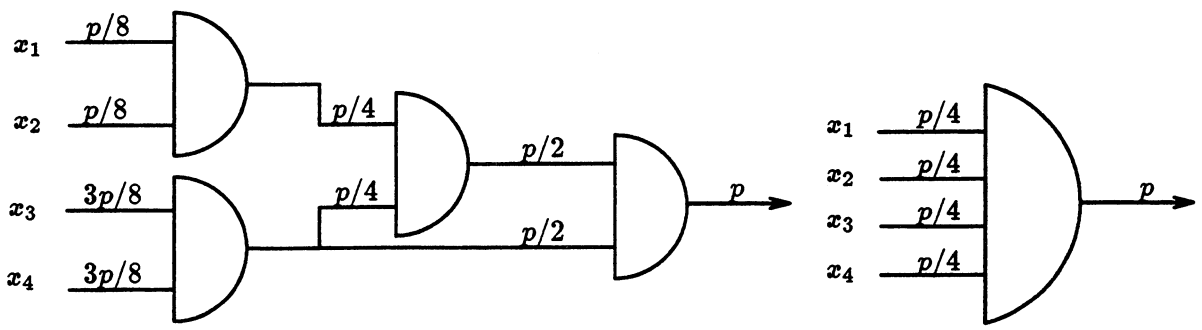


Figure 13.1: Prise en compte de la topologie du réseau

topologie du réseau sans changer la sémantique de la fonction qu'il dénote. La figure 13.1 illustre une telle possibilité. Dans le réseau initial, la pondération est inexacte, car toutes les variables ont en fait la même importance. En modifiant le réseau, on retrouve une pondération équitable. Ainsi, on peut transformer les opérateurs binaires $\wedge$ et $\vee$ en opérateurs n-aires grâce à leurs propriétés d'associativité. Cette opération correspond aux règles de réécriture suivantes:

$$\forall \star \in \{\vee, \wedge\},\ \forall e, e_1, \ldots, e_n,$$

$$\begin{array}{lcl} \star(e, \star(e_0, \ldots, e_n)) & \rightarrow & \star(e, e_0, \ldots, e_n) \\ \star(\star(e_0, \ldots, e_n), e) & \rightarrow & \star(e_0, \ldots, e_n, e) \end{array}$$

D'autre part, en tenant compte des règles liées à la négation booléenne:

$$\begin{array}{lcl} \forall\ e, e_1, \ldots, e_n, & & \\ \neg \vee (e_0, \ldots, e_n) & \rightarrow & \wedge (\neg e_0, \ldots, \neg e_n) \\ \neg \wedge (e_0, \ldots, e_n) & \rightarrow & \vee (\neg e_0, \ldots, \neg e_n) \\ \neg\neg e & \rightarrow & e \end{array}$$

A la suite de ces deux transformations, on obtient des opérateurs $\vee$ et $\wedge$ n-aires auxquels on peut encore appliquer une règle de réduction sur les opérandes:

$$\forall \star \in \{\vee, \wedge\},\ \forall e, e_0, \ldots, e_n$$

$$\star(e, e, e_0, \ldots, e_n) \rightarrow \star(e, e_0, \ldots, e_n)$$

Les optimisations que nous venons de décrire ne concernent que les aspects topologiques du réseau à pondérer. On peut aussi tenir compte de sa sémantique. L'idée consiste alors à se servir des BDDs. En se fixant un ordre arbitraire (par exemple l'ordre naïf déterminé par le compilateur), on peut parcourir le réseau en associant à chaque opérateur le BDD qui lui correspond. Les BDDs ainsi générés traduisent la sémantique de chaque sous-réseau. En particulier, ils fournissent leur domaine de définition exact, c'est-à-dire l'ensemble des variables qui ont une réelle influence sur la sortie du sous-réseau considéré. Connaissant ce domaine, la pondération s'effectuera de façon

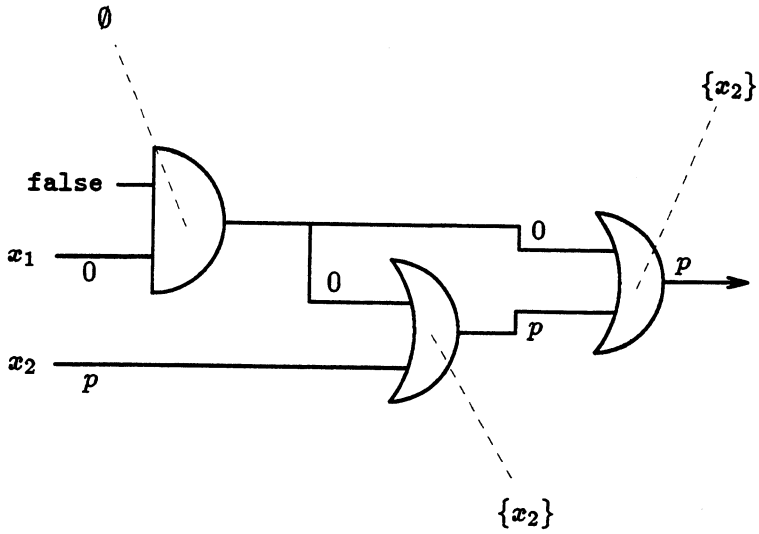


Figure 13.2: Prise en compte de la sémantique du réseau

beaucoup plus fine, car seules les variables de celui-ci seront pondérées. La figure 13.2 illustre cette optimisation.

Enfin, une dernière possibilité d'optimisation de l'ordonnancement topologique consiste à définir une méthode de choix de la variable la plus discriminante parmi plusieurs variables de même poids. En effet, la pondération d'un réseau conduit quelquefois à donner le même poids maximal à plusieurs variables. Il serait alors judicieux d'effectuer un traitement supplémentaire sur ces variables pour déterminer laquelle est la plus discriminante. Toutefois, nous n'avons pas exploité cette idée ici.

13.1.2 Ordonnancement topologique global

La méthode d'ordonnancement "global" que nous présentons maintenant consiste à calculer un nouvel ordre des variables à partir d'une combinaison des ordres locaux de plusieurs fonctions données. L'ordre résultant devant correspondre à une "moyenne" des ordres locaux, il est naturel d'envisager de combiner linéairement ces derniers. Le problème se formalise donc ainsi:
Soit $F = \{f_1, \ldots, f_m\}$ un ensemble de fonctions booléennes sur n variables et $\Pi_F = \{\pi_1, \ldots, \pi_m\}$ l'ensemble des ordres locaux calculés pour ces fonctions. Les π_i sont des permutations sur $\{1, \ldots, n\}$. Soit une combinaison linéaire des π_i:

$$\sum_{i=1}^{n} \rho_i.\pi_i$$

où les ρ_i sont des coefficients réels restant à définir. L'ordre global π sur F se calcule à partir de cette combinaison de la manière suivante:

$$\forall\, j < k,\ \pi(j) < \pi(k) \iff \sum_{i=1}^{n} \rho_i.\pi_i(j) \leq \sum_{i=1}^{n} \rho_i.\pi_i(k)$$

Le poids "global" associé à chaque variable peut s'interpréter comme le barycentre des positions de la variable dans les ordres locaux, affectées des coefficients $\rho_1, \ldots, \rho_n$. L'ordre global est construit en classant les variables selon les valeurs décroissantes de leurs poids globaux.
Les coefficients ρ_i peuvent s'interpréter comme des "coefficients d'importance" de l'ordre de chaque fonction de F pour la détermination de l'ordre global. Pour une fonction et un coefficient donnés, suivant l'importance de ce dernier par rapport aux autres, l'ordre global sera plus ou moins proche de l'ordre local de la fonction, donc il favorisera plus ou moins la compacité de la représentation de la fonction. C'est pourquoi les coefficients ρ_i doivent être proportionnels à l'importance "estimée" de chaque fonction dans la représentation globale de F sous forme de BDD. Selon les critères considérés, on peut attribuer aux ρ_i différentes valeurs, entre autres:

- $\rho_i = 1$ et $\forall j \neq i, \rho_j = 0$. On retombe alors sur un ordre local déterminé à partir de la fonction f_i.
- $\forall\ 1 \leq i \leq n,\ \rho_i = 1$. On suppose alors que toutes les fonctions f_i ont même importance dans la représentation. Le poids global de chaque variable correspond alors à la moyenne de ses positions dans les ordres locaux.
- On attribue à chaque ρ_i une valeur spécifique, proportionnelle à l'importance de chaque fonction f_i dans la représentation. Dans ce cas, il faut que ces valeurs soient réalistes.

Il est donc très simple de combiner des ordres locaux de manière linéaire. Cela n'est cependant pas très astucieux car les ordres traduisent mal l'influence de chaque variable dans les fonctions. En effet, la position d'une variable dans l'ordre n'est pas du tout proportionnelle à son importance sur le résultat de la fonction. Plutôt que de calculer un ordre global directement à partir des ordres locaux, l'idée consiste alors a générer des "pondérations locales" associées à chaque fonction. Ces pondérations doivent préserver l'ordre des variables, mais les poids associés à chacune d'elles seraient alors proportionnels à leur importance sur le résultat de la fonction. Nous avons défini une méthode de calcul d'une pondération "locale" des variables d'une fonction. Celle-ci est basée sur la pondération des variables décrite dans la section 12.1. Nous l'explicitons ci-dessous.

Etant donnée une fonction $f(x_1, \ldots, x_n)$ et π un ordre sur les variables de f calculé par ordonnancement topologique local, on cherche à calculer une pondération "locale" de f, notée Ω. Pour simplifier, on suppose que π est l'identité: $\forall i, \pi(i) = i$. Nous rappelons que cet ordre a été obtenu à la suite de n pondérations successives Ω_i d'un réseau d'opérateurs dénotant f. A l'issue de chaque pondération, une variable considérée comme la plus influente est "déconnectée" du réseau et placée dans l'ordre. Avec les hypothèses que nous avons faites et d'après la manière dont le réseau est pondéré, la i-ème pondération Ω_i fournit un vecteur de poids $[\omega_{i\,1}, \ldots, \omega_{i\,n}]$ où $\omega_{i\,j}$ est le poids de la variable j dans la pondération Ω_i, vérifiant les équations suivantes:

$$\begin{aligned} \sum_{j=1}^{n} \omega_{i\,j} &= 1 \\ \forall\ j,\ (j < i) &\Rightarrow \omega_{i\,j} = 0 \\ \omega_{i\,i} &= \max_{j=1}^{n} \omega_{i\,j} \end{aligned}$$

La deuxième équation provient du fait que chaque variable introduite dans l'ordre est aussitôt déconnectée du réseau, donc son poids est nul dans toute pondération ultérieure.

La troisième équation traduit le fait que la i-ème variable de l'ordre est la variable x_i (car l'ordre est l'identité). Son poids est donc maximal lors de la i-ème pondération.

L'ordonnancement local de $f(x_1,\ldots,x_n)$ produit donc n pondérations du type Ω_i. A partir de ces pondérations, on va calculer $\Omega = [\omega_1,\ldots,\omega_n]$ de telle sorte que Ω préserve l'ordre calculé à partir des Ω_i ainsi que l'importance de chaque variable par rapport *à sa suivante* dans l'ordre. Les Ω_i peuvent être exprimés en extension sous la forme d'une matrice triangulaire supérieure:

$$\begin{bmatrix} \omega_{11} & \omega_{12} & \cdots & \omega_{1n} \\ 0 & \omega_{22} & \cdots & \omega_{2n} \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & \omega_{nn} \end{bmatrix}$$

Considérons Ω_2. Au lieu d'avoir $\omega_{21} = 0$, on va lui attribuer une valeur telle que l'importance de la variable x_1 par rapport à la variable x_2 reste la même que dans Ω_1, ce qui s'exprime par l'équation:

$$\frac{\omega_{11}}{\omega_{12}} = \frac{\omega_{21}}{\omega_{22}}$$

D'où on déduit:

$$\omega_{21} = \omega_{22} \cdot \frac{\omega_{11}}{\omega_{12}}$$

Mais dans ce cas on a alors:

$$\sum_{j=i}^{n} \omega_{2j} = 1 + \omega_{22}.\frac{\omega_{11}}{\omega_{12}}$$

On définit donc une nouvelle pondération $\Omega'_2 = [\omega'_{21},\ldots,\omega'_{2n}]$ telle que:

$$\begin{aligned} \sum_{i=1}^{n} \omega'_{2i} &= 1 \\ \frac{\omega'_{2i}}{\omega'_{2j}} &= \frac{\omega_{2i}}{\omega_{2j}} \end{aligned}$$

Soit encore:

$$\forall\, 1 \leq i \leq n,\ \omega'_{2i} = \frac{\omega_{2i}}{1 + \omega_{22}.\frac{\omega_{11}}{\omega_{12}}}$$

Ce genre de calcul est itéré pour $i < n$ de manière à obtenir à chaque fois Ω'_{i+1} en fonction de Ω'_i et Ω_{i+1}. En fin de compte, la pondération locale Ω est égale à Ω'_n.

Nous définissons ci-dessous le calcul formel de Ω'_{i+1} en fonction de Ω'_i et Ω_{i+1}.
Rappelons les relations existant sur Ω'_i et Ω_{i+1}:

$$\sum_{j=1}^{n} \omega'_{ij} = 1$$

$$\begin{aligned} \sum_{j=1}^{n} \omega_{i+1\,j} &= 1 \\ \forall\, j,\ j \leq i &\Rightarrow \omega_{i+1\,j} = 0 \end{aligned}$$

Dans un premier temps, on calcule pour $j \leq i$ des nouveaux coefficients γ_j remplaçant les $\omega_{i+1\,j}$ (qui sont nuls) dans Ω_{i+1}. Ceux-ci sont tels que:

$$\begin{aligned} \forall\, j > i,\ \gamma_j &= \omega_{i+1\,j} \\ \forall\, j \leq i,\ \frac{\gamma_j}{\gamma_{j+1}} &= \frac{\omega'_{i\,j}}{\omega'_{i\,j+1}} \end{aligned}$$

Pour $j \leq i$, les coefficients γ_j sont définis par une relation de récurrence décroissante sur j:

$$\forall\, 1 \leq j \leq i,\ \gamma_j = \gamma_{j+1} \cdot \frac{\omega'_{i\,j}}{\omega'_{i\,j+1}}$$

Finalement, la pondérations Ω'_{i+1} est définie par:

$$\forall\, 1 \leq j \leq n,\ \omega'_{i+1\,j} = \frac{\gamma_j}{\sum_{k=1}^{n} \gamma_k}$$

Nous disposons donc maintenant d'une méthode de calcul d'une pondération "locale" permettant d'associer aux variables des poids proportionnels à l'importance de celles-ci dans les fonctions. Ces pondérations locales sont plus riches que les ordres au niveau de ce qu'elles expriment, donc on peut obtenir de meilleurs ordres globaux en les combinant.

Nous avons ensuite étendu le calcul de pondération locale aux fonctions obtenues par composition fonctionnelle, du type $h = g(f_1, \ldots, f_n)$. Nous avons vu que du point de vue topologique, la composition fonctionnelle de g par $[f_1, \ldots, f_n]$ consiste en une substitution de chaque occurrence des variables x_i dans g par les fonctions f_i. Par conséquent, on peut faire *l'approximation* consistant à considérer que l'influence de f_i sur h est égale à celle de x_i sur g. Si Ω_g est la pondération locale de g, cette influence est égale à $\omega_{g\,i}$. Par extension, l'influence sur h des variables apparaissant dans f_i est proportionnelle à $\omega_{g\,i}.\Omega_i$, où Ω_i est la pondération locale de f_i. Il en est de même pour toutes les f_i, donc l'influence des variables $x_1, \ldots, x_n$ sur h est proportionnelle à une combinaison linéaire Γ des Ω_i:

$$\Gamma = \sum_{j=1}^{n} \omega_{g\,i}.\Omega_i$$

et pour chaque variable j, le coefficient γ_i correspondant est:

$$\gamma_i = \sum_{i=1}^{n} \omega_{g\,i}.\omega_{i\,j}$$

Définissons alors la matrice $M_{(f_1, \ldots, f_n)}$ des pondérations locales des fonctions $f_1, \ldots, f_n$:

$$M_{(f_1,\ldots,f_n)} = \begin{bmatrix} \omega_{11} & \cdots & \omega_{1j} & \cdots & \omega_{1n} \\ \vdots & & \vdots & & \vdots \\ \omega_{i1} & \cdots & \omega_{ij} & \cdots & \omega_{in} \\ \vdots & & \vdots & & \vdots \\ \omega_{n1} & \cdots & \omega_{nj} & \cdots & \omega_{nn} \end{bmatrix}$$

Les calculs des γ_i correspond alors au calcul matriciel suivant:

$$\Gamma = {}^t M_{(f_1,\ldots,f_n)} . \Omega_g$$

D'où on peut déduire une pondération locale Ω_h pour h en posant:

$$\forall\, 1 \leq i \leq n,\ \omega_{hi} = \frac{\gamma_i}{\sum_{j=1}^{n} \gamma_j}$$

Nous avons donc défini une méthode pour calculer un ordre global des variables à partir d'un ensemble de fonctions donné. Evidemment, une telle approche n'a pas d'intérêt si elle est appliquée à un ensemble quelconque de fonctions. Ainsi, si l'ensemble considéré est grand ou si les fonctions qu'il contient sont complètement indépendantes (c'est le cas par exemple si leurs domaines de définition sont disjoints), ou encore si celles-ci sont connues pour ne pas avoir de représentation sub-exponentielle avec les BDDs, l'ordre global calculé ne sera pas bon. C'est pourquoi la méthode que nous venons de décrire doit être appliquée à un ensemble de fonctions bien défini. Nous montrons dans la partie suivante que c'est le cas dans le cadre de la vérification en arrière.

13.1.3 Application à la vérification en arrière

Le modèle encodant tout programme LUSTRE est défini à partir d'un ensemble de fonctions bien identifiées et caractérisées. Ce sont:

- La fonction d'assertion.
- La fonction de sortie.
- La fonction de transition.

Ces fonctions sont complètement explicitées avant le début de la vérification. On peut donc calculer une pondération locale pour chacune d'elles. Dans un premier temps, un ordre global des variables peut alors être directement obtenu à partir d'une combinaison linéaire de ces pondérations. Appelons $\Omega_\alpha, \Omega_\sigma, \Omega_{\delta_k}$ $(1 \leq k \leq n)$ les pondérations locales correspondant respectivement à la fonction d'assertion, à la fonction de sortie et aux fonctions partielles de la fonction de transition. Il n'y a *a priori* pas de raison de discriminer ces fonctions les unes par rapport aux autres, donc on calcule une pondération globale Ω à partir de la moyenne des pondérations locales:

$$\Omega = \Omega_\sigma + \Omega_\alpha + \sum_{k=1}^{n} \Omega_{\delta_k}$$

On déduit un ordre global à partir de Ω.

On peut maintenant essayer d'ajouter à l'ensemble de fonctions prises en considération dans le calcul d'un ordre global les fonctions issues de compositions fonctionnelles, qui sont calculées au cours de la vérification en arrière. Les expériences réalisées montrent en effet que ces fonctions correspondent en général à de très gros BDDs, d'où l'idée d'essayer de générer un ordre adapté à leur représentation. Initialement, on ne dispose pas de représentation de ces fonctions sous forme de réseaux d'opérateurs, donc on ne peut pas en calculer directement une pondération locale. Par contre, on sait que celles-ci sont générées *au cours* de deux calculs de points fixes, l'un correspondant à l'évaluation des assertions, l'autre à l'évaluation des spécifications.

Pour l'évaluation des assertions, ces fonctions forment une suite $(f_i)_{i\in N}$, définie comme suit:

$$\begin{cases} f_0 &= Src(\alpha) \\ f_{p+1} &= Src(f_p \circ \vec{\delta}) \end{cases}$$

Pour l'évaluation des spécifications, elles forment une suite de fonctions $(g_j)_{j\in N}$, définie par:

$$\begin{cases} g_0 &= \widetilde{Src}(\alpha_v \Rightarrow \sigma) \\ g_{p+1} &= \widetilde{Src}(g_p \circ \vec{\delta}) \end{cases}$$

En faisant abstraction des opérateurs de quantification sur les variables d'entrée, on définit deux nouvelles suites (f'_i) et (g'_i):

$$\begin{cases} f'_0 &= \alpha \\ f'_{p+1} &= f'_p \circ \vec{\delta} \end{cases}$$

$$\begin{cases} g'_0 &= \sigma \\ g'_{p+1} &= g'_p \circ \vec{\delta} \end{cases}$$

Les premiers termes de ces suites sont connus initialement, et leurs pondérations locales respectives sont Ω_α et Ω_σ. Chaque autre terme se déduit du précédent par composition fonctionnelle de ce dernier avec la fonction de transition. On peut donc employer la méthode d'évaluation de pondération locale des fonctions issues de compositions fonctionnelles décrites précédemment. Si on définit la matrice des pondérations locales de la fonction de transition:

$$M_\delta = [\Omega_1, \ldots, \Omega_n]$$

On peut calculer une pondération locale "estimée" des fonctions f'_i et g'_i:

$$\begin{cases} \Omega_{f'_0} &= \Omega_\alpha \\ \Omega_{f'_{p+1}} &= {}^tM_\delta.\Omega_{f'_p} \end{cases}$$

$$\begin{cases} \Omega_{g'_0} &= \Omega_\sigma \\ \Omega_{g'_{p+1}} &= {}^tM_\delta.\Omega_{g'_p} \end{cases}$$

A partir de ces pondérations locales, on peut avoir une idée de l'importance des variables dans les fonctions qui vont être générées. Elles peuvent aussi être utilisées pour le calcul d'un ordre global des variables, à condition de répondre aux deux questions suivantes:

- Combien de préconditions faut-il considérer? Les suites (f_i) et (g_j) étant convergentes, l'ensemble des termes générés est fini. Cependant, on ne sait pas *a priori* à partir de quel rang la convergence est effective. On est donc obligé de considérer un nombre *arbitraire* N de termes.

- Quels poids donner aux pondérations locales des fonctions dans la combinaison linéaire de celles-ci? La réponse à cette question doit tenir compte de deux aspects. D'une part, les pondérations obtenues pour ces fonctions sont issues de deux approximations. La première provient directement de la manière dont sont calculées les pondérations des fonctions obtenues par composition. La seconde est liée au fait que les opérateurs de quantification sont négligés dans l'expression des fonctions effectivement générées au cours des calculs de points fixes. L'autre aspect à prendre en compte dans le choix des poids associés aux pondérations de ces fonctions est lié au fait que la probabilité de générer le n-ième terme des suites considérées diminue au fur et à mesure que n augmente, du fait de la convergence.

Les remarques évoquées ci-dessus font que la prise en compte des fonctions obtenues par composition dans la méthode de vérification en arrière est très délicate.

En conclusion, nous avons défini dans cette section une nouvelle méthode de calcul d'un ordre des variables dans les BDDs. Celle-ci sera expérimentée pour la vérification en arrière.

13.2 Simplification de fonctions

Nous avons vu que les performances des BDDs en mémoire et en temps de calcul dépendent essentiellement de la taille des BDDs manipulés. Or, dés qu'un ordre a été choisi pour les variables, la taille des BDDs devient un paramètre *statique*, qui ne peut plus être modifié en cours d'exécution. La seule manière d'agir sur la taille des BDDs consiste alors à les simplifier de manière *dynamique*, en tenant compte de certaines hypothèses ou informations connues sur le contexte dans lequel ils sont manipulés. Cette approche a donné naissance à une classe d'opérateurs de simplification. Avant de détailler leur fonctionnement, nous rappelons le cadre formel par rapport auquel ceux-ci sont définis.

13.2.1 Simplification de fonctions

Dans ce qui suit, la simplification de fonctions se pose dans les termes suivants: étant donnée une fonction f de $\{0,1\}^n$ dans $\{0,1\}$ et un domaine *non vide* $D \subset \{0,1\}^n$, simplifier f sur D consiste à définir une fonction h équivalente à f sur ce domaine. Cette définition s'exprime par la relation:

$$\forall x,\ x \in D \Rightarrow (h(x) = f(x))$$

qui indique que h est égale à f sur le domaine D, et peut prendre n'importe quelle valeur en dehors. On peut tout de suite remarquer que si le domaine de définition de f et D sont disjoints, aucune simplification de f sur D n'est possible. Par ailleurs, D pouvant être dénoté par sa fonction caractéristique χ_D, la relation précédente s'écrit encore:

$$\forall x,\ \chi_D(x) \Rightarrow (h(x) = f(x))$$

Par définition, on dira que h est une *restriction* de f à χ_D. Réciproquement, étant donnée une fonction $g \neq 0$, celle-ci dénote un unique domaine $D_g \subset \{0,1\}^n$, et la restriction de f à g sera interprétée comme la simplification de f sur le domaine D_g. Dans la suite, on ne parlera plus que de restriction d'une fonction booléenne par une autre fonction booléenne. On peut établir des résultats intéressants et importants au sujet de la restriction de fonctions:

Théorème 4 *Etant données trois fonctions f, $g \neq 0$ et h de $\{0,1\}^n$ dans $\{0,1\}$ h est une restriction de f à g si et seulement si:*

$$(f \wedge g) \Rightarrow h \Rightarrow (\neg g \vee f)$$

Preuve:
$\Rightarrow$: h est une restriction de f à g donc $\forall x,\ g(x) \Rightarrow (h(x) = f(x))$.
Cette relation s'écrit encore:

$$\neg g \vee (h \wedge f) \vee (\neg h \wedge \neg f) = 1$$

ou, après complémentation:

$$g \wedge (h \vee f) \wedge (\neg h \vee \neg f) = 0$$

D'autre part, h vérifie comme toute fonction booléenne la double implication suivante:

$$0 \Rightarrow h \Rightarrow 1$$

On a donc:

$$g \wedge (h \vee f) \wedge (\neg h \vee \neg f) \Rightarrow h \Rightarrow \neg g \vee (h \wedge f) \vee (\neg h \wedge \neg f)$$

d'où on déduit les deux équations ci-dessous:

$$\begin{cases} h \vee \neg g \vee (\neg h \wedge \neg f) \vee (h \wedge f) &= 1 \\ \neg h \vee \neg g \vee (h \wedge f) \vee (\neg h \wedge \neg f) &= 1 \end{cases}$$

Après simplification, ces équations se réécrivent sous la forme:

$$\begin{cases} h \vee \neg f \vee \neg g = 1 \\ \neg h \vee f \vee \neg g = 1 \end{cases}$$

soit finalement

$$\begin{cases} (f \wedge g) \Rightarrow h \\ h \Rightarrow (f \vee \neg g) \end{cases}$$

$\Leftarrow$: Soit h telle que $(\neg g \vee f) \Rightarrow h \Rightarrow (f \wedge g)$. Cette double implication se réécrit sous la forme:

$$(\neg f \vee \neg g \vee h) \wedge (\neg h \vee \neg g \vee f) = 1$$

D'où, après développement et simplification

$$\neg g \vee (h \wedge f) \vee (\neg h \wedge \neg f) = 1$$

et, finalement:

$$g \Rightarrow (h = f)$$

Donc h est une restriction de f à g.

D'après cette double implication, il est clair qu'il existe en général plusieurs solutions au problème de la restriction d'une fonction f par une fonction g.

D'autre part, nous établissons à partir de cette double implication un dernier résultat simple mais très utile sur les restrictions. Etant donné deux fonctions f et $g \neq 0$, et h une restriction de f à g, on a l'égalité:

$$f \wedge g = h \wedge g$$

Preuve:
Immédiate sachant que $f \wedge g \Rightarrow h \Rightarrow \neg g \vee f$.

13.2.2 Opérateurs de restriction sur les BDDs

Appliquée aux BDDs, le but de toute restriction de f à g est de réduire la taille du BDD représentant f sur le domaine caractérisé par g. L'intérêt de cette réduction réside essentiellement dans l'abaissement du coût des opérateurs qui sont sensibles à la taille de leurs opérandes. Il existe plusieurs solutions possibles au problème de la restriction, la meilleure étant celle dont le BDD est de taille minimale. Cependant, la recherche d'une telle solution ne doit pas être d'un coût supérieur au gain qu'elle peut apporter aux autres opérateurs. C'est pourquoi en pratique, on ne cherche pas une restriction optimale, mais juste une "bonne" restriction. Le choix de cette dernière s'effectue selon des critères heuristiques. Plusieurs opérateurs de restriction sur les BDDs ont été conçus, chacun avec des critères spécifiques de choix d'une solution. Ils fournissent donc en général des résultats différents les uns des autres. En ce qui nous concerne, le critère principal est la réduction de la taille des BDDs. Notre choix s'est porté sur celui offrant le meilleur compromis entre le gain de taille et le temps de calcul nécessaire à l'obtention du résultat: il s'agit d'un opérateur appelé "restrict" [Mad90], et noté $\Uparrow$. Cet opérateur fonctionne selon les règles suivantes:

Règles terminales:
La valeur ϕ utilisée ci-dessous désigne *n'importe quelle* fonction.

Il peut est assimilable à une fonction indéterminée.

$f \Uparrow 0 = \phi$
$f \Uparrow 1 = f$
$f \Uparrow f = 1$
$0 \Uparrow g = 0$
$1 \Uparrow g = 1$

Règles d'inférence:
Soit $f = \neg x \wedge f_{\bar{x}} \vee x \wedge f_x$ et $g = \neg y \wedge g_{\bar{y}} \vee y \wedge g_y$
si $x \prec y$, $f \Uparrow g = combiner(x, f_{\bar{x}} \Uparrow g, f_x \Uparrow g)$
si $x = y$, $f \Uparrow g = combiner(x, f_{\bar{x}} \Uparrow g_{\bar{x}}, f_x \Uparrow g_x)$
si $y \prec x$, $f \Uparrow g = f \Uparrow (g_{\bar{x}} \vee g_x)$

La fonction *combiner* fonctionne quant à elle de la façon suivante:

$combiner(x, f, \phi) = f$
$combiner(x, \phi, g) = g$
$combiner(x, f, g) = \neg x \wedge f \vee x \wedge g$

Nous mentionnons deux autres opérateurs de restriction qui présentent un grand intérêt dans certains cas: ce sont le "constrain" [OC89b,OC89a] (noté $\uparrow$) et le "minimize" [Ray91] (noté $\downarrow$):

- L'opérateur "constrain" assure que le résultat obtenu a des propriétés particulières [Cou91, HJT90]. Par rapport à l'opérateur $\Uparrow$, la seule différence se situe au niveau de la dernière règle d'inférence, qui est la suivante:

  si $y \prec x$, $f \uparrow g = combiner(y, f_{\bar{y}} \uparrow g, f_y \uparrow g)$

  Intuitivement, l'opérateur $\uparrow$ autorise l'introduction dans le résultat de variables appartenant à g mais n'appartenant pas à f, alors que l'opérateur $\Uparrow$ interdit cette possibilité.

- L'opérateur "minimize" permet quant à lui de fournir des restrictions de taille en général inférieure aux deux opérateurs précédents, et de réduire le domaine de définition des fonctions restreintes. Il est utilisé pour la génération de code dans le compilateur LUSTRE, afin de diminuer le nombre de tests à effectuer [Ray91]. Malheureusement, cet opérateur présente un coût exponentiel alors que le "restrict" et le "constrain" restent polynômiaux.

Un exemple d'application de ces opérateurs de restriction est donné dans la figure 13.3. Celui-ci a été choisi au hasard et n'entend pas prouver que l'un des opérateurs est meilleur que les autres.

Il faut remarquer que, dans certains cas, les opérateurs de restriction présentés ci-dessus fournissent de mauvais résultats, dans le sens où le BDD obtenu est plus important en taille que le BDD d'origine (voir exemple de la figure 13.4). Dans ce cas, le gain apporté par ces opérateurs est *négatif*, étant donné qu'on a consommé du temps de calcul sans pour autant réduire la complexité du BDD considéré. Heureusement, ces cas surviennent rarement de sorte

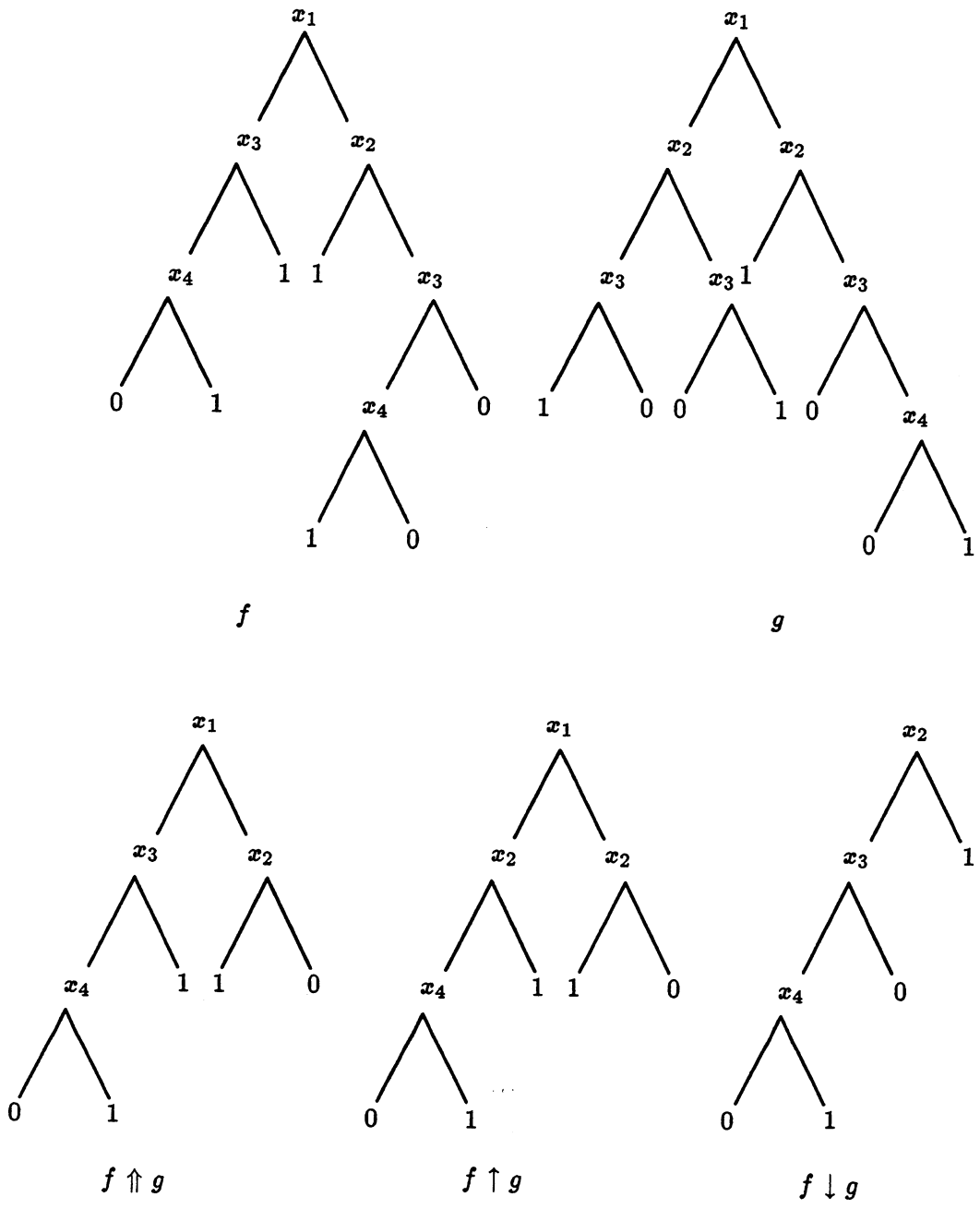


Figure 13.3: Exemple d'application des opérateurs de restriction

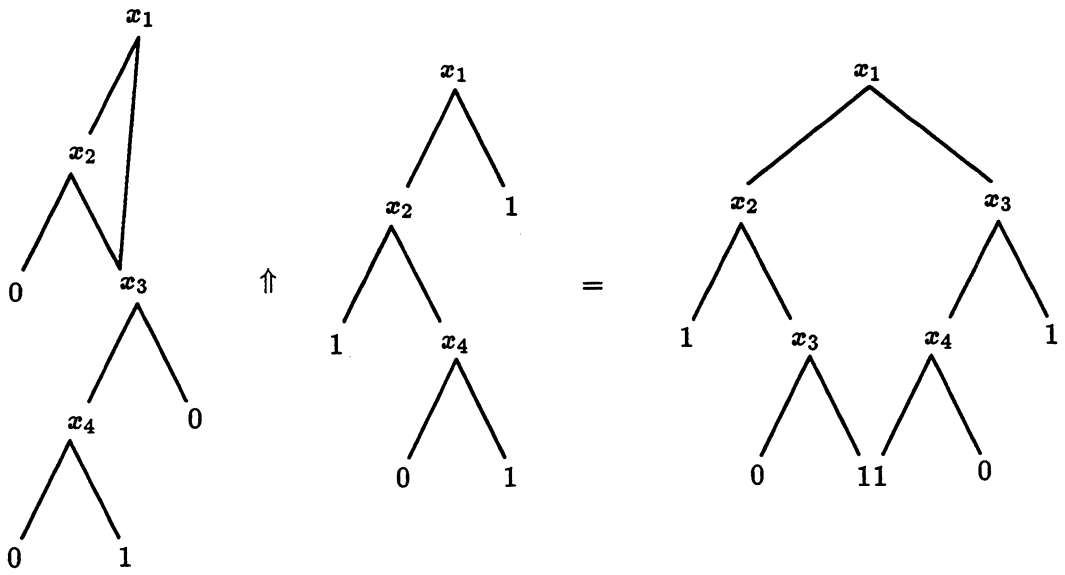


Figure 13.4: Exemple de mauvaise restriction

que globalement, le gain fourni par les opérateurs de restriction est largement positif. D'autre part, on peut aussi se poser la question de savoir à partir de quelle taille de BDD il est intéressant d' appliquer les opérateurs de restriction. En effet, il est logique de croire que sur les "petits" BDDs, le gain est trop faible pour être intéressant.

13.2.3 Application à la vérification en arrière

Les opérateurs de restriction peuvent tout de suite être utilisés pour la simplification du modèle sous les hypothèses fournies par les assertions. En effet, ces dernières s'expriment sous la forme d'une fonction booléenne et toutes les autres fonctions du modèle peuvent être simplifiées sachant que les assertions sont vraies. Ainsi, à partir de tout modèle $\mathcal{M} = (Q, E, S, q_{init}, \alpha, \sigma, \vec{\delta})$, on peut définir un modèle $\mathcal{M}_\alpha = (Q, E, q_{init}, \alpha, \sigma_\alpha, \vec{\delta}_\alpha)$ équivalent à $\mathcal{M}$ sur le domaine où les assertions sont vérifiées. Ce nouveau modèle peut être obtenu de la façon suivante, en utilisant par exemple l'opérateur $\Uparrow$:

$$\sigma_\alpha = \sigma \Uparrow \alpha$$

$$\vec{\delta}_\alpha = [\delta_1 \Uparrow \alpha, \ldots, \delta_n \Uparrow \alpha]$$

Les opérateurs de restriction vont aussi être utilisés dans plusieurs autres cas qui seront détaillées ultérieurement.

13.3 Calcul de plus grand point fixe

Les algorithmes d'évaluation des assertions et des spécifications sont tous les deux basés sur le calcul de plus grands points fixes de fonctions. Nous allons optimiser ce calcul à l'aide des opérateurs de simplification de fonctions.

13.3.1 Optimisation des opérations de composition fonctionnelle

De manière générale, considérons une suite de fonctions booléennes $(g_i)_{i\in N}$ vérifiant les relations:

$$\forall\, i \in N,\ g_i \neq 0$$

$$\forall\, (i,k) \in N^2,\ i < k \Rightarrow (g_k \Rightarrow g_i)$$

Cette suite est monotone décroissante au sens de l'implication. On cherche maintenant à calculer les termes de la suite $(h_i)_{i\in N}$ définie par:

$$\forall\, i \in N,\ h_i = g_i \circ \vec{f}$$

où $\vec{f} = [f_1, \ldots, f_n]$ est un vecteur fonctionnel booléen. Les h_i peuvent être calculées directement à partir de leur définition. Cependant, la composition fonctionnelle est une opération très coûteuse. On va donc utiliser les propriétés de la suite (g_i) et les opérateurs de simplification de fonction vus dans la section précédente pour faire décroître la complexité du calcul de la suite (h_i).

D'après la relation vérifiée par la suite $(g_i)_{i\in N}$, on a en particulier:

$$\forall\, i \in N,\ g_{i+1} \Rightarrow g_i$$

Donc

$$\forall\, i \in N,\ g_{i+1} = g_{i+1} \wedge g_i$$

Nous notons "$\Uparrow$" un opérateur de simplification de fonction quelconque. Pour toute fonction f et toute fonction $g \neq 0$, $f \Uparrow g$ est une restriction de f à g, donc d'après les résultats établis dans la section précédente:

$$g_{i+1} = g_{i+1} \wedge g_i = (g_{i+1} \Uparrow g_i) \wedge g_i$$

Compte tenu de la distributivité de la composition fonctionnelle sur le produit booléen:

$$\forall\, i \in N,\ g_{i+1} \circ \vec{f} = (g_{i+1} \Uparrow g_i) \circ \vec{f} \wedge g_i \circ \vec{f}$$

d'où on déduit une relation de récurrence suivante sur les h_i:

$$\forall\, i \in N,\ h_{i+1} = (g_{i+1} \Uparrow g_i) \circ \vec{f} \wedge h_i$$

La relation ci-dessus est une alternative au calcul direct des h_i. La question qui se pose alors consiste à déterminer quelle implémentation du calcul de ces termes est la plus efficace. Naturellement, nous abordons ici le problème sous l'angle des BDDs.

A priori, la méthode de calcul des h_i par récurrence parait plus coûteuse, car elle nécessite deux opérations supplémentaires pour le calcul de chaque terme: une simplification ($\Uparrow$) et un produit ($\wedge$). En revanche, la composition fonctionnelle effectuée sur $(g_{i+1} \Uparrow g_i)$ va être moins complexe que celle effectuée directement sur g_{i+1}, à cause de la simplification apportée par l'opérateur $\Uparrow$. Si on compare maintenant le coût des opérateurs impliqués dans les calculs, celui-ci est polynômial pour la simplification et le produit booléen, tandis qu'il est exponentiel pour la composition. On a donc largement intérêt à favoriser la baisse du coût de la composition par rapport aux autres opérations. C'est pourquoi la seconde méthode de calcul de la suite (h_i) est plus intéressante que la méthode directe.

13.3.2 Optimisation des opérations de quantification universelle

L'optimisation décrite ci-dessus pour la composition fonctionnelle est valable pour tout autre opérateur distributif sur le produit booléen. Néanmoins, cette optimisation n'est intéressante qu'à condition que l'opérateur considéré ait un coût élevé sur les BDDs, auquel cas le coût des opérations supplémentaires est compensé par le gain sur l'opérateur en question. C'est le cas de la quantification universelle. Etant donnée la suite de fonctions booléennes (g_i) définie précédemment, on définit la suite de fonctions (k_i) de la façon suivante:

$$\forall\, i \in N,\ k_i = \forall X\ g_i$$

où X est un ensemble quelconque de variables du domaine de définition de g_i, et la notation "$\forall X$" dénote la quantification universelle sur les variables de X (voir section 11.2.3). La quantification universelle étant distributive sur le produit booléen,

$$\forall\, i \in N,\ \forall X\ (g_{i+1} \wedge g_i) = (\forall X\ g_{i+1}) \wedge (\forall X\ g_i)$$

d'où on déduit une relation de récurrence suivante sur les k_i:

$$\forall\, i \in N,\ k_{i+1} = [\forall X\ (g_{i+1} \Uparrow g_i)] \wedge k_i$$

13.3.3 Application à la vérification en arrière

Les résultats établis ci-dessus ont une application directe dans le cadre de la vérification en arrière, pour l'évaluation des assertions et des spécifications. En effet, nous avons vu que cette évaluation s'effectue par calcul de plus grand point fixe de fonctions booléennes. En pratique, ce calcul revient à générer les termes successifs des deux suites Λ et Σ définies par:

$$\Lambda : \begin{cases} \alpha_0 &= \alpha \\ \alpha_{n+1} &= \alpha_n \wedge Src(\alpha_n) \circ \vec{\delta} \end{cases}$$

$$\Sigma : \begin{cases} \sigma_0 &= \alpha_v \Rightarrow \sigma \\ \sigma_{n+1} &= \alpha_v \Rightarrow [\sigma_n \wedge \widetilde{Src}(\sigma_n) \circ \vec{\delta}] \end{cases}$$

Il a déjà été démontré que ces deux suites sont convergentes, donc elles vérifient bien:

$$\forall\ (i,k) \in N^2,\ i < k \Rightarrow (\alpha_k \Rightarrow \alpha_i)$$

$$\forall\ (i,k) \in N^2,\ i < k \Rightarrow (\sigma_k \Rightarrow \sigma_i)$$

Cette relation peut être étendue aux termes quantifiés:

$$\forall\ (i,k) \in N^2,\ i < k \Rightarrow (Src(\alpha_k) \Rightarrow Src(\alpha_i))$$

$$\forall\ (i,k) \in N^2,\ i < k \Rightarrow (\widetilde{Src}(\sigma_k) \Rightarrow \widetilde{Src}(\sigma_i))$$

On définit alors les suites $\Lambda' = (\alpha'_n)_{n\in N}$ et $\Sigma' = (\sigma'_n)_{n\in N}$:

$$\Lambda' : \left\{ \begin{array}{rcl} \alpha'_0 & = & 1 \\ \alpha'_{n+1} & = & Src(\alpha_n) \end{array} \right.$$

$$\Sigma' : \left\{ \begin{array}{rcl} \sigma'_0 & = & 1 \\ \sigma'_{n+1} & = & \widetilde{Src}(\sigma_n) \end{array} \right.$$

Comme cela vient d'être établi, ces suites sont convergentes. Intéressons-nous maintenant au calcul des termes des suites $\Lambda'' = (\alpha''_n)_{n\in N}$ et $\Sigma'' = (\sigma''_n)_{n\in N}$ définies ci-dessous:

$$\Lambda'' :\ \forall\ n \in N,\ \alpha''_n = \alpha'_n \circ \vec{\delta} = Src(\alpha'_n) \circ \vec{\delta}$$

$$\Sigma'' :\ \forall\ n \in N,\ \sigma''_n = \sigma'_n \circ \vec{\delta} = \widetilde{Src}(\sigma'_n) \circ \vec{\delta}$$

Tant que les termes des suites Λ' et Σ' ne convergent pas vers 0, on peut alors appliquer la méthode de calcul par récurrence des termes de Λ'' et Σ'':

$$\forall\ n \in N,\ \alpha''_{n+1} = (\alpha'_{n+1} \Uparrow \alpha'_n) \circ \vec{\delta} \wedge \alpha''_n$$

$$\forall\ n \in N,\ \sigma''_{n+1} = (\sigma'_{n+1} \Uparrow \sigma'_n) \circ \vec{\delta} \wedge \sigma''_n$$

On en déduit une nouvelle définition des suites Λ et Σ à partir de Λ'' et Σ'':

$$\Lambda : \left\{ \begin{array}{rcl} \alpha_0 & = & \alpha \\ \alpha_{n+1} & = & \alpha_n \wedge \alpha''_{n+1} \end{array} \right.$$

$$\Sigma : \left\{ \begin{array}{rcl} \sigma_0 & = & \alpha_v \Rightarrow \sigma \\ \sigma_{n+1} & = & \alpha_v \Rightarrow (\sigma_n \wedge \sigma''_{n+1}) \end{array} \right.$$

L'évaluation des assertions et des spécifications peut donc être optimisée en utilisant un opérateur de simplification booléenne de la manière décrite ci-dessus.

13.4 Partitionnement de fonctions

La taille des BDDs étant le problème majeur pour leur représentation et leur manipulation, il est primordial de gérer efficacement l'espace mémoire qu'ils occupent. Or cet espace dépend de deux paramètres: la taille et le nombre de BDDs représentés. L'utilisation effective des BDDs nécessite donc de minimiser ces valeurs. En ce qui concerne la taille, celle-ci ne dépend que de l'ordre fixé au départ pour les variables. Par conséquent, il est impossible de la contrôler dynamiquement. En revanche, le nombre de BDDs représentés peut être *partiellement* contrôlé [HJT90]. En effet, il est possible de connaître à chaque instant l'ensemble des BDDs utilisés, ou encore "accessibles". L'espace mémoire occupé par les BDDs non accessibles est alors récupérable. Dans ce cas, et pour un ordre des variables fixé, l'espace mémoire effectivement occupé par les BDDs n'est alors plus fonction que du *nombre* de BDDs accessibles. Or celui-ci est sujet à de grosses fluctuations au cours du temps, dues essentiellement aux opérations effectuées sur les BDDs. En effet, les opérateurs symboliques génèrent des résultats intermédiaires pendant leur exécution. Bien qu'utilisés temporairement, ces résultats doivent néanmoins être conservés durant toute ou partie des calculs, d'où un accroissement du nombre de BDDs accessibles pendant l'exécution des opérateurs symboliques. Cet accroissement peut devenir considérable dans certains cas, et aboutir à une saturation complète de l'espace mémoire disponible. Les calculs doivent alors être abandonnés, faute d'espace mémoire suffisant.

Evidemment, un tel cas d'échec n'est pas acceptable et doit être absolument évité. Une manière d'y parvenir consiste à ne pas effectuer d'opérations risquant de générer trop de résultats intermédiaires. Or, le coût des opérateurs BDDs est proportionnel à la taille de leurs opérandes. Il en est de même pour le nombre d'opérations intermédiaires effectuées, donc pour le nombre de résultats intermédiaires générés. Par conséquent, il faut éviter d'effectuer des opérations dont les opérandes sont de trop grosse taille. Cela n'est pas toujours possible, mais nous décrivons ici un moyen d'y arriver dans certains cas.

13.4.1 Couverture de fonctions

On appelle couverture *disjonctive* d'une fonction booléenne f tout ensemble fini de fonctions booléennes $\mathcal{C}_\vee(f) = \{f_1, \ldots, f_n\}$ vérifiant:

$$f = \bigvee_{i=1}^{n} f_i$$

$$\forall\, 1 \leq j \leq n,\ (\bigvee_{i=1, i\neq j}^{n} f_i) \neq f$$

Intuitivement, la somme booléenne des fonctions de $\mathcal{C}_\vee(f)$ est égale à f, et tout sous ensemble de $\mathcal{C}_\vee(f)$ n'est pas une couverture de f. D'autre part, $\mathcal{C}_\vee(f)$ est une couverture *exclusive* si elle vérifie en plus la relation suivante:

$$\forall\, 1 \leq i < j \leq n,\ f_i \wedge f_j = 0$$

Toute fonction f admet au moins une couverture disjonctive exclusive (elle-même).
Dualement, on définit la couverture *conjonctive* exclusive $\mathcal{C}_\wedge(f)$ de toute fonction f:

$$f = \bigwedge_{i=1}^{n} f_i$$

$$\forall\, 1 \leq j \leq n,\ (\bigwedge_{i=1, i\neq j}^{n} f_i) \neq f$$

$$\forall\, 1 \leq i < j \leq n,\ f_i \vee f_j = 1$$

Toute fonction f admet au moins une couverture *conjonctive* exclusive. De plus, toute couverture disjonctive exclusive d'une fonction est une couverture conjonctive exclusive de sa négation:

$$\neg f = \bigvee_{i=1}^{n} f_i'$$

D'où:

$$f = \bigwedge_{i=1}^{n} (\neg f_i')$$

En posant $f_i = \neg f_i'$:

$$f = \bigwedge_{i=1}^{n} f_i$$

et:

$$\forall\, 1 \leq i < j \leq n,\ \neg f_i \wedge \neg f_j = 0$$

D'où:

$$f_i \vee f_j = 1$$

Toute fonction booléenne f possède donc au moins une couverture disjonctive et une couverture conjonctive exclusives. Dans la suite, on appellera indistinctement "couverture" l'un ou l'autre de ces types de couvertures.

13.4.2 Intérêt des couvertures de fonctions

L'intérêt des couvertures de fonctions se situe au niveau des opérateurs distributifs sur la somme ou le produit booléen. En effet, pour tout opérateur n-aire $\star$ distributif sur la somme, et pour toute fonction f dont une couverture est $C_\vee(f) = \bigvee_{k=1}^{n} f_k$, on a:

$$\forall\, 1 \leq i \leq n,\ \star(x_1, \ldots, x_{i-1}, f, x_{i+1}, \ldots, x_n) = \bigvee_{k=1}^{n} \star(x_1, \ldots, x_{i-1}, f_k, x_{i+1}, \ldots, x_n)$$

De même, pour tout opérateur $\star$ distributif sur le produit, et pour toute fonction f dont une couverture est $\mathcal{C}_\wedge(f) = \bigwedge_{k=1}^{n} f_k$, on a:

$$\forall\, 1 \leq i \leq n,\ \star(x_1, \ldots, x_{i-1}, f, x_{i+1}, \ldots, x_n) = \bigwedge_{k=1}^{n} \star(x_1, \ldots, x_{i-1}, f_k, x_{i+1}, \ldots, x_n)$$

On peut donc scinder toute opération $\star(x_1, \ldots, f, \ldots, x_n)$ en n opérations $\star(x_1, \ldots, f_k, \ldots, x_n)$ et $n-1$ sommes ou produits booléens. Bien que le nombre d'opérations effectuées soit alors plus élevé que pour l'opération directe, celles-ci sont d'une complexité moins grande. C'est à ce niveau que l'utilisation des couvertures de fonctions peut présenter un intérêt.

13.4.3 Application aux BDDs

Au niveau des BDDs, le problème que nous cherchons à résoudre consiste à éviter d'appliquer les opérateurs symboliques sur des opérandes de trop grande taille. Si les opérateurs considérés sont distributifs sur la somme ou le produit booléen, on peut alors calculer des couvertures de leurs opérandes et utiliser les résultats établis ci-dessus. Ainsi, étant donnée une fonction f et une couverture disjonctive $\mathcal{C}_\vee(f) = \{f_1, \ldots, f_n\}$ de f, toute opération $\star(\ldots, f, \ldots)$ (où $\star$ est un opérateur distributif sur la somme booléenne) peut alors être implémentée par un algorithme du type:

debut
soit $\mathcal{C}_\vee(f) = \{f_1, \ldots, f_n\}$;
$res = 0$;
pour *chaque* $f_i \in \mathcal{C}_\vee(f)$ **faire**
    $res = res \vee \star(\ldots, f_i, \ldots)$;
fin

Il reste à déterminer sous quelles conditions l'algorithme ci-dessus aura un coût moindre que l'opération directe $\star(\ldots, f, \ldots)$ sur les BDDs. A ce niveau, il est clair que ce coût dépend complètement de la couverture de f utilisée:

- En mémoire: l'objectif étant avant tout de réduire le coût en mémoire de l'opération à effectuer en l'appliquant à des opérandes de taille plus raisonnable, il est *obligatoire* que la représentation de chaque fonction f_i de la couverture soit de taille inférieure à celle de f, afin d'abaisser le nombre de résultats intermédiaires générés pendant chaque opération. Cependant, la représentation de $\mathcal{C}_\vee(f)$ va générer un surcoût, puisque chaque fonction de la couverture doit être représentée par un BDD supplémentaire.

- En temps de calcul: le coût total des appels à $\star$ dépendra à la fois de la taille et du nombre de BDDs représentant la couverture. A cela, il faut ajouter le coûts des appels à l'opérateur $\vee$, qui dépend lui aussi de ces deux paramètres.

Finalement, le coût de l'algorithme dépend à la fois de la taille et du nombre de BDDs représentant la couverture. On peut alors envisager les cas extrêmes suivants:

- Les BDDs de la couverture sont de taille "très petite". Le coût de chaque appel à $\star$ sera alors diminué à la fois en temps et en mémoire. Par contre, le nombre de fonctions

nécessaires pour former la couverture va être "très élevé". Donc, la taille globale de la couverture et le nombre d'opérations à effectuer vont augmenter d'autant.

- La couverture n'est formée que par un seul BDD, qui est donc nécessairement celui de f. Le coût de l'algorithme est donc égal à celui de l'opération directe.

En pratique, on n'a pas intérêt à ce que les BDDs de la couverture soient trop petits, car ils deviennent alors très nombreux, ce qui produit une augmentation du coût de leur représentation en mémoire et du nombre d'opérations effectuées. Il faut donc trouver un compromis entre la taille et le nombre de BDDs formant la couverture, ces deux paramètres étant évidemment liés.

A partir des remarques ci-dessus, nous avons défini un opérateur de "partitionnement" sur les BDDs. Cet opérateur calcule, pour un BDD donné, une couverture répondant aux critères évoqués ci-dessus. Naturellement, il existe en général un très grand nombre de couvertures d'un même BDD. Le choix de l'une d'entre elles ne peut être qu'heuristique. La méthode de partitionnement que nous décrivons ci-dessous est basée sur des critères syntaxiques: elle s'appuie en effet sur une analyse topologique du BDD à couvrir, c'est-à-dire que la couverture générée dépend uniquement de la structure du BDD partitionné.
L'heuristique mise en œuvre consiste simplement à générer une couverture d'un BDD dont tous les éléments sont de taille inférieure à une limite *constante*. Nous décrivons ci-dessous l'algorithme générant une couverture disjonctive exclusive d'un BDD f sous forme d'un ensemble de BDD de taille inférieure à une limite $M \in N^*$.

fonction *partitioner*(f): $\mathcal{C}_\vee(f)$
debut
si $|f| \leq M$
alors
    $\mathcal{C}_\vee(f) = \{f\}$;
sinon
    debut
    choisir f', p, q *telles que* $|f'| < M$ *et* $f = p \wedge f' \vee q \wedge \neg f'$;
    $\mathcal{C}_\vee(f)$ = *partitioner*$(p \wedge f') \cup$ *partitioner*$(q \wedge \neg f')$;
    fin
fin

Dans cet algorithme, le partitionnement de f dépend exclusivement du choix de la fonction f'. Il est facile de prouver l'existence d'une telle fonction. L'existence d'un sous-BDD f' de f vérifiant $|f'| \leq M$ est évidente. En effet, si $f = \neg x \wedge f_{\bar{x}} \vee x \wedge f_x$, on a au niveau de la taille des arbres:

$$|f_x| < |f|$$
$$|f_{\bar{x}}| < |f|$$

La taille des sous-arbres de f étant strictement inférieure à celle de l'arbre de f, il existe nécessairement un sous-arbre de f de taille inférieure à M. Soit f' la fonction dénotée par un de ces sous-arbres. D'après les propriétés de l'expansion de Shannon, il existe un couple (unique) de fonctions (p, q) tel que $f = p \wedge f' \vee q \wedge \neg f'$. Les fonctions $(p \wedge f')$ et $(q \wedge \neg f')$ forment trivialement une couverture disjonctive de f. De plus, cette couverture est exclusive:

$$(p \wedge f') \wedge (q \wedge \neg f') = 0$$

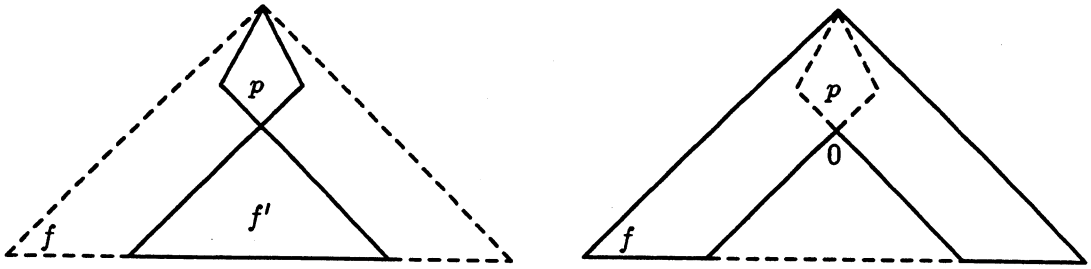
L'existence de f' étant prouvée, il faut d'abord trouver un sous-arbre f' de f vérifiant $|f'| < M$, puis calculer le couple de fonctions $(p \wedge f', q \wedge \neg f')$ sur lequel le partitionnement doit être itéré. La recherche de f' s'effectue par un parcours de f:

fonction *chercher*$(f,\ M)$: f'
debut
si $(f = 0)$ **ou** $(f = 1)$
alors
    $f' = \perp$
sinon
si $(|f| \leq M)$
alors
    $f' = f$
sinon
    debut
    soit $f = \neg x \wedge f_{\bar{x}} \vee x \wedge f_x$
    $f' =$ *chercher*$(f_{\bar{x}}, M-1)$
    si $(f' = \perp)$
    alors
        $f' =$ *chercher*$(f_x, M-1)$
    fin
fin

La taille limite M est passée en paramètre de la fonction *chercher* afin de prendre en compte le nombre m de nœuds visités durant le parcours de f. En effet, tout chemin menant de la racine de f à celle de f' va être inclus dans la fonction p associée à f'. Donc la taille du BDD représentant p *peut* (ce n'est pas forcé compte tenu des réductions) être supérieure à m, auquel cas il faut que la taille de f' soit inférieure à $M - m$. D'autre part, il faut remarquer que M ne peut être inférieure au nombre de variables apparaissant dans les BDDs. Si elle est choisie égale à ce nombre, tout BDD sera décomposé en somme (ou produit) de *monômes*. La fonction *chercher* choisit f' de telle sorte que ce ne soit pas une fonction constante, car autrement l'expansion de Shannon par rapport à une telle fonction n'a aucun intérêt pour le partitionnement.

La recherche du couple (p, q) associé à f' et tel que $f = p \wedge f' \vee q \wedge \neg f'$ s'obtient très simplement par parcours du BDD de f. En effet, p est le BDD issu de la réunion de tous les chemins menant à f' dans f, et q est le BDD issu de la réunion des autres chemins, comme indiqué dans la figure 13.5. L'algorithme de calcul de (p, q) sur les BDDs est le suivant:

fonction *partager*(f, f'): (p, q)
debut
soit $f' = \neg x \wedge f'_{\bar{x}} \vee x \wedge f'_x$
soit $f = \neg y \wedge f_{\bar{y}} \vee y \wedge f_y$
si $(f = f')$
alors
    $(p, q) = (1, 0)$



La fonction $p \wedge f'$ La fonction $q \wedge \neg f'$

Figure 13.5: Partitionnement d'un BDD

sinon
si $(x \prec y)$
alors
$(p, q) = (0, f)$
sinon
debut
$(p_0, q_0) = partager(f_{\bar{y}}, f')$
$(p_1, q_1) = partager(f_y, f')$
$(p, q) = (\neg y \wedge p_0 \vee y \wedge p_1, \neg y \wedge q_0 \vee y \wedge q_1)$
fin
fin

La méthode de partitionnement que nous venons de décrire permet de transformer n'importe quel BDD de taille supérieure à M en une couverture disjonctive exclusive de BDDs tous de taille inférieure à M. Cette opération est coûteuse en mémoire puisque tous les BDDs de la couverture doivent être représentés, et aussi en temps de calcul. En revanche, l'utilisation du partitionnement permet d'abaisser les coûts liés à certaines opérations, à condition que la valeur de M soit bien choisie.

Enfin, remarquons qu'il est possible de définir de très nombreuses heuristiques de génération de couvertures à partir des BDDs. Par exemple, dans la méthode qui vient d'être présentée, la valeur de la limite M pourrait être choisie en fonction de la taille du BDD à partitionner ou de l'opérateur pour lequel une couverture des opérandes doit être calculée. On peut aussi définir d'autres critères que la taille de chaque BDD constituant la couverture. Ainsi, on peut essayer de trouver des méthodes réduisant la taille globale de la couverture générée, ou des méthodes basées sur la sémantique des BDDs.

13.5 Composition fonctionnelle restreinte

L'opérateur de composition fonctionnelle sur les BDDs est de loin le plus coûteux parmi ceux utilisés pour implémenter les algorithmes d'évaluation des assertions et des spécifications. Nous

définissons ici un nouvel opérateur, dérivé de celui décrit dans la section 12.2, dont le but est d'abaisser le coût de la composition fonctionnelle.

13.5.1 Principe

Cet opérateur est basé sur l'utilisation massive des opérateurs de restriction booléenne, d'où son appellation de composition fonctionnelle *restreinte*. Les opérateurs de simplification booléenne s'appliquent à des fonctions dont on limite le domaine de définition. Or si on reprend la définition inductive de l'opérateur de composition fonctionnelle classique, étant donné un vecteur fonctionnel $\vec{f} = [f_1, \ldots, f_n]$, le calcul de $g \circ \vec{f} = g(f_1, \ldots, f_n)$ s'effectue en appliquant la règle suivante:

$$\begin{array}{lcccl} \forall\, 1 \leq i \leq n,\ g \circ \vec{f} & = & \neg f_i & \wedge & g_{\bar{x}_i}(f_1, \ldots, f_{i-1}, f_{i+1}, \ldots, f_n) \\ & \vee & f_i & \wedge & g_{x_i}(f_1, \ldots, f_{i-1}, f_{i+1}, \ldots, f_n) \end{array}$$

Nous allons maintenant montrer que pour toute fonction h, on a:

$$h \wedge (g \circ \vec{f}) = h \wedge g(f_1 \Uparrow h, \ldots, f_i \Uparrow h, \ldots, f_n \Uparrow h)$$

Pour $h = 0$, l'égalité est évidente. Supposons maintenant que $h \neq 0$. On peut alors écrire:

$$\begin{array}{lcccccl} \forall\, 1 \leq i \leq n,\ h \wedge (g \circ \vec{f}) & = & h & \wedge & \neg f_i & \wedge & g_{\bar{x}_i}(f_1, \ldots, f_{i-1}, f_{i+1}, \ldots, f_n) \\ & \vee & h & \wedge & f_i & \wedge & g_{x_i}(f_1, \ldots, f_{i-1}, f_{i+1}, \ldots, f_n) \end{array}$$

Or d'après les résultats établis sur la restriction de fonctions, et comme $h \neq 0$, on sait que:

$$\begin{array}{rcl} h \wedge f_i & = & h \wedge (f_i \Uparrow h) \\ h \wedge \neg f_i & = & h \wedge ((\neg f_i) \Uparrow h) \end{array}$$

De plus:

$$h \wedge \neg f_i = h \wedge \neg (f_i \Uparrow h)$$

En effet, d'après la double implication sur la restriction:

$$(h \wedge f_i) \Rightarrow (f_i \Uparrow h) \Rightarrow (\neg h \vee f_i)$$

d'où, en complémentant:

$$(\neg f_i \wedge h) \Rightarrow \neg (f_i \Uparrow h) \Rightarrow (\neg h \vee \neg f_i)$$

et en faisant le produit avec h:

$$(h \wedge \neg f_i) \Rightarrow [h \wedge \neg (f_i \Uparrow h)] \Rightarrow (h \wedge \neg f_i)$$

Par conséquent:

$$\forall\, 1 \leq i \leq n,\ h \wedge g(f_1, \ldots, f_i, \ldots, f_n) = h \wedge g(f_1, \ldots, f_i \Uparrow h, \ldots, f_n)$$

d'où:

$$h \wedge g \circ \vec{f} = h \wedge g\,(f_1 \Uparrow h, \ldots, f_i \Uparrow h, \ldots, f_n \Uparrow h)$$

Ce résultat peut être appliqué directement à la définition inductive de la composition fonctionnelle, d'où une nouvelle définition de celle-ci:

$$\begin{array}{lrcl} \forall\, 1 \leq i \leq n,\ g \circ \vec{f} & = & \neg f_i \ \wedge & g_{\bar{x}_i}\,(f_1 \Uparrow \neg f_i, \ldots, f_{i-1} \Uparrow \neg f_i, f_{i+1} \Uparrow \neg f_i, \ldots, f_n \Uparrow \neg f_i) \\ & \vee & f_i \ \wedge & g_{x_i}\,(f_1 \Uparrow f_i, \ldots, f_{i-1} \Uparrow f_i, f_{i+1} \Uparrow f_i, \ldots, f_n \Uparrow f_i) \end{array}$$

De la même façon que pour la composition fonctionnelle classique, la règle ci-dessus finira par être appliquée aux fonctions constantes 0 et 1, pour lesquelles on a:

$$\forall\, \vec{f},\ g = 0 \ \Rightarrow\ g \circ \vec{f} = 0$$
$$\forall\, \vec{f},\ g = 1 \ \Rightarrow\ g \circ \vec{f} = 1$$

L'algorithme implémentant la composition restreinte sur les BDDs est alors défini de la manière suivante:

Règles terminales:

$$\mathit{compose\_restrict}(0, \vec{f}) = 0$$
$$\mathit{compose\_restrict}(1, \vec{f}) = 1$$

Règles d'inférence:

soit $g = \neg x_i \wedge g_{\bar{x}_i} \vee x_i \wedge g_{x_i}$
soit $\vec{f} = [f_1, \ldots, f_n]$
$\vec{h} = [f_1 \Uparrow f_i, \ldots, f_{i-1} \Uparrow f_i, f_{i+1} \Uparrow f_i, \ldots, f_n \Uparrow f_i]$
$\vec{l} = [f_1 \Uparrow \neg f_i, \ldots, f_{i-1} \Uparrow \neg f_i, f_{i+1} \Uparrow \neg f_i, \ldots, f_n \Uparrow \neg f_i]$
$\mathit{compose\_restrict}(g, \vec{f}) = \mathit{ite}\,(f_i, \mathit{compose\_restrict}\,(g_{x_i}, \vec{h}), \mathit{compose\_restrict}\,(g_{\bar{x}_i}, \vec{l}))$

L'implémentation sur les BDDs de cet algorithme a pour effet de diminuer la complexité des BDDs représentant les f_i au fur et à mesure du parcours de g, puisque l'opérateur de restriction va en réduire la taille. Par conséquent, le coût de chaque appel à l'opérateur *ite* va être diminué d'autant, d'où un gain sur la complexité globale de l'opérateur de composition. Ce gain est loin d'être négligeable puisqu'on sait que la composition est exponentielle sur les BDDs. Cependant, il est tout d'abord compensé par le coût de toutes les opérations de restriction effectuées à chaque pas de l'algorithme. De plus, ce nouvel opérateur présente un autre inconvénient majeur: contrairement à l'opérateur classique qui peut être rendu unaire, celui-ci est n-aire, car le vecteur fonctionnel passé en paramètre change à chaque appel. La figure 13.6 illustre ce problème. Le nœud x_3 est un nœud partagé dans le BDD auquel est appliqué la composition. Mais en fonction du "chemin" suivi pour parvenir à ce nœud, la fonction f_3 va être restreinte soit par f_1 puis $\neg f_2$, soit par $\neg f_1$ puis f_2, ce qui ne donne pas le même résultat. Du même coup, il ne peut plus y avoir partage des résultats intermédiaires générés au niveau du nœud, puisqu'ils sont différents. L'algorithme ne peut donc plus être linéarisé comme pour la composition fonctionnelle classique. On pourrait utiliser un cache de résultats intermédiaires pour réduire le nombre d'opérations effectuées, mais le taux de réutilisation des résultats du cache risquerait d'être très faible. En

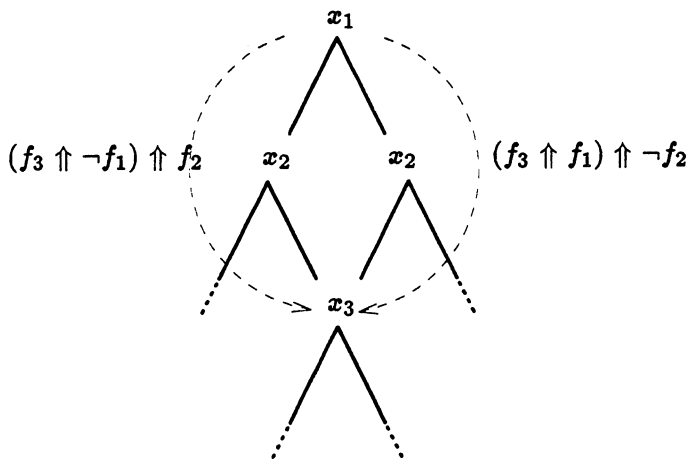


Figure 13.6: Variation des paramètres dans la composition restreinte

effet, nous avons déjà souligné que la probabilité de réutiliser une opération dépendant de n paramètres est d'autant plus faible que n est grand. Par conséquent, l'algorithme de composition fonctionnelle restreinte est *exponentiel*. Un tel résultat rend son utilisation *a priori* complètement rédhibitoire. Nous allons voir que ce n'est pas forcément le cas.

13.5.2 Application à la vérification

Nous allons montrer que dans le contexte de la vérification, l'utilisation de l'algorithme de composition fonctionnelle restreinte est réaliste. Il est clair que cet algorithme ne va être utilisé qu'à la seule condition que son coût soit moins élevé que celui de l'algorithme classique. Les arguments développés dans cette section mettent en évidence les raisons pour lesquelles le nouvel algorithme peut s'avérer efficace dans le contexte où nous l'utilisons.

Tout d'abord, il faut remarquer que dans le cadre de la vérification en arrière, toutes les compositions fonctionnelles effectuées sont du type $\chi \circ \vec{\delta}$, où χ est la fonction caractéristique d'un ensemble d'états quelconque du modèle et où $\vec{\delta}$ est la fonction de transition du modèle. Cette singularité va être exploitée intensivement pour la mise en œuvre de la composition fonctionnelle.

A l'origine, l'algorithme de composition fonctionnelle restreinte est conçu pour diminuer potentiellement la complexité du vecteur fonctionnel par rapport auquel s'effectue la composition, par restriction des fonctions composant ce vecteur entre elles. L'intérêt de l'algorithme est d'abaisser le coût de la composition, qui est exponentiel, par utilisation d'un opérateur de restriction, dont le coût est polynômial. Le gain apporté par l'algorithme dépend donc essentiellement de l'efficacité des opérations de restriction. Or nous avons vu qu'une condition indispensable pour qu'une restriction soit efficace est que les fonctions mises en jeu soient définies sur des domaines similaires. Dans le cadre de la vérification en arrière, le vecteur fonctionnel de *toutes* les compositions effectuées est la fonction de transition du modèle. Les fonctions partielles qui le composent étant définies sur le même domaine $Q \times E$, il est très probable que la restriction de ces fonctions entre elles soit efficace.

D'autre part, on peut remarquer que dans le calcul de $g(f_1, \ldots, f_n)$ par l'algorithme de composition fonctionnelle restreinte, toutes les restrictions effectuées entre les f_i sont *indépendantes* de g. Par conséquent, dans chaque opération de composition restreinte, les restrictions calculées entre les δ_i sont les mêmes. D'où l'idée de mémoriser ces restrictions. En pratique, cela revient à associer à l'opérateur de composition fonctionnelle restreinte un cache des restrictions calculées entre les fonctions δ_i.

Le coût du calcul de $g(f_1, \ldots, f_n)$ par la composition fonctionnelle restreinte peut aussi être abaissé en tenant compte du domaine de définition de la fonction g. En effet, les fonctions de $\vec{f}$ doivent se substituer aux variables de g. Pour les variables qui n'apparaissent pas dans g, les fonctions de $\vec{f}$ correspondantes ne vont pas être utilisées, donc il ne sert à rien de les simplifier. Par conséquent, on peut réduire le nombre de restrictions effectuées sur les fonctions composant $\vec{f}$ en calculant à chaque fois le domaine de définition des fonctions sur lesquelles la composition fonctionnelle restreinte va être effectuée.

Enfin, on a vu que le calcul de $g(f_1, \ldots, f_n)$ par la composition fonctionnelle restreinte sur les BDDs nécessite en théorie un parcours exponentiel du BDD de g. Cependant, les restrictions successives des fonctions de $\vec{f}$ aboutissent dans certains cas aux fonctions constantes 0 ou 1. Dans ce cas, la règle d'inférence de l'opérateur *compose_restrict* défini précédemment se simplifie. Nous rappelons tout d'abord cette règle:

soit $g = \neg x_i \wedge g_{\bar{x}_i} \vee x_i \wedge g_{x_i}$
soit $\vec{f} = [f_1, \ldots, f_n]$
$\vec{h} = [f_1 \Uparrow f_i, \ldots, f_{i-1} \Uparrow f_i, f_{i+1} \Uparrow f_i, \ldots, f_n \Uparrow f_i]$
$\vec{l} = [f_1 \Uparrow \neg f_i, \ldots, f_{i-1} \Uparrow \neg f_i, f_{i+1} \Uparrow \neg f_i, \ldots, f_n \Uparrow \neg f_i]$
$compose\_restrict(g, \vec{f}) = ite(f_i, compose\_restrict(g_{x_i}, \vec{h}), compose\_restrict(g_{\bar{x}_i}, \vec{l}))$

Si $f_i = 0$, la simplification est la suivante:

$$compose\_restrict(g, \vec{f}) = compose\_restrict(g_{\bar{x}_i}, \vec{l})$$

et comme $\neg f_i = 1$, $\vec{l} = \vec{f}$ donc finalement:

si $f_i = 0$, $compose\_restrict(g, \vec{f}) = compose\_restrict(g_{\bar{x}_i}, \vec{f})$

De même, si $f_i = 1$:

si $f_i = 1$, $compose\_restrict(g, \vec{f}) = compose\_restrict(g_{x_i}, \vec{f})$

Ces simplifications occasionnent une réduction du parcours du BDD de g en évitant le parcours de g_{x_i} lorsque $f_i = 0$ et celui de $g_{\bar{x}_i}$ lorsque $f_i = 1$.

Finalement, en tenant compte de toutes ces remarques, on peut définir un nouvel algorithme de composition fonctionnelle restreinte spécifiquement étudié pour les compositions fonctionnelles effectuées dans la méthode de vérification en arrière:

Règles terminales:

$$compose\_restrict(0, \vec{\delta}) = 0$$

$compose\_restrict(1, \vec{\delta}) = 1$

Règles d'inférence:

soit $\chi = \neg x_i \wedge \chi_{\bar{x}_i} \vee x_i \wedge \chi_{x_i}$
si $\delta_i = 0$ **alors** $compose\_restrict(\chi, \vec{\delta}) = compose\_restrict(\chi_{\bar{x}_i}, \vec{\delta})$
si $\delta_i = 1$ **alors** $compose\_restrict(\chi, \vec{\delta}) = compose\_restrict(\chi_{x_i}, \vec{\delta})$
soit $\vec{\delta'}$ et $\vec{\delta''}$ tels que $\forall\ k\ /\ i < k \leq n$
  si $x_k \in domaine(\chi_{x_i})$ **alors** $\delta'_k = restrict\_cache(\delta_k \Uparrow \delta_i)$
  si $x_k \in domaine(\chi_{\bar{x}_i})$ **alors** $\delta''_k = restrict\_cache(\delta_k \Uparrow \neg\delta_i)$
$compose\_restrict(\chi, \vec{\delta}) = ite(\delta_i, compose\_restrict(\chi_{x_i}, \vec{\delta'}), compose\_restrict(\chi_{\bar{x}_i}, \vec{\delta''}))$

La fonction *domaine* calcule le domaine de définition d'un BDD, c'est-à-dire l'ensemble des variables qui le constituent. Ce calcul n'est pas coûteux, puisqu'il est effectué par un parcours linéaire du BDD considéré. Sa définition est la suivante:

Règles terminales:

$domaine(0) = \emptyset$
$domaine(1) = \emptyset$

Règle d'inférence:

soit $f = \neg x \wedge f_{\bar{x}} \vee x \wedge f_x$
$domaine(f) = \{x\} \cup domaine(f_{\bar{x}}) \cup domaine(f_x)$

La fonction *restrict_cache* gère quant à elle les restrictions effectuées sur les fonctions δ_i au cours des compositions restreintes successives.

En conclusion, le nouvel algorithme que nous avons présenté tente d'abaisser la complexité de la composition fonctionnelle, mais le gain qu'il apporte est compensé par le fait qu'il n'est pas linéaire. Cependant, sa mise en œuvre pratique dans le cadre spécifique de la vérification en arrière telle qu'elle a été décrite ci-dessus s'est avérée plus efficace que l'algorithme classique pour une très large majorité des exemples traités. Les performances des deux algorithmes seront mesurées et comparées ultérieurement.

Il reste évidemment des cas où le nouvel algorithme est plus coûteux que l'algorithme classique. C'est le cas en particulier lorsque les restrictions effectuées n'apportent pas ou peu de simplifications sur les fonctions servant à la composition. Le coût de l'algorithme tend alors à être exponentiel. Pour éviter ce genre de problème, on peut envisager de combiner l'utilisation des deux algorithmes. Ainsi l'algorithme de composition restreinte pourrait être utilisé dans le calcul de $g \circ \vec{f}$ tant que le parcours de g ne dépasse pas un certain facteur "d'expansion" par rapport à la taille de g. Au delà de ce facteur, l'algorithme classique serait utilisé pour terminer le calcul.

13.6 Evaluation des assertions

Les optimisations décrites dans les sections précédentes peuvent maintenant être réunies pour l'écriture d'algorithmes optimisés d'évaluation des assertions et des spécifications. Toutefois, certaines d'entre elles restent invisibles à ce niveau. Ainsi, l'ordonnancement des variables et

l'opérateur de composition fonctionnelle utilisé restent anonymes dans les algorithmes décrits ci-après. Néanmoins, on pourra juger de ces optimisations de façon pratique en étudiant les résultats des expérimentations réalisées.

Quoi qu'il en soit, l'intégration des optimisations liées au partitionnement et au calcul de point fixe produit un large remaniement des algorithmes d'évaluation classique. Les résultats de cette prise en compte sont des algorithmes optimisés qui sont décrits ci-dessous. Dans cette section, nous traitons l'évaluation des assertions.

L'évaluation des assertions correspond au calcul d'une fonction booléenne α_v définie par l'équation de point fixe suivante:

$$\alpha_v = \nu y.\ [\alpha \wedge Src(y) \circ \vec{\delta}]$$

Or si $\alpha = \bigwedge_{i=1}^{n} \alpha_i$, on a:

$$\alpha_v = \nu y.\ [(\bigwedge_{i=1}^{n} \alpha_i) \wedge Src(y) \circ \vec{\delta}]$$

Les propriétés de points fixes permettent finalement d'écrire:

$$\alpha_v = \bigwedge_{i=1}^{n} (\nu y.\ [\alpha_i \wedge Src(y) \circ \vec{\delta}])$$

Donc, il est possible de calculer α_v à partir d'une couverture conjonctive de α. Les couvertures de fonctions peuvent aussi être utilisées pour la composition fonctionnelle. Cette opération possède des propriétés de distributivité sur la somme et le produit booléen:

$$\forall\ g_1, \ldots, g_p \in (\{0,1\}^n \mapsto \{0,1\}),\ \forall\ \vec{f} \in (\{0,1\}^m \mapsto \{0,1\})^n,$$

$$(\bigvee_{i=1}^{p} g_i) \circ \vec{f} = \bigvee_{i=1}^{p} (g_i \circ \vec{f})$$

et

$$(\bigwedge_{i=1}^{p} g_i) \circ \vec{f} = \bigwedge_{i=1}^{p} (g_i \circ \vec{f})$$

D'autre part, l'évaluation des assertions correspondant à un calcul de plus grand point fixe, on peut mettre en œuvre les optimisations proposées dans la section 13.3.3. La combinaison des optimisations ci-dessus donne un algorithme optimisé. Pour le décrire, nous employons les mêmes notations que celles de l'algorithme standard. Toutefois, certaines variables supplémentaires sont définies:

- α correspond à la fonction d'assertion du modèle.
- α_v est la fonction caractéristique des transitions valides.

- α_c et α_p correspondent respectivement au terme courant (d'où α_c) et au terme précédent (d'où α_p) de la suite Λ définie dans la section 13.3.3.

- α'_c et α'_p correspondent respectivement au terme courant et au terme précédent de la suite Λ' définie dans la section 13.3.3.

- α''_c correspondent au terme courant de la suite Λ'' définie dans la section 13.3.3.

L'algorithme ci-dessous traduit alors l'évaluation des assertions dans le domaine fonctionnel. Il est directement implémentable sur les BDDs.

debut
$\alpha_v = 1$
soit $\mathcal{C}_\wedge(\alpha) = \{\alpha_1, \ldots, \alpha_m\}$
pour *chaque* $\alpha_i \in \mathcal{C}_\wedge(\alpha)$ **faire**
  debut
  $\alpha_c = \alpha_i \Uparrow \alpha_v$
  $\alpha_p = 1$
  $\alpha'_c = 1$
  tantque $(\alpha_p \neq \alpha_c)$ **faire**
    debut
    $\alpha'_p = \alpha'_c$
    $\alpha'_c = Src(\alpha_c)$
    si $\alpha'_c(q_{init})$
    alors
      debut
      $\alpha''_c = 1$
      soit $\mathcal{C}_\wedge(\alpha'_c \Uparrow \alpha'_p) = \{\alpha'_1, \ldots, \alpha'_n\}$
      pour *chaque* $\alpha'_j \in \mathcal{C}_\wedge(\alpha'_c \Uparrow \alpha'_p)$ **faire**
        $\alpha''_c = \alpha''_c \wedge \alpha'_j \circ \vec{\delta}$
      $\alpha_p = \alpha_c$
      $\alpha_c = \alpha_c \wedge \alpha''_c$
      fin
    sinon
      retourner (Assertions Non Satisfaisables)
    fin
  $\alpha_v = \alpha_v \wedge \alpha_c$
  fin
si $(\alpha_v \neq \alpha)$
alors
  afficher (WARNING: Assertions Non Causales)
fin

13.7 Evaluation des spécifications

L'évaluation des spécifications correspond au calcul d'une fonction booléenne σ_v définie par l'équation de point fixe suivante:

$$\sigma_v = \nu y.\ [\alpha_v \Rightarrow [\sigma \wedge \widetilde{Pre}(y)]$$

On peut appliquer à l'évaluation des spécifications les mêmes optimisations que celles utilisées pour l'évaluation des assertions: calcul de couverture pour les opérations complexes et calcul de point fixe optimisé. Mais auparavant, on peut simplifier l'expression du point fixe calculé.

13.7.1 Optimisation du calcul de point fixe

Dans l'algorithme standard d'évaluation des spécifications, le calcul de σ_v est effectué par évaluation des termes successifs de la suite:

$$\begin{cases} y_0 &= \alpha_v \Rightarrow \sigma \\ y_{n+1} &= y_n \wedge (\alpha_v \Rightarrow [\sigma \wedge \widetilde{Pre}(y_n)]) \end{cases}$$

On peut alors montrer par récurrence que:

$$\forall\ n \geq 0,\ y_{n+1} = \alpha_v \Rightarrow [\sigma \wedge \widetilde{Pre}(y_n)]$$

Pour $n = 0$:

$$y_1 = (\alpha_v \Rightarrow \sigma) \wedge (\alpha_v \Rightarrow [\sigma \wedge \widetilde{Pre}(y_0))$$

d'où, après simplifications:

$$y_1 = \alpha_v \Rightarrow [\sigma \wedge \widetilde{Pre}(y_0)]$$

Donc l'égalité est vérifiée.

Pour $n > 0$, on suppose que $y_n = \alpha_v \Rightarrow [\sigma \wedge \widetilde{Pre}(y_{n-1})]$. On a alors:

$$y_{n+1} = \alpha_v \Rightarrow [\sigma \wedge \widetilde{Pre}(y_{n-1}) \wedge \widetilde{Pre}(y_n)]$$

Or $y_n \Rightarrow y_{n-1}$, donc d 'après les propriétés de l'opérateur $\widetilde{Pre}$:

$$\widetilde{Pre}(y_{n-1}) \wedge \widetilde{Pre}(y_n) = \widetilde{Pre}(y_n)$$

ce qui achève la démonstration.

Le calcul de σ_v revient donc à la génération des termes de la suite:

$$\begin{cases} y_0 &= \alpha_v \Rightarrow \sigma \\ y_{n+1} &= \alpha_v \Rightarrow [\sigma \wedge \widetilde{Pre}(y_n)] \end{cases}$$

Dés lors, on peut alors montrer que:

$$\forall\, n \geq 0,\ y_{n+1} = \alpha_v \Rightarrow [y_n \wedge \widetilde{Pre}(y_n)]$$

En effet:

$$\alpha_v \Rightarrow [y_n \wedge \widetilde{Pre}(y_n)] = \alpha_v \Rightarrow [(\alpha_v \Rightarrow [\sigma \wedge \widetilde{Pre}(y_{n-1})]) \wedge \widetilde{Pre}(y_n)]$$

d'où:

$$\alpha_v \Rightarrow [y_n \wedge \widetilde{Pre}(y_n)] = \alpha_v \Rightarrow [\sigma \wedge \widetilde{Pre}(y_{n-1}) \wedge \widetilde{Pre}(y_n)]$$

et, finalement:

$$\forall\, n \geq 0,\ \alpha_v \Rightarrow [y_n \wedge \widetilde{Pre}(y_n)] = \alpha_v \Rightarrow [\sigma \wedge \widetilde{Pre}(y_n)]$$

Le calcul de σ_v revient donc à la génération des termes de la suite:

$$\begin{cases} y_0 &= \alpha_v \Rightarrow \sigma \\ y_{n+1} &= \alpha_v \Rightarrow [y_n \wedge \widetilde{Pre}(y_n)] \end{cases}$$

Ce sont les termes de cette suite qui vont être calculés dans l'algorithme optimisé qui est présenté ci-dessous.

13.7.2 Algorithme optimisé

L'évaluation des spécifications correspondant à un calcul de plus grand point fixe, on peut mettre en œuvre les optimisations proposées dans la section 13.3.3. La combinaison des optimisations ci-dessus donne un nouvel algorithme. Pour le décrire, nous employons les mêmes notations que celles de l'algorithme standard. Toutefois, certaines variables supplémentaires sont définies:

- σ est la fonction de sortie du modèle, qui correspond aux spécifications à vérifier.
- α_v est la fonction caractéristique des transitions valides.
- σ_c et σ_p correspondent respectivement au terme courant (d'où σ_c) et au terme précédent (d'où σ_p) de la suite Σ définie dans la section 13.3.3.
- σ'_c et σ'_p correspondent respectivement au terme courant et au terme précédent de la suite Σ' définie dans la section 13.3.3.
- σ''_c correspondent au terme courant de la suite Σ'' définie dans la section 13.3.3.

L'algorithme ci-dessous traduit alors l'évaluation des assertions dans le domaine fonctionnel. Il est directement implémentable sur les BDDs.

debut
$\sigma_c = \alpha_v \Rightarrow \sigma$
$\sigma_p = 1$
$\sigma'_c = 1$

$\sigma''_c = 1$
tantque $(\sigma_p \neq \sigma_c)$ **faire**
debut
$\sigma'_p = \sigma'_c$
$\sigma'_c = \sigma'_c \wedge \widetilde{Src}(\sigma_c \Uparrow \sigma_p)$
si $\sigma'_c(q_{init})$
alors
debut
$\sigma''_p = \sigma''_c$
$\sigma''_c = 1$
soit $\mathcal{C}_\wedge(\sigma'_c \Uparrow \sigma'_p) = \{\sigma'_1, \ldots, \sigma'_n\}$
pour *chaque* $\sigma'_i \in \mathcal{C}_\wedge(\sigma'_c \Uparrow \sigma'_p)$ **faire**
$\sigma''_c = \sigma''_c \wedge \sigma'_i \circ \vec{\delta}$
$\sigma_p = \sigma_c$
$\sigma_c = \alpha_v \Rightarrow (\sigma_c \wedge (\sigma''_c \Uparrow \sigma''_p))$
fin
sinon
retourner (Spécifications Non Satisfaites)
fin
retourner (Spécifications Satisfaites)
fin

13.7.3 Optimisation en l'absence d'assertions

Le cas particulier où la vérification s'effectue en l'absence d'assertions est intéressant à traiter, car il permet une optimisation supplémentaire par rapport à l'algorithme qui vient d'être présenté. En effet, l'absence d'assertions correspond à $\alpha = 1$. Dans ce cas, l'évaluation des assertions fournit $\alpha_v = 1$. L'expression du point fixe correspondant à l'évaluation des spécifications se simplifie alors sous la forme:

$$\sigma_v = \nu y.\ [\sigma \wedge \widetilde{Pre}(y)]$$

L'optimisation supplémentaire qui peut alors être faite consiste à calculer le point fixe ci-dessus sous la forme d'un produit de points fixes. En effet, si $\sigma = \bigwedge_{i=1}^{n} \sigma_i$, on a:

$$\sigma_v = \nu y.\ [\bigwedge_{i=1}^{n} \sigma_i \wedge \widetilde{Pre}(y)]$$

et d'après les propriétés de points fixes:

$$\sigma_v = \bigwedge_{i=1}^{n} [\nu y.\ \sigma_i \wedge \widetilde{Pre}(y)]$$

Donc il est possible de calculer σ_v à partir d'une couverture conjonctive de σ grâce à la propriété ci-dessus. On obtient alors l'algorithme suivant:

debut

$\sigma_v = 1$
soit $\mathcal{C}_\wedge(\sigma) = \{\sigma_1, \ldots, \sigma_m\}$
pour *chaque* $\sigma_i \in \mathcal{C}_\wedge(\sigma)$ **faire**
debut
$\sigma_c = \sigma_i \Uparrow \sigma_v$
$\sigma_p = 1$
$\sigma'_c = 1$
$\sigma''_c = 1$
tantque $(\sigma_p \neq \sigma_c)$ **faire**
debut
$\sigma'_p = \sigma'_c$
$\sigma'_c = \sigma'_c \wedge \widetilde{Src}(\sigma_c \Uparrow \sigma_p)$
si $\sigma'_c(q_{init})$
alors
debut
$\sigma''_p = \sigma''_c$
$\sigma''_c = 1$
soit $\mathcal{C}_\wedge(\sigma'_c \Uparrow \sigma'_p) = \{\sigma'_1, \ldots, \sigma'_n\}$
pour *chaque* $\sigma'_j \in \mathcal{C}_\wedge(\sigma'_c \Uparrow \sigma'_p)$ **faire**
$\sigma''_c = \sigma''_c \wedge \sigma'_j \circ \vec{\delta}$
$\sigma_p = \sigma_c$
$\sigma_c = \sigma_c \wedge (\sigma''_c \Uparrow \sigma''_p)$
fin
sinon
retourner (Spécifications Non Satisfaites)
fin
$\sigma_v = \sigma_v \wedge \sigma_c$
fin
retourner (Spécifications Satisfaites)
fin

Chapitre 14

Expérimentations

Les algorithmes d'évaluation des assertions et des spécifications d'un programme LUSTRE sur son modèle d'exécution qui sont décrits dans les chapitres 12 et 13 ont été implémentés au sein d'outils de vérification. Ces outils sont basés sur une librairie efficace de BDDs spécifiquement conçue dans ce but, et contenant tous les opérateurs symboliques nécessaires à la mise en œuvre des algorithmes. Ces deux outils sont donc issus d'une traduction directe des algorithmes de vérification décrits précédemment. L'un met en œuvre l'implémentation standard à l'aide de tous les opérateurs décrits dans le chapitre 12. L'autre correspond à l'implémentation optimisée, dans laquelle toutes les optimisations relatées dans le chapitre 13 ont été intégrées. Nous relatons ici les résultats des expérimentations[1] menées sur ces versions "standard" et "optimisée" de la méthode de vérification en arrière. Ceux-ci proviennent de la vérification de programmes LUSTRE issus pour la plupart de descriptions de circuits [EJ90,GT92,HL90]. Les programmes choisis ont été sélectionnés uniquement en raison de la taille (nombre de variables) de leur modèle d'exécution, et non pas en fonction de possibles "prédispositions" à être vérifiés efficacement par la méthode en arrière.

14.1 Comparaison avec la méthode en avant

Dans un premier temps, les algorithmes développés pour la méthode en arrière ont été comparés avec l'outil actuel de vérification par la méthode énumérative en avant. Les résultats obtenus sont regroupés dans la figure 14.1. Pour chaque programme, la colonne "#SV" indique le nombre de variables d'état et la colonne "#IV" indique le nombre de variables d'entrée. Pour la méthode énumérative en avant, les indications entre parenthèses correspondent aux cas où la vérification n'a pu être achevée, à cause d'une limitation technique de l'outil utilisé, qui n'est pas capable de vérifier des machines ayant plus de 50000 états accessibles. Les chiffres entre parenthèses correspondent donc au cas où cette limite est atteinte. Ils indiquent respectivement le nombre d'états effectivement vérifiés par l'outil et le temps CPU au bout duquel la vérification est abandonnée.

Pour la méthode en arrière, son implémentation sur les BDDs nécessite de déterminer un ordre des variables. Pour les comparaisons avec la méthode en avant, l'ordre choisi est l'ordre naïf déterminé "aléatoirement" par le compilateur LUSTRE. Les temps entre parenthèses corre-

[1] Résultats obtenus sur Sun Sparc Station

Méthode			AV enum.		AR std.	AR opt.
Programme	#SV	#IV	#états	CPU	CPU	CPU
arbiter2.ec	34	6	83	2.9	21.1	15.3
arbiter3.ec	30	6	2890	121.0	34.0	47.3
arbiter.ec	38	4	10334	262.2	935.8	142.4
clm1.ec	62	16	(21992)	(945.4)	24.9	14.4
clm5.ec	67	16	(28326)	(1148.3)	(3160.0)	841.3
clm7.ec	65	16	(20866)	(832.4)	6.3	8.2
ess3.ec	23	13	(12778)	(229.8)	0.4	0.8
minmax.ec	37	6	3525	103.8	(307.8)	98.3
seq.ec	30	13	(14526)	(627.5)	21.5	8.8
ver.ec	47	5	4258	50.8	305.3	42.3
verif1.ec	30	4	39	1.0	4.8	3.6
verif3.ec	35	6	245	10.7	10.8	8.5
verif_n.ec	39	7	312	54.8	622.2	103.1

Figure 14.1: Comparaison avec la méthode en avant

spondent alors aux cas où la vérification n'a pu être achevée, faute de mémoire pour les BDDs. Toutefois, il faut indiquer à ce sujet que tous les tests ont été effectués avec 4M octets de mémoire pour les BDDs, ce qui est faible. Les temps entre parenthèses indiquent la durée au bout de laquelle la vérification est abandonnée.

On constate que sauf pour les toutes petites machines, la méthode en arrière est toujours meilleure que la méthode énumérative en avant. Notons que ce résultat est indépendant des limitations techniques imposées à la méthode en avant, car lorsque celle-ci échoue, la méthode en arrière a déjà achevé la vérification avec succès depuis longtemps.

D'autre part, si on compare les résultats des versions standard et optimisée des algorithmes de vérification en arrière, on constate que lorsque le temps de calcul de l'algorithme standard est court (inférieur à 5 s.), celui-ci est plus performant que l'algorithme optimisé. Cela peut s'expliquer par le fait que les opérations supplémentaires liées aux optimisations dans la version optimisée sont trop peu nombreuses pour être rentables. Par contre, dés que le temps de calcul commence à être conséquent, la version optimisée est bien meilleure que la version standard.

14.2 Versions standard et optimisée

Nous avons ensuite comparé les versions standard et optimisée des algorithmes de vérification par la méthode en arrière. Pour cela, nous avons effectué deux séries de tests:

- La première a été réalisée avec un ordre local des variables.
- La seconde a été réalisée à partir d'un ordre global des variables.

Les résultats obtenus sont donnés ci-dessous.

14.2.1 Ordonnancement local

Les deux versions de l'algorithme en arrière ont été testées avec un ordre local calculé par la méthode décrite dans la section 12.1, déterminé à partir du réseau d'opérateurs représentant les spécifications à vérifier. Les résultats obtenus sont les suivants:

Méthode			AR std.	AR opt.
Programme	#SV	#IV	CPU	CPU
arbiter2.ec	34	6	4.3	5.5
arbiter3.ec	30	6	3.8	6.1
arbiter.ec	38	4	1371.1	918.1
clm1.ec	62	16	7.1	7.0
clm5.ec	67	16	(5869.0)	(545.6)
clm7.ec	65	16	3.4	3.8
ess3.ec	23	13	0.8	0.8
minmax.ec	37	6	303.0	73.9
seq.ec	30	13	9.7	8.8
ver.ec	47	5	32.0	41.6
verif1.ec	30	4	2.1	2.9
verif3.ec	35	6	12.9	6.5
verif_n.ec	39	7	16.6	16.6

On constate à nouveau que lorsque le temps de calcul de la version standard est court, celle-ci est légèrement plus efficace que la version optimisée.

Si on compare maintenant ces temps de calcul avec ceux obtenus pour un ordre naïf, les performances des deux versions sont en général très nettement améliorées. Cela démontre à quel point les BDDs sont sensibles à l'ordre fixé au départ. Selon celui-ci, les temps de calcul sont facilement décuplés et parfois, la vérification ne peut s'achever par manque de mémoire.

Il existe aussi des cas (`arbiter.ec`, `clm5.ec`) où l'ordre local obtenu est moins bon que l'ordre naïf, ce qui engendre de mauvais résultats. Cela prouve que l'heuristique mise en œuvre pour le calcul d'un ordre local n'est pas infaillible.

14.2.2 Ordonnancement global

Nous avons enfin testé les algorithmes avec un ordonnancement global des variables, déterminé à partir des spécifications et de la fonction de transition du modèle à vérifier. Les résultats obtenus sont les suivants:

Méthode			AR std.	AR opt.
Programme	#SV	#IV	CPU	CPU
arbiter2.ec	34	6	30.7	5.5
arbiter3.ec	30	6	107.1	16.1
arbiter.ec	38	4	263.0	49.6
clm1.ec	62	16	43.6	21.6
clm5.ec	67	16	(7430.3)	457.6
clm7.ec	65	16	7.5	8.8
ess3.ec	23	13	1.1	1.2
minmax.ec	37	6	(328.1)	26.5
seq.ec	30	13	19.2	8.4
ver.ec	47	5	145.3	39.6
verif1.ec	30	4	4.8	2.9
verif3.ec	35	6	12.7	7.4
verif_n.ec	39	7	213.6	19.8

On constate que par rapport à l'ordonnancement local, les résultats sont assez mitigés. Pour la version standard, ils sont franchement moins bons qu'auparavant. Par contre, pour la version optimisée les gains obtenus sur les vérifications "coûteuses" (`arbiter.ec`, `minmax.ec`) sont très significatifs, tandis que les résultats sont moins bons pour les vérifications "simples". On peut attribuer cela au calcul de l'ordre global, dont le coût est plus élevé que celui de l'ordre local, et qui n'est pas compensé lorsque les temps de calcul sont courts. En outre, les cas où la version optimisée fonctionne mal correspondent à de mauvaises performances de l'opérateur de composition fonctionnelle restreinte. Pour résoudre ce problème, il faudrait envisager de basculer de cet opérateur vers l'opérateur classique dés qu'il devient "trop" exponentiel (voir section 13.5.2).

14.2.3 Conclusions

Le choix d'un ordre non aléatoire dans la méthode en arrière provoque pratiquement toujours un gain très substantiel en mémoire et en temps de calcul. De plus, pour les exemples traités, il existe toujours un ordre permettant à la vérification d'aboutir.

D'autre part, la version optimisée de la méthode est pratiquement toujours nettement plus performante que la version standard pour les vérifications complexes. En outre, elle est moins sensible aux problèmes de saturation de la mémoire (appel au *garbage-collector*).

En conclusion, ces expérimentations mettent en évidence l'importance de l'ordre des variables sur l'efficacité de la vérification. Elles soulèvent par conséquent un problème intéressant consistant à déterminer *quels* sont les "bons" ordres pour la vérification, et *pourquoi* ils sont bons.

14.3 Conclusions

14.3.1 La méthode en arrière

Les expérimentations réalisées prouvent que la méthode de vérification en arrière fonctionne très bien, surtout comparée à la méthode énumérative en avant. Le fait que les calculs soient guidés par les assertions et les spécifications des programmes semble être un facteur déterminant pour l'efficacité de la méthode. En outre, celle-ci apporte une solution définitive aux problèmes intrinsèques liés à la vérification des programmes LUSTRE.

14.3.2 La technique symbolique

Sur le plan de l'implémentation, l'efficacité des méthodes de génération et de vérification de modèles dépend de deux paramètres:

- Leur coût en mémoire. Cet aspect touche aux techniques utilisées pour la représentation du modèle en mémoire.
- Leur coût en temps de calcul. Cet aspect touche à la complexité des algorithmes mis en œuvre pour générer ou parcourir le modèle.

Les diverses implémentations proposées des méthodes existantes cherchent à minimiser ces facteurs. Cependant, la technique de représentation du modèle reste une priorité, afin de limiter les problèmes d'explosion à la génération.

On peut maintenant dresser un bilan global de l'approche symbolique pour la vérification des machines séquentielles, en faisant la synthèse des différents travaux menés dans ce domaine concernant l'implémentation de la méthode en avant et de la méthode en arrière.

En premier lieu, il faut noter que pratiquement toutes les méthodes de vérification basées sur l'approche symbolique emploient la même technique de représentation: les BDDs. Il est indéniable que le succès de ces méthodes est très largement lié aux performances des BDDs, tant au niveau de la compacité des objets représentés qu'à celui des opérateurs qui permettent de les manipuler.

D'autre part, l'approche "tout symbolique" de la vérification consiste à utiliser les BDDs à la fois en tant que technique de représentation et en tant que technique de manipulation des machines séquentielles. C'est celle qui a été utilisée pour implémenter la vérification en arrière, ainsi que la vérification en avant. Pourtant, l'objectif initial de l'approche symbolique réside dans la possibilité de rendre la taille de la représentations des machines indépendante de leur propre taille.

Les expériences de "vérification symbolique" menées autour de la méthode en avant et de la méthode en arrière montrent que les BDDs ont des limites:

- La représentation des objets manipulés au cours de la vérification n'est pas toujours suffisamment compacte, d'où une augmentation des coûts en mémoire.
- Bien qu'étant pour la plupart d'un coût raisonnable en temps de calcul, les opérateurs sur les BDDs génèrent souvent beaucoup de résultats intermédiaires, qui sont gros consommateurs

de mémoire. Souvent, il y a suffisamment de mémoire pour représenter le résultat final d'une opération entre BDDs, mais celui-ci ne peut être obtenu car il n'y a pas assez de mémoire pour conserver tous les résultats intermédiaires nécessaires à l'achèvement des calculs.

Finalement, le problème majeur des BDDs réside dans leur coût en mémoire. La solution consiste alors à revenir à la raison initiale de leur utilisation, qui concerne la représentation du modèle. Nous avons donc développé une approche *hybride*, basée sur une interprétation différente des BDDs, et permettant de définir des opérateurs économiques pour manipuler les modèles.

Dans la partie suivante, nous explicitons la nouvelle interprétation donnée aux BDDs, puis nous l'appliquons à la vérification énumérative par la méthode en avant.

Partie IV

Autres Approches

Chapitre 15

Autre interprétation des BDDs

Nous avons vu que les BDDs constituent une forme de représentation de fonctions booléennes quelconques, dont l'interprétation première correspond à l'expansion de Shannon complète des fonctions selon un ordre fixé des variables encodant leur domaine de définition. Néanmoins, les BDDs peuvent s'interpréter de façon plus simple et plus intuitive sous forme de *sommes de monômes*. En effet, du point de vue structurel, le BDD d'une fonction est un graphe orienté acyclique, constitué de nœuds dénotant les variables de la fonction, et d'arcs orientés reliant ces nœuds entre eux. Tout BDD possède un unique nœud *racine*, et deux nœuds *feuilles* qui sont les représentations des fonctions constantes 0 et 1. Chaque chemin d'un BDD reliant sa racine à une feuille s'interprète alors comme une valuation des variables de la fonction représentée, pour laquelle la valeur de la feuille en indique le résultat. Les BDDs peuvent donc être interprétés comme une forme de représentation symbolique des fonctions booléennes sous forme de somme de monômes.

On peut alors se baser sur cette nouvelle interprétation pour distinguer deux types de fonctions booléennes: les fonctions s'exprimant sous la forme d'un monôme, et les autres. Cette distinction nous amène à envisager une approche *intermédiaire* pour la représentation et la manipulation des fonctions booléennes, dans laquelle doivent être définis:

- Une forme de représentation des monômes.
- Une forme de représentation des fonctions quelconques.
- Des opérateurs entre monômes et fonctions quelconques.

Il est clair que les BDDs restent le moyen le plus adapté pour la représentation des fonctions quelconques. Dans la suite, nous ne nous intéressons donc qu'à la représentation des monômes et à la définition d'opérateurs entre monômes et fonctions quelconques.

15.1 Représentation de monômes

Nous faisons brièvement quelques rappels de définition et de terminologie concernant la notion de monômes et leur relation avec les fonctions booléennes quelconques.

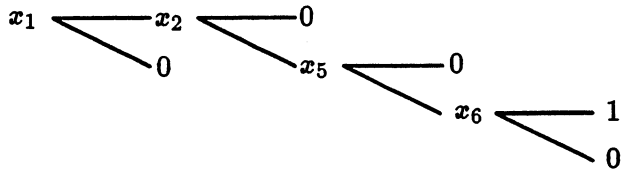


Figure 15.1: Représentation du monôme $x_1.\bar{x}_2.\bar{x}_5.x_6$ par un BDD

Définition 4 *Etant donné un n-uplet de variables booléennes* $X = (x_1, \ldots, x_n) \in \{0,1\}^n$, *un monôme est un produit de* p $(1 \leq p \leq n)$ *variables distinctes de* X *sous forme normale* (x) *ou complémentée* $(\neg x)$, p *étant la longueur du monôme.*

Un monôme sur X est *complet* si et seulement si il est de longueur n. Tout monôme non complet sera dit *quelconque.*

Définition 5 *Etant donnée une fonction booléenne* $f : \{0,1\}^n \mapsto \{0,1\}$ *définie sur* X, *toute valeur de* X *est appelée* point *de* f. *L'ensemble des points pour lesquels* f *est vraie forme sa* couverture *à* 1.

Un monôme quelconque de longueur p couvre 2^{n-p} points de f. Un monôme complet couvre un seul point de f. Nous étudions maintenant deux formes de représentation des monômes quelconques.

15.1.1 Représentation à l'aide des BDDs

Un monôme est un genre particulier de fonction booléenne, qu'il est donc parfaitement possible de représenter par un BDD. Ce dernier se réduit alors à un graphe linéaire du type de l'exemple donné dans la figure 15.1. Tout monôme de longueur p est alors représenté par un BDD constitué de p nœuds. D'un point de vue "local", la représentation d'un monôme est relativement chère car chaque nœud a une taille assez importante. Il parait donc nécessaire de définir une forme de représentation spécifique pour les monômes. Ce point est abordé dans la section suivante.

15.1.2 Représentation à l'aide de vecteurs booléens

Les monômes quelconques peuvent être représentés de manière plus compacte qu'avec les BDDs sous la forme de vecteurs booléens. En effet, dans un monôme quelconque, chaque variable prend la valeur 0, 1 ou une valeur indéterminée φ. L'encodage de ces valeurs nécessite seulement 2 bits pour la représentation. Donc un monôme de longueur p peut être représenté à l'aide d'un vecteur booléens de longueur $2p$. La figure 15.2 donne un exemple de monôme représenté par un vecteur de bits. Dans ce vecteur, la valeur φ est encodée par le couple $(0,0)$.

	1	2	3	4	5	6
x	1	0	0	0	0	1
$\bar{x}$	0	1	0	0	1	0

Figure 15.2: Représentation du monôme $x_1.\bar{x}_2.\bar{x}_5.x_6$ sous forme de vecteur booléen

15.1.3 Comparaison des représentations

Dans un premier temps, on peut comparer les BDDs et les vecteurs booléens au niveau de la mémoire nécessaire pour représenter les monômes. La représentation d'un monôme de longueur p nécessite exactement p nœuds avec les BDDs. A l'implémentation, un nœud de BDD est constitué d'environ 20 octets, donc la représentation d'un monôme de longueur p nécessite $80p$ bits. Le même monôme représenté par un vecteur nécessitera seulement $2n$ bits, indépendamment de sa longueur (n est la longueur maximale des monômes). Le gain γ obtenu grâce aux vecteurs booléens est alors minimal pour les monômes de longueur 1, et maximal pour les monômes de longueur n. Il se situe donc dans la fourchette ci-dessous:

$$\frac{40}{n} \leq \gamma \leq 40$$

Dans le meilleur des cas, le gain obtenu la représentation des monômes sous forme de vecteurs booléens est linéaire par rapport aux BDDs. Cependant, il faut remarquer que ce gain s'effondre dés qu'il s'agit de représenter des ensembles de monômes. En effet, la représentation d'un ensemble sous forme d'une énumération de vecteurs booléens est linéaire par rapport à sa taille, alors qu'une représentation efficace à l'aide des BDDs est possible grâce au partage des nœuds.

D'un autre point de vue, l'accès à la valeur de toute variable dans un monôme est déterminée en temps constant sur les vecteurs booléens, tandis qu'elle est linéaire sur les BDDs. De ce point de vue, les vecteurs booléens présentent un avantage certain par rapport aux BDDs.

Les deux formes de représentation des monômes présentées ci-dessus sont donc complémentaires. Les vecteurs booléens s'avèrent plus intéressants pour la représentation et la manipulation individuelle des monômes, tandis que les BDDs sont plus efficaces pour la gestion d'un ensemble de monômes. Il parait donc intéressant d'utiliser ces deux formes de représentation de manière combinée. Cela nécessite de disposer d'un certain nombre d'opérateurs "hybrides" entre BDDs et vecteurs booléens. Ces opérateurs sont décrits dans la section suivante.

15.2 Opérateurs hybrides

Nous définissons ici un certain nombre d'opérateurs dits "hybrides" car ceux-ci opèrent sur deux formes de représentation simultanément: les BDDs et les vecteurs booléens. On peut remarquer que tout vecteur booléen est convertible sous forme de BDD, donc on peut disposer après conversion de tous les opérateurs déjà existants sur les BDDs. Cependant, certains opérateurs peuvent être redéfinis avantageusement sous forme d'opérateurs hybrides, dont voici les principaux:

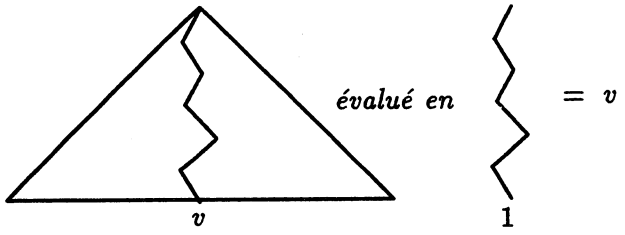


Figure 15.3: Evaluation d'un BDD sur un monôme

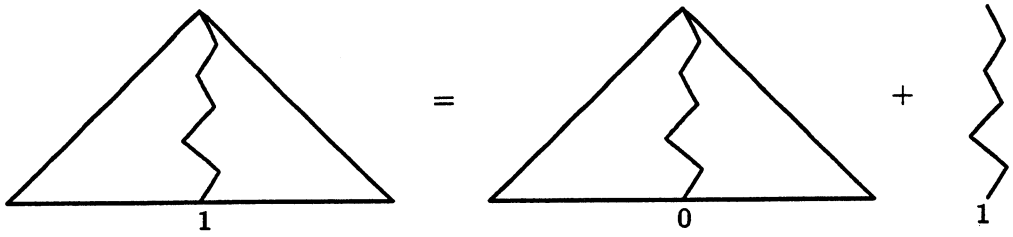


Figure 15.4: Retrait d'un monôme dans un BDD

- **Conversion:** Il s'agit des opérateurs de conversion d'un vecteur booléen vers un BDD et, réciproquement, d'un BDD représentant un monôme vers un vecteur booléen. Ces opérateurs sont en $O(n)$, où n est la longueur maximale des monômes.
- **Evaluation:** Cet opérateur permet d'évaluer la valeur d'une fonction représentée par un BDD sur un monôme représenté par un vecteur. Son coût est en $O(n)$. La figure 15.3 schématise le principe de l'opérateur d'évaluation.
- **Retrait et Ajout:** Ils permettent respectivement d'extraire un monôme inclus dans un BDD, ou d'ajouter un monôme à un BDD. Leur coût est en $O(n)$. La figure 15.4 et la figure 15.5 illustrent le principe de ces opérateurs.
- **Restriction:** Cet opérateur définit la restriction d'un BDD à un monôme. Dans certaines conditions qui seront précisées ultérieurement, son coût est en $O(n)$.

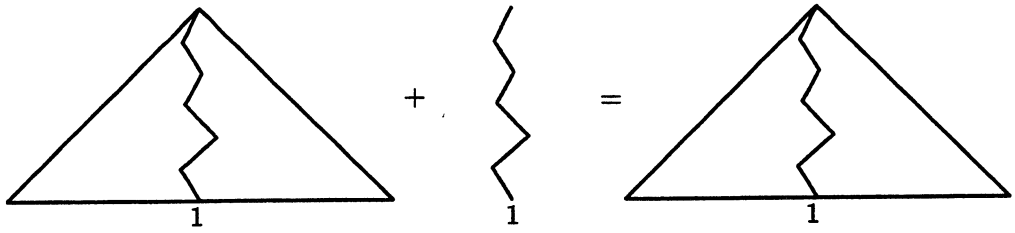


Figure 15.5: Ajout d'un monôme dans un BDD

L'intérêt de ces opérateurs est leur faible coût à la fois en mémoire et en temps de calcul, du fait de leur linéarité.

Chapitre 16

La méthode énumérative et les BDDs

Il existe actuellement deux approches différentes pour implémenter la méthode de vérification en avant:

- Le "tout énumératif". Le modèle est alors représenté en extension, et il est généré de manière énumérative. Dans cette approche, les calculs effectués pour générer le modèle sont très simples, mais son inconvénient réside dans le fait que la représentation du modèle est linéaire par rapport au modèle lui-même.

- Le "tout symbolique". Le modèle est alors représenté en compréhension, et il est généré de manière symbolique. Par cette approche, la représentation du modèle est plus compacte, mais les opérateurs symboliques utilisés pour sa génération sont d'un coût très élevé.

Nous proposons ici une implémentation de la vérification par la méthode en avant, fondée sur une approche intermédiaire, combinant une représentation compacte du modèle et une génération de ce dernier basée sur l'utilisation d'opérateurs simples et économiques. Cette approche est basée sur l'interprétation des BDDs sous forme de sommes de monômes. Dans un premier temps, nous décrivons une implémentation de l'algorithme énumératif dans laquelle la génération du modèle est basée sur les BDDs. Puis, nous étendons l'utilisation des BDDs à la représentation du modèle généré.

16.1 Principe de l'approche énumérative

Nous affinons ci-dessous l'algorithme défini dans la section 7.3.1, qui décrivait la vérification du modèle à la volée. Nous faisons ainsi apparaître les opérateurs hybrides définis dans la section 15.2 entre fonctions booléennes quelconques et monômes. Les notations sont les suivantes:

- V désigne l'ensemble des états Visités.

- A désigne l'ensemble des états Accessibles mais non visités.

L'algorithme est donné ci-dessous:

debut
$V = \emptyset$
$A = \{q_{init}\}$
tantque $A \neq \emptyset$ **faire**
  debut
  soit $q \in A$
  $A = A \setminus \{q\}$
  $V = V \cup \{q\}$
  pour *chaque* e / $(q,e) \in \alpha_v$ **faire**
    si $(q,e) \notin \sigma$
    alors
      retourner (Spécifications Non Satisfaites)
    sinon
      debut
      $q' = \vec{\delta}(q,e)$
      si $q' \notin V$ **et** $q' \notin A$
      alors
        $A = A \cup \{q'\}$
      fin
  fin
retourner (Spécifications Satisfaites)
fin

Dans cet algorithme, les seuls objets manipulés sont les états et les entrées du modèle. Les ensembles A et V peuvent être représentés de manière extensive par *énumération* des états qui leur appartiennent. Or, si le modèle est encodé dans le domaine booléen, les états et les entrées sont représentés par des monômes *complets* sur les variables d'état et les variables d'entrée de celui-ci. Les fonctions définissant le modèle peuvent alors être traduites par des fonctions booléennes représentées sous forme de sommes de monômes.

L'algorithme ci-dessus peut donc être implémenté à partir des monômes. Il suffit pour cela de traduire les opérateurs nécessaires à son fonctionnement sur ce type de représentation. Or ces opérateurs sont d'une grandes simplicité: ajout et retrait d'un état dans un ensemble, test d'appartenance d'un état à un ensemble, évaluation de fonction.

L'implémentation actuelle de LESAR est basée sur une représentation des monômes sous forme de vecteurs booléens. Cette représentation possède l'avantage d'être la plus compacte mais l'expérience montre qu'elle présente deux inconvénients majeurs:

- La représentation des ensembles est proportionnelle à leur taille, d'où un risque d'échec de l'algorithme dû à l'explosion des modèles.
- Les opérations sur les vecteurs booléens sont peu efficaces.

En plus, cet outil doit faire face à un problème impossible à résoudre par la méthode en avant: la prise en compte des assertions.

Nous proposons ici deux implémentations de la méthode énumérative permettant de solutionner ces problèmes grâce à l'utilisation des BDDs. La première est une approche intermédiaire, dans laquelle seuls les calculs effectués dans l'algorithme sont réalisés à l'aide des BDDs, tandis que la représentation du modèle reste confiée aux vecteurs booléens. Dans la seconde, les BDDs sont aussi utilisés pour représenter le modèle. Dans les sections suivantes, nous décrivons ces deux implémentations et nous mettons en évidence l'intérêt d'utiliser les BDDs.

16.2 Génération du modèle à l'aide des BDDs

Dans cette première implémentation basée sur les BDDs, les fonctions définissant le modèle (fonctions d'assertions, de sortie et de transition) sont représentées par des BDDs, alors que le modèle reste représenté par des vecteurs booléens. Ainsi, les opérations sur le modèle sont effectuées à l'aide d'opérateurs sur les BDDs, tandis que les ensembles d'états manipulés sont exprimés sous la forme d'une énumération de leurs éléments. L'implémentation à l'aide des BDDs des calculs de génération du modèle par la méthode énumérative apportent une solution à deux des problèmes rencontrés dans les implémentations classiques, en ce sens qu'ils permettent:

- D'améliorer l'efficacité des opérateurs.
- De prendre en compte des assertions.

Nous détaillons ci-dessous les avantages apportés par les BDDs sur ces deux points.

16.2.1 Implémentation des opérateurs

L'utilisation des BDDs pour la génération permet de diminuer la complexité des calculs en exploitant leurs propriétés. Le gain obtenu grâce à ces derniers est lié au fait que tous les calculs s'effectuent dans *un état donné*. Cela permet de simplifier les fonctions du modèle dans chaque état traité, grâce à l'opérateur hybride de restriction d'un BDD à un monôme donné. On obtient ainsi directement les fonctions partielles du modèle pour un état q_c quelconque:

$$\alpha_{q_c}(e) = \alpha(q,e) \Uparrow q_c$$

$$\sigma_{q_c}(e) = \sigma(q,e) \Uparrow q_c$$

$$\forall\, 1 \leq i \leq n,\ \delta_{iq_c}(e) = \delta_i(q,e) \Uparrow q_c$$

De plus, en choisissant l'ordre des variables correctement, les opérations de restriction effectuées ne nécessitent *aucun* calcul symbolique, d'où un coût extrêmement faible, en $O(n)$. En effet, si toute variable d'état précède toute variable d'entrée dans l'ordre, la restriction à un état donné q_c d'une fonction $f\,(q,e)$ sur les états et les entrées du modèle correspond exactement à l'évaluation "partielle" de f en q_c, soit $f\,(q_c,e)$. En termes de BDDs, le BDD de f_{q_c} est un sous-BDD de f. La figure 16.1 illustre le résultat d'une telle restriction.

D'autre part, l'utilisation des BDDs dans les calculs permet de profiter de leurs propriétés sémantiques. En particulier, ceux-ci possèdent la caractéristique remarquable de ne faire intervenir dans la représentation d'une fonction *que* les variables qui influencent la définition de

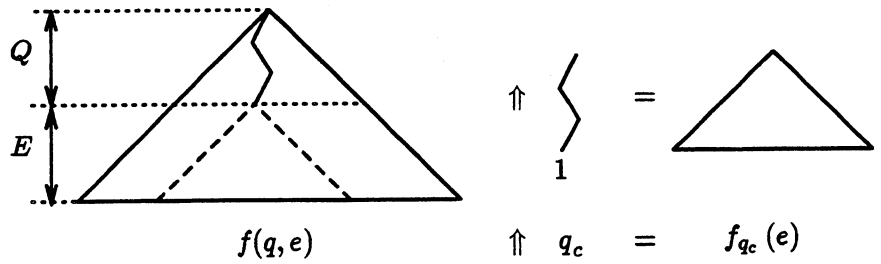


Figure 16.1: Restriction d'une fonction du modèle à un état donné

celle-ci. Autrement dit, toute variable qui n'a aucune influence sur le résultat de la fonction n'apparaît pas dans son BDD, et ce quel que soit l'ordre. Cette caractéristique peut être exploitée de manière très intéressante pour le calcul des successeurs des états du modèle généré.

L'algorithme classique d'évaluation des successeurs d'un état q consiste à générer tous les vecteurs d'entrée possibles e du modèle, et à évaluer pour chacun d'eux l'état successeur $q' = \vec{\delta}(q, e)$. Cet algorithme est en $O(2^m)$, où m est le nombre de variables d'entrée du modèle. Une description schématique de l'algorithme est donnée ci-dessous. Les objets définis et manipulés sont les suivants:

- q est l'état dont on cherche les successeurs. Il est représenté par un vecteur booléen de longueur n.
- e dénote une entrée possible du modèle, elle aussi représentée par un vecteur booléen. Initialement, $e = [\varphi, \ldots, \varphi]$.
- $\vec{\delta} = [\delta_1, \ldots, \delta_n]$ est la fonction de transition du modèle, dénotée par un vecteur de fonctions booléennes.
- i est une variable entière contrôlant la fonction d'évaluation des successeurs. Initialement, $i = 1$.

fonction *calculer_successeurs* (q, e, i)
debut
si $i > n$
alors
    $q' = \vec{\delta}(q, e)$
sinon
    debut
    $e_i = 0$
    calculer_successeurs $(q, e, i + 1)$
    $e_i = 1$
    calculer_successeurs $(q, e, i + 1)$
    $e_i = \varphi$
    fin
fin

Le défaut de l'algorithme ci-dessus est d'être purement exponentiel par rapport aux nombre de variables d'entrée. En effet, une valeur est attribuée à chaque variable d'entrée du modèle avant que l'état successeur ne soit évalué, et indépendamment de savoir si cela est "nécessaire". En effet, si la fonction de transition est indépendante d'une variable, attribuer une valeur à cette dernière est coûteux et inutile. De là, il est possible d'imaginer un algorithme d'évaluation des successeurs dans lequel les variables d'entrée ne sont valuées que s'il est nécessaire de connaître leur valeur exacte. Un tel algorithme peut être schématisé de la façon suivante (les notations restant les mêmes par rapport à l'algorithme précédent):

fonction *calculer_successeurs* (q, e, i)
debut
si $i > n$
alors
    $q' = \vec{\delta}(q, e)$
sinon
si $\delta_i(q, e) = 0$ **ou** $\delta_i(q, e) = 1$
alors
    calculer_successeurs $(q, e, i + 1)$
sinon
    debut
    soit j *tel que* $\delta_i(q, e)$ *dépend de* e_j
    $e_j = 0$
    calculer_successeurs (q, e, i)
    $e_j = 1$
    calculer_successeurs (q, e, i)
    $e_j = \varphi$
    fin
fin

Dans cet algorithme, chaque fonction de transition partielle est successivement évaluée, en attribuant aux variables d'entrée une valeur "à la demande", uniquement si cela est nécessaire pour connaitre la valeur de la fonction à évaluer. Les propriétés sémantiques des BDDs rendent son implémentation réaliste, puisque les variables rencontrées dans les BDDs sont *toujours* nécessaires pour la définition des fonctions représentées.

16.2.2 Prise en compte des assertions

Le problème de la prise en compte des assertions ne peut être résolu qu'en calculant leur sémantique exacte. Or, nous avons donné un algorithme de calcul des assertions, dont l'implémentation est réalisée sur les BDDs. La sémantique des assertions est alors donnée par un BDD dénotant une fonction booléenne. En admettant que les assertions ont été évaluées préalablement par la méthode symbolique, et qu'elles dénotent une fonction α_v représentée par un BDD, celles-ci peuvent être aisément prises en compte dans le cadre d'une implémentation de l'algorithme de vérification énumérative basée sur les BDDs.

Les assertions forment une fonction $\alpha_v(q, e)$ sur les états et les entrées du modèle, qui spécifient les transitions valides de celui-ci. Pour un état q donné, ces mêmes assertions spécifient

l'ensemble des entrées e valides dans cet état. Une entrée e est valide si et seulement si $\alpha_v(q,e) = 1$. Les états successeurs q' de q sont alors les états pour lesquels il existe une entrée valide e telle que $q' = \vec{\delta}(q,e)$. La prise en compte des assertions nécessite alors simplement la définition d'une fonction générant toutes les entrées valides à partir d'un état, et calculant l'état successeur pour chacune d'elle. L'algorithme ci-dessous en donne une définition schématique. Les notations restent les mêmes que précédemment. Les règles terminales de cet algorithme sont les suivantes:

- Si l'entrée générée est valide, on calcule l'état successeur (appel à la fonction *calculer_successeurs*).
- Si l'entrée générée n'est pas valide, on ne fait rien.

fonction *generer_entree_valide*(q,e)
debut
si $\alpha_v(q,e) = 1$
alors
    calculer_successeurs$(q,e,1)$
sinon
si $\alpha_v(q,e) \neq 0$
alors
    debut
    soit j *tel que* $\alpha_v(q,e)$ *dépend de* e_j
    $e_j = 0$
    generer_entree_valide(q,e)
    $e_j = 1$
    generer_entree_valide(q,e)
    $e_j = \varphi$
    fin
fin

De même que pour le calcul des successeurs, les variables d'entrée ne sont valuées "qu'à la demande".

16.2.3 Evaluation des spécifications

La dernière étape de la vérification consiste à évaluer les spécifications. Celles-ci constituent une fonction $\sigma(q,e)$ sur les états et les entrées du modèle, donnant les transitions correctes de celui-ci. Pour un état q donné, elles déterminent l'ensemble des entrées correctes e du modèle. La vérification consiste alors à contrôler que chaque entrée valide (du point de vue des assertions) est une entrée correcte du modèle, ce qui revient à évaluer les spécifications pour chaque entrée valide. La fonction ci-dessous définit de façon schématique l'algorithme d'évaluation des spécifications. Les règles terminales sont les suivantes:

- Si l'entrée valide générée est correcte, on calcule l'état successeur (appel à la fonction *calculer_successeurs*).

- Si l'entrée valide générée est incorrecte, les spécifications ne sont pas satisfaites, donc la vérification s'achève.

La fonction ci-dessous est appelée par la fonction *generer_entree_valide* en lieu et place de la fonction *calculer_successeurs*.

fonction *evaluer_specifications* (q, e)
debut
si $\sigma(q, e) = 0$
alors
    retourner (Spécifications Non Satisfaite)
sinon
si $\sigma(q, e) = 1$
alors
    calculer_successeurs (q, e)
sinon
    debut
    soit j *tel que* $\sigma(q, e)$ *dépend de* e_j
    $e_j = 0$
    evaluer_specifications (q, e)
    $e_j = 1$
    evaluer_specifications (q, e)
    $e_j = \varphi$
    fin
fin

Ici encore, les variables d'entrée ne sont valuées "qu'à la demande". En pratique, la vérification consiste donc à générer récursivement les entrés valides pour chaque état accessible, puis à évaluer les spécifications et calculer l'état successeur pour chaque entrée valide.

L'utilisation des BDDs pour implémenter les algorithmes de vérification par la méthode énumérative permet donc d'optimiser les calculs liés à cette méthode, en évitant de valuer systématiquement toutes les entrées du modèle pour générer les entrées valides, évaluer les spécifications et calculer les états successeurs. D'autre part, en adoptant le même type de représentation pour la génération du modèle comme pour l'évaluation des assertions, on peut prendre en compte ces dernières et éviter ainsi le problème des assertions non causales incontournable dans l'implémentation classique. Cependant, les inconvénients liés à la représentation en extension des ensembles d'états par des vecteurs booléens persistent (représentation des ensembles linéaire). On peut alors essayer de pousser l'utilisation des BDDs encore plus loin en s'en servant pour représenter les ensembles d'états du modèle. Cette perspective est développée ci-dessous.

16.3 Représentation du modèle à l'aide des BDDs

Le modèle étant encodé dans le domaine booléen, les états sont dénotés par des monômes complets sur les variables d'état de celui-ci. Tout ensemble d'états correspond alors à une somme de

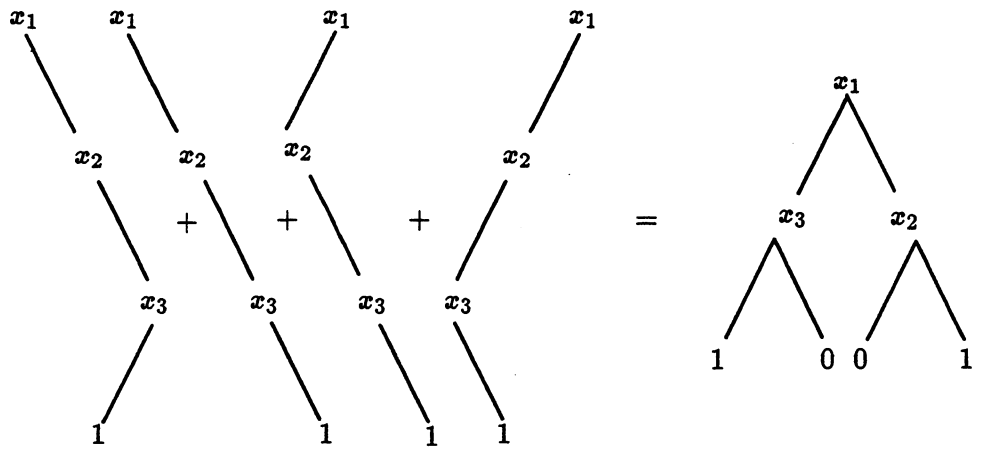


Figure 16.2: Représentation implicite de monômes complets dans un BDD

monômes, qui peut être représentée par un BDD. Ainsi, les ensembles A et V des états respectivement accessibles et visités peuvent être représentés par des BDDs. L'énorme avantage que procurent les BDDs dans ce cas est de permettre une implémentation de la méthode énumérative capable de traiter un nombre potentiellement *illimité* d'états. En effet, nous avons vu que la taille de la représentation d'un ensemble par un BDD n'est pas proportionnelle à sa taille réelle. A ce niveau, le gain enregistré par rapport aux implémentations classiques est considérable, car ces dernières sont toujours limitées techniquement au niveau du nombre d'états pouvant être traités.

Mais l'utilisation des BDDs engendre aussi de nouvelles optimisations au niveau des algorithmes de vérification développés précédemment, qui sont liées à l'exploitation de leurs propriétés sémantiques. Ces optimisations proviennent du simple fait que dans un BDD, toute somme de monômes complets est en général représentée *implicitement* par une somme de monômes *non complets*. Ainsi, dans l'exemple de la figure 16.2, la somme de quatre monômes complets aboutit à un BDD dénotant deux monômes non complets. Il est alors possible d'exploiter cette propriété pour définir un nouvel algorithme "pseudo-énumératif", dans lequel les états ne sont plus traités *un par un*, mais par *groupes* correspondant à des monômes non complets. Il suffit pour cela de modifier l'algorithme classique au niveau du retrait des états contenus dans l'ensemble A. Si A est dénoté par son BDD χ_A, le retrait d'un groupe d'états de A correspond à l'extraction d'un monôme dans χ_A. Cette opération est très simple à mettre en œuvre, comme cela a été démontré dans la section 15.2. De plus, il est possible de choisir le monôme à extraire de χ_A. Ainsi, le monôme sélectionné peut être celui qui dénote le plus grand nombre d'états contenus dans A, c'est à dire celui dont la longueur est *minimale*.

Le traitement simultané d'un ensemble d'états dénoté par un monôme non complet nécessite la modification des algorithmes de génération des entrées valides, d'évaluation des spécifications et de calcul des états successeurs définis dans la section précédente. En effet, dans les algorithmes de la section 16.2, seules les variables d'entrée sont valuées "à la demande". Dans les nouveaux algorithmes à définir, les variables d'état doivent aussi être valuées à la demande, puisque le monôme dénotant les états traités n'est en général pas complet. Il faut remarquer que cette

valuation des variables d'état est correcte car du point de vue sémantique, tous les monômes complets recouverts par un monôme non complet dans un BDD appartiennent implicitement à ce BDD.

La représentation de A et de V par des BDDs nécessite aussi d'implémenter l'ajout et le test d'appartenance d'états à ces ensembles, sous la forme d'opérateurs sur les BDDs. La première opération correspond à une somme booléenne entre une fonction et un monôme, tandis que la seconde correspond à une évaluation de fonction sur un monôme. Ces opérations sont simples et faciles à implémenter sur les BDDs, à l'aide des opérateurs décrits dans la section 15.2.

Toutefois, une dernière optimisation peut être apportée aux algorithmes de vérification. Celle-ci concerne les tests d'appartenance des états accédés lors du calcul des successeurs d'un groupe d'états. En effet, l'algorithme de calcul des états successeurs d'un état qui a été défini dans la section 16.2.1 fonctionne en calculant successivement la valeur de la première variable d'état, puis de la seconde, etc.... En disposant d'une représentation de A et V sous forme de BDD, on peut tester la présence de l'état généré dans ces ensembles au fur et à mesure que les valeurs des variables d'état sont déterminées.

La fonction définie ci-dessous schématise l'algorithme de calcul des états successeurs d'un ensemble d'états dénoté par un monôme non complet q. Elle intègre les deux modifications apportées à l'algorithme défini dans la section précédente:

- Les tests d'appartenance des états successeurs aux ensembles A et V s'effectuent au fur et à mesure de l'évaluation des variables d'état.
- Les variables d'états du monôme non complet q sont valuées "à la demande".

Les notations χ_A et χ_V servent à dénoter les fonctions caractéristiques des ensembles A et V.

fonction *calculer_successeurs* (q, e, i)
debut
si $i > n$
alors
    $q' = \vec{\delta}(q, e)$
sinon
si $\delta_i(q, e) = 0$ **ou** $\delta_i(q, e) = 1$
alors
    debut
    si $\chi_A(\delta_1(q, e), \ldots, \delta_i(q, e)) \neq 1$
    et $\chi_V(\delta_1(q, e), \ldots, \delta_i(q, e)) \neq 1$
    alors
        calculer_successeurs $(q, e, i + 1)$
    fin
sinon
si $\delta_i(q, e)$ *dépend de* q_j
alors
    debut
    $q_j = 0$
    calculer_successeurs (q, e, i)

$q_j = 1$
calculer_successeurs (q, e, i)
$q_j = \varphi$
fin
sinon
debut
soit k *tel que* $\delta_i(q, e)$ *dépend de* e_k
$e_k = 0$
calculer_successeurs (q, e, i)
$e_k = 1$
calculer_successeurs (q, e, i)
$e_k = \varphi$
fin
fin

Les fonctions de génération des entrées valides et d'évaluation des spécifications définies dans la section précédente sont modifiées de la même façon au niveau de la valuation des variables d'état.

16.4 Conclusions

La méthode qui vient d'être présentée a été implémentée sous la forme d'un nouvel outil de vérification. Elle s'avère être très économique en mémoire, comme cela était attendu. D'autre part, l'avantage déterminant lié à cette méthode se situe dans le fait qu'elle n'est théoriquement pas limitée au niveau du nombre d'états pouvant être traités. Elle est en outre actuellement en cours d'expérimentation.

D'autre part, l'interprétation des BDDs sous forme de sommes de monômes a aussi été utilisée pour implémenter une méthode de génération de modèle "réduit", dérivée de la méthode de génération de modèle minimal. La différence se situe au niveau du calcul des classes d'états équivalents: ce calcul est beaucoup plus économique que pour la méthode générant le modèle minimal (en particulier, toute les opérations de composition fonctionnelle se ramènent à des produits de fonctions), mais le modèle obtenu n'est en revanche plus minimal: il est seulement réduit. Cette méthode a des applications évidentes dans le domaine de la compilation. Néanmoins, elle pourrait permettre - si elle donne les résultats escomptés - la vérification de spécifications sur un modèle complet. Cette méthode est actuellement en cours d'expérimentation.

Enfin, il faut remarquer que les résultats des deux approches décrites ci-dessus dépendent beaucoup de l'ordre fixé pour les variables. Là encore, il serait très intéressant de déterminer quels sont les bons ordres, et pourquoi ils sont bons.

Conclusion

Bilan

L'objectif de cette étude était de définir et d'implémenter un outil de vérification formelle de programmes LUSTRE. Les travaux antérieurs réalisés dans ce domaine ont constitué la base du travail entrepris. En effet, ceux-ci ont permis de dégager les principes originaux de la vérification en LUSTRE (spécifications, description comportementale de l'environnement), mais aussi de mettre en évidence les problèmes intrinsèques à celle-ci (assertions non causales), ainsi que ceux liés aux méthodes de vérification classiques (explosion du modèle généré).

La première partie de l'étude a d'abord porté sur la définition d'une méthode de vérification formelle des programmes LUSTRE. La possibilité d'obtenir pour tout programme un modèle d'exécution à partir d'une sémantique formelle du langage nous a naturellement incité à envisager une méthode de vérification basée sur les modèles (*model checking*). Dans un premier temps, la sémantique des assertions et des spécifications d'un programme LUSTRE sur son modèle d'exécution a été définie formellement en termes d'opérateurs sur la structure du modèle. A partir de ces définitions, il fallait concevoir une méthode permettant l'évaluation effective des spécifications et des assertions sur le modèle. La conception de celle-ci a été guidée par deux impératifs: d'une part la capacité à mettre en œuvre les opérations nécessaires sur le modèle; d'autre part la résolution des problèmes rencontrés pour la vérification des programmes LUSTRE avec les méthodes précédemment développées. Une nouvelle méthode de vérification a donc été étudiée, basée sur un algorithme original de génération de modèles minimaux. En théorie, cette méthode permet de résoudre tous les problèmes rencontrés jusqu'à présent.

Pour son implémentation, la méthode fait appel aux BDDs, qui sont une technique de représentation et de manipulation symbolique des fonctions booléennes. Le modèle d'exécution d'un programme LUSTRE pouvant être encodé dans le domaine booléen, les BDDs ont été utilisés pour représenter et manipuler le modèle. Il faut noter à ce sujet que les BDDs constituent le *seul* moyen d'implémenter la méthode de vérification proposée. L'utilisation des BDDs nous a poussé à approfondir l'étude de leur fonctionnement afin d'abaisser la complexité de la vérification. En particulier, l'ordonnancement des variables, l'optimisation des opérateurs standards, la conception de nouveaux opérateurs et la combinaison "astucieuse" de ces opérateurs entre eux ont été largement explorés. Ces travaux ont abouti sur la réalisation d'une bibliothèque de BDDs optimisée, écrite en C++ [Str91], qui a été spécifiquement développée pour implémenter la méthode de vérification. Cette bibliothèque inclut de nombreuses fonctionnalités indispensables au fonctionnement des BDDs: gestion de la canonicité, réutilisation des résultats intermédiaires, récupération des nœuds inutilisés (*garbage collection*). Elle est d'ailleurs utilisée au sein du groupe SPECTRE pour d'autres applications de vérification [JCF,SB92,FM92].

Les essais de vérification de programmes LUSTRE réalisés sur un jeu de programmes issus d'applications réelles démontrent l'efficacité de la méthode développée, au moins par rapport à la méthode énumérative classique. De plus, on constate un gain intéressant en temps de calcul et en mémoire pour l'implémentation optimisée de la méthode.

D'autre part, l'utilisation intensive des BDDs au sein de l'outil a permis d'en révéler les qualités et aussi les défauts. Au chapitre des qualités, citons leur facilité et leur souplesse d'utilisation, ainsi que leur aptitude à réaliser efficacement des opérations complexes sur les fonctions booléennes. Les problèmes de représentation compacte des fonctions s'avèrent assez rares en pratique, ce qui plaide en faveur de leur utilisation. Par contre, nous avons tout d'abord pu constater l'extrême sensibilité des BDDs à l'ordre fixé pour les variables dans la méthode de vérification étudiée. On a aussi pu s'apercevoir que les BDDs sont quelquefois très coûteux en mémoire, particulièrement lors de certaines opérations. Ces constatations nous ont donc amené à développer une nouvelle approche de la vérification, toujours basée sur les BDDs pour la représentation du modèle d'exécution, mais utilisant uniquement des opérateurs simplifiés et économiques en mémoire pour manipuler celui-ci. Cette approche nous a permis d'obtenir une implémentation originale de la méthode de vérification énumérative en avant, dans laquelle les problèmes rencontrés jusqu'à présent pour la vérification des programmes LUSTRE (prise en compte des assertions, explosion du modèle) sont potentiellement résolus.

En ce qui concerne l'aspect utilisation de l'outil de vérification, il est apparu que la vérification nécessite d'importants efforts pour être mise en œuvre de façon adéquate. Plus précisément, deux problèmes sont apparus au cours des expérimentations réalisées. Le premier concerne l'environnement, dont il est indispensable de connaître au moins partiellement une description du comportement. Or, cette tâche s'avère ardue, et cela pour deux raisons:

- Tout d'abord, la description comportementale de l'environnement est intrinsèquement délicate: si celle-ci est "trop vague" (pas assez restrictive par rapport à la réalité), on risque de ne jamais réussir à vérifier les spécifications sur le programme, même si celui-ci est correct. Au contraire, si elle est trop restrictive (certains comportements possibles sont arbitrairement exclus), les spécifications vont être vérifiées pour des conditions ne correspondant à la réalité, ce qui est le pire des cas car on peut prouver comme cela qu'un programme faux est correct.

- D'autre part, il est généralement impossible d'exprimer n'importe quelle caractéristique du comportement de l'environnement sous forme d'invariants. Le mécanisme des assertions semble donc parfois offrir une puissance d'expression limitée par rapport à ce qu'il est nécessaire d'exprimer.

Le second problème apparu lors de l'utilisation de l'outil de vérification concerne la détection des erreurs en cas d'échec de la vérification. A ce niveau, une fonctionnalité de diagnostic complémentaire de la vérification a été implémentée. Ce diagnostic permet de mettre en évidence un comportement incorrect du programme vérifié. Cependant, la nécessité de disposer d'un outil de diagnostic plus puissant s'est faite ressentir.

Perspectives

En perspective, nous faisons un tour d'horizon des extensions possibles du travail réalisé. Au niveau de l'outil de vérification des programmes LUSTRE, plusieurs points nécessitent encore d'être développés ou approfondis.

Du point de vue technique, des améliorations doivent pouvoir encore être apportées au niveau de l'implémentation de la méthode de vérification, essentiellement au niveau des BDDs: recherche d'ordres de variables performants, gestion optimisée de l'espace mémoire occupé, optimisation des opérateurs.

En ce qui concerne l'utilisation effective de l'outil, il est dans l'immédiat indispensable d'évaluer plus en détail les limites de la nouvelle méthode proposée. Pour cela, il faudrait absolument comparer les performances de l'outil réalisé aux autres outils de vérification existants, par rapport à un jeu de tests communs (*benchmarks*). Dans la même optique, il serait très intéressant de mesurer les capacités effectives de la méthode de vérification sur une application industrielle de taille réaliste écrite en LUSTRE. Ce dernier point va être incessamment étudié par C. Dubois à Merlin Gerin.

A moyen terme, les interfaces des outils réalisés pourraient être améliorées de manière à rendre leur utilisation plus aisée:

- *Interface avec l'outil de vérification:* Actuellement, pour des raisons d'interfacage rapide avec les outils de compilation nécessaires à la génération du modèle d'exécution des programmes, l'utilisateur doit réaliser "à la main" un programme de vérification. Cette phase fastidieuse pourrait être évitée en développant une interface entre l'outil de vérification et l'utilisateur. Ainsi, pour un programme donné, ce dernier pourrait entrer les spécifications et les assertions du programme de manière interactive, puis faire effectuer à sa guise la vérification de telle ou telle spécification.

- *Interface avec l'outil de diagnostic:* L'outil de diagnostic actuel souffre de l'impossibilité de traduire les séquences diagnostiques directement en LUSTRE. Cela provient du fait que tout programme de vérification est d'abord expansé et traduit dans le format intermédiaire EC avant d'être vérifié. Pour remonter au langage source, il faudrait que le programme au format EC contienne des informations (*pragmas*) sur l'origine des instructions qui le composent. Cette fonctionnalité n'est pas encore réalisée à l'heure actuelle.

Enfin, à long terme, le développement d'un environnement complet de vérification des programmes LUSTRE s'avère nécessaire. En particulier, deux fonctionnalités de cet environnement semblent devoir être approfondies:

- Le développement d'un outil de diagnostic *interactif* comparable à un classique outil de débogage de programmes, offrant à l'utilisateur une aide pour rechercher interactivement les erreurs dans ses programmes. La puissance des BDDs devrait permettre la réalisation d'un outil réellement efficace.

- La description comportementale de l'environnement des programmes. Cet aspect est d'ordre plus méthodologique que le précédent. Une étude de cas similaire à celle menée au niveau des spécifications devrait être envisagée pour déterminer le genre des propriétés

qu'il est nécessaire de connaitre à propos du comportement de l'environnement, et comment il est possible de les exprimer. Ce travail pourrait éventuellement conduire à la définition d'un formalisme moins restrictif que les assertions pour décrire le comportement de l'environnement. A ce sujet, il faut remarquer que le problème de la description comportementale de l'environnement ne semble pas encore avoir été traité de manière approfondie. Cela s'explique facilement par rapport à la nature des systèmes traités jusqu'à présent. Dans le domaine de la vérification de circuits, l'environnement se comporte généralement de manière aléatoire. Quant à la vérification de protocoles, les systèmes forment souvent un groupe de processus communiquant entre eux, sans grandes interactions avec leur environnement.

Au delà de la vérification de programmes LUSTRE, la méthode de vérification qui a été développée peut être adaptée plus généralement à la vérification de n'importe quelle machine d'états finis. De plus, son principe même en fait une méthode plus puissante que les méthodes préexistantes, car elle permet l'évaluation de certaines propriétés de *vivacité* (Le calcul des assertions en LUSTRE correspond à l'évaluation de ce genre de propriété).

En conclusion, la méthode de vérification que nous avons proposée est plus puissante que les méthodes préexistantes. A ce titre, elle pourrait être considérée comme meilleure que les autres. Cependant, l'expérience montre que toutes ces méthodes sont complémentaires les unes des autres, c'est-à-dire que pour chaque cas traité, il y en a généralement au moins une qui fonctionne mieux que les autres. C'est pourquoi la méthode proposée ne prétend pas surclasser les méthodes existantes, mais elle constitue une chance supplémentaire de réussir à vérifier des systèmes dont la complexité est de toute manière exponentielle!

Bibliographie

[Ake78] S. B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, C-27(6), 1978.

[ASU86] A. Aho, R. Sethi, and J. Ullman. *Compilers : Principles, Techniques and Tools.* Addison-Wesley, 1986.

[BA85] F. B. Schneider B. Alpern. Defining liveness. In *Information Processing Letters 21*, pages 181–185, North-Holland, October 1985.

[BB91] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, September 1991.

[BCG87] G. Berry, P. Couronné, and G. Gonthier. Programmation synchrone des systèmes réactifs , le langage ESTEREL. *Technique et Science Informatique*, 4:305–316, 1987.

[BCG88] G. Berry, P. Couronné, and G. Gonthier. Synchronous programming of reactive systems, an introduction to ESTEREL. In K. Fuchi and M. Nivat, editors, *Programming of Future Generation Computers*, pages 35–55, Elsevier Science Publisher B.V. (North Holland), 1988. INRIA Report 647.

[Ben89] A. Benveniste. *Les langages synchrones: des noyaux logiciels pour la spécification et la conception des systèmes de contrôle commande.* Rapport "Collaboration CAO Automatique" n° 1, février 1989.

[Ber89a] C. L. Berman. *Circuit Width, Register Allocation and Reduced Function Graphs.* Technical Report RC 14129, IBM, 1989.

[Ber89b] C. L. Berman. Ordered binary decision diagrams and circuit structure. In *Proc. of ICCD'89*, September 1989.

[Ber89c] G. Berry. Real time programming: special purpose or general purpose languages. In *IFIP World Computer Congress, San Francisco*, 1989.

[BFH90a] A. Bouajjani, J. C. Fernandez, and N. Halbwachs. Minimal model generation. In *Workshop on Computer-Aided Verification*, June 1990.

[BFH90b] A. Bouajjani, J. C. Fernandez, and N. Halbwachs. *On the verification of safety properties.* Technical Report SPECTRE L12, IMAG, Grenoble, March 1990.

[BFH*92] A. Bouajjani, J. C. Fernandez, N. Halbwachs, P. Raymond, and C. Ratel. Minimal state graph generation. *To appear in Science of Computer Programming*, 1992.

[BG88] G. Berry and G. Gonthier. *The synchronous programming language* ESTEREL*: design, semantics, implementation.* Technical Report 842, INRIA, 1988. To appear in Science of Computer Programming.

[Bil87] J. P. Billon. *Perfect Normal Forms for Discrete Programs.* Technical Report 87039, BULL, September 1987.

[BL90] A. R. Brayton B. Lin, H. J. Touati. Don't care minimization of multi-level sequential logic networks. In *Proc. of ICCAD'90*, November 1990.

[Bry86] R. E. Bryant. Graph-based algorithms for boolean functions manipulation. *IEEE Transactions on Computers*, C-35(8):667–692, August 1986.

[Bry88] R. E. Bryant. *On the Complexity ogf VLSI Implementations and Graph Representations of Boolean Functions with Application to Integer Multiplication.* Technical Report, Carnegie Mellon University, September 1988.

[CB90] J.C. Madre C. Berthet, O. Coudert. New ideas on symbolic manipulations of finite state machines. In *Proc. of ICCD'90*, September 1990.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symposium on Principles of Programming Languages*, Los Angeles, January 1977.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *TOPLAS*, 8(2), 1986.

[Cou91] O. Coudert. *SIAM: Une Boîte à Outils pour la Preuve Formelle de Systèmes Séquentiels.* PhD thesis, Ecole Nationale Supérieure des Télécommunications, Paris, France, octobre 1991.

[CPHP87] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. LUSTRE: a declarative language for programming synchronous systems. In *14th ACM Symposium on Principles of Programming Languages*, January 1987.

[DER90] R. Kapur M. R. Mercer D. E. Ross, K. M. Butler. Fast functionnal evaluation of candidate obdd variable orderings. In *Proc. of The 2nd EDAC*, pages 4–10, February 1990.

[EE91] D. Taubner E. Enders, T. Filkorn. Generating bdds for symbolic model checking in ccs. In *Proc. of CAV'91*, pages 203–213, July 1991.

[EJ90] E. Laurent E. Jolly. Méthode de validation de circuit. application à une interface de bus. Projet 3ème Année ENSERG, 1990. Grenoble, France.

[Fer90] J. C. Fernandez. An implementation of an efficient algorithm for bisimulation equivalence. *Science of Computer Programming*, 13(2-3), May 1990.

[FM91] J.C. Fernandez and L. Mounier. On the fly verification of behavioural equivalences and preorders. In *Workshop on Computer-Aided Verification, Aalborg (Denmark)*, June 1991.

[FM92] M. Vachon F. Maraninchi. An experience in compiling a mixed imperative/declarative language for reactive systems. In *Proc. of The International Workshop on Computer Construction*, October 1992. To appear in LNCS, Springer-Verlag.

[Ger] Merlin Gerin. *Manuel d'Assurance Qualité.*

[Glo89] A-C. Glory. *Vérification de propriétés de programmes flots de données synchrones.* Thèse, Université Joseph Fourier, Grenoble, décembre 1989.

[GS90] Hubert Garavel and Joseph Sifakis. Compilation and verification of lotos specifications. In L. Logrippo, R. L. Probert, and H. Ural, editors, *Proceedings of the 10th IFIP International Symposium on Protocol Specification, Testing and Verification (Ottawa)*, North Holland, June 1990.

[GT92] B. Berkane G. Thuau. Using the language lustre for sequential circuit verification. In *Proc. of The International Workshop on Designing Correct Circuits*, Lingby, Denmark, January 1992.

[HCRP91a] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. Programmation et vérification des systèmes réactifs à l'aide du langage flot de données synchrone LUSTRE. *Technique et Science Informatique*, 10(2), 1991.

[HCRP91b] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[HJT90] B. Lin R. K. Brayton H. J. Touati, H. Savoj. Implicit state enumeration of finite state machines using bdd's. In *Proc. of ICCAD'90*, Santa Clara, CA, November 1990.

[HL90] N. Halbwachs and F. Lagnier. *An experience in proving regular networks of processes by modular model checking.* Technical Report SPECTRE L13 (to appear in Acta Informatica), IMAG, Grenoble, March 1990.

[Hol87] G. J. Holzmann. On limits and possibilities of automated protocols analysis. In *IFIP WG-6.1 7th. International Conference on Protocol Specification, Testing and Verification*, North Holland, 1987.

[HP85] D. Harel and A. Pnueli. On the development of reactive systems. In *Logic and Models of Concurrent Systems*, NATO *Advanced Study Institute on Logics and Models for Verification and Specification of Concurrent Systems*, Springer Verlag, 1985.

[HRR91] N. Halbwachs, P. Raymond, and C. Ratel. Generating efficient code from dataflow programs. In *the Third International Symposium on Programming Language Implementation and Logic Programming*, Passau, West Germany, August 1991.

[HS91] H. J. Touati H. Savoj, R. K. brayton. Extracting local don't cares for network optimization. In *Proc. of ICCAD'91*, November 1991.

[JCF] L. Mounier J. C. Fernandez, A. Kerbrat. Symbolic equivalence checking. Soumis à Forte 92.

[JJ89] C. Jard and Th. Jéron. On-line model checking for finite linear temporal logic specifications. In *International Workshop on Automatic Verification Methods for Finite State Systems, LNCS 407*, Springer Verlag, 1989.

[JLB88] E. Pilaud J. L. Bergerand. Saga: a software development environment for dependability automatic control. In *Proc. of SAFECOMP'88, IFAC*, Fulda, West Germany, 1988.

[JPB88] J. C. Madre J. P. Billon. Proving curcuit correctness using formal comparison between expected and extracted behaviour. In *Proc. of The 25th DAC*, July 1988.

[Kah74] G. Kahn. The semantics of a simple language for parallel programming. In *IFIP 74*, North Holland, 1974.

[KLM91] J. Schwalbe K. L. McMillan. Formal verification of the encore gigamax cache. *International Symposium on Shared Memory Multiprocessors*, 1991.

[Koz83] D. Kozen. Results on the propositional μ-calculus. *Transaction on Computer Science*, 27, 1983.

[KSB90] R. E. Bryant K. S. Brace, R. L. Rudell. Efficient implementation of a bdd package. In *Proc. of The 27th Design Automation Conference*, pages 40–45, Orlando, FL, June 1990.

[LGLL91] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire. Programming real-time applications with signal. *Proceedings of the IEEE*, 79(9), September 1991.

[LPZ85] O. Lichtenstein, A. Pnueli, and L. Zuck. The glory of the past. In *Conference on Logics of Programs, LNCS 194*, Springer Verlag, 1985.

[Mad90] J.C. Madre. *PRIAM, Un Outil de Preuve Formelle de Circuits Digitaux.* PhD thesis, Ecole Nationale Supérieure des Télécommunications, Paris, France, juin 1990.

[Mar90] F. Maraninchi. *Argos, un langage graphique pour la conception, la description et la validation des systèmes réactifs.* Thèse, Université Joseph Fourier, Grenoble, 1990.

[McG89] R. McGeer. *On the Interaction of Functionnal and Timing Behavior of Combinational Logic Circuits.* PhD thesis, U.C. Berkeley, November 1989.

[MF88] N. Kawato M. Fujita, H. Fujisawa. Evaluation and improvements of boolean comparison method based on binary decision diagrams. In *Proc. of ICCAD'88*, November 1988.

[Mil83] R. Milner. Calculi for synchrony and asynchrony. *TCS*, 25(3), juillet 1983.

[MP88] Z. Manna and A. Pnueli. The anchored version of the temporal framework. In W.P. de Roever J.W. de Bakker and G. Rozenberg, editors, *Linear Time, Branching Time and Partial Orders in Logics and Models for Concurrency, LNCS 354*, Springer Verlag, June 1988.

[NI91] S. Yajima N. Ishiura, H. Sawada. Minimization of binary decision diagrams based on exchanges of variables. In *Proc. of ICCAD'91*, Santa Clara, CA, November 1991.

[OC89a] J. C. Madre O. Coudert, C. Berthet. Verification of sequential machines using boolean functionnal vectors. In L. J. M. Claesen, editor, *Formal VLSI Correctness Verification*, pages 179–196, North-Holland, November 1989.

[OC89b] J. C. Madre O. Coudert, C. Berthet. Verification of synchronous sequential machines based on symbolic execution. In J. Sifakis, editor, *Lecture Notes in Computer Science: Automatic Verification Methods for Finite State Systems*, pages 365–373, Springer-Verlag, 1989.

[PH87] J. A. Plaice and N. Halbwachs. LUSTRE-V2 *User's guide and reference manual.* Technical Report SPECTRE L2, IMAG, Grenoble, October 1987.

[Pla88] J. A. Plaice. *Sémantique et compilation de* LUSTRE*, un langage déclaratif synchrone.* Thèse, Institut National Polytechnique de Grenoble, 1988.

[QS82] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *International Symposium on Programming, LNCS 137*, Springer Verlag, avril 1982.

[Rat88] C. Ratel. *Etude de la conformité d'un programme* LUSTRE *et de ses specifications en logique temporelle arborescente.* Rapport de DEA, Institut National Polytechnique de Grenoble, juin 1988.

[Ray91] P. Raymond. *Compilation efficace d'un langage déclaratif synchrone : le générateur de code Lustre-V3.* Thèse Grenoble, France, novembre 1991.

[Roc92] F. Rocheteau. *Extension du langage Lustre et application à la conception de circuits: Le langage Lustre-V4 et le système Pollux.* Thèse, Institut National Polytechnique de Grenoble, juin 1992.

[Ros77] D, T. Ross. Structured analysis (sa): a language for communicating ideas. *IEEE Transactions on Software Engineering*, SE-3(1):710–713, January 1977.

[RRSV87] J. L. Richier, C. Rodriguez, J. Sifakis, and J. Voiron. XESAR *user's guide.* Technical Report, IMAG, Grenoble, September 1987.

[SB90] K. L. McMillan D. L. Dill J. Hwang S. Burch, E. M. Clarke. Symbolic model checking: 10^{20} states and beyond. In *LICS*, 1990.

[SB92] C. Loiseaux J. Sifakis S. Bensalem, A. Bouajjani. Property preserving simulations. In *Proc. of CAV'92*, Montreal, Canada, June 1992.

[Sha38] C. E. Shannon. A symbolic analysis of relay and switching circuits. *Transactions AIEE*, 57:305–316, 1938.

[Sif82] J. Sifakis. A unified approach for studying the properties of transition systems. *TCS*, 3, 1982.

[SJF90] K. J. Supowit S. J. Friedman. Finding the optimal variable ordering for binary decision diagrams. *IEEE Transactions on Computer*, C-39(5):710–713, May 1990.

[SM88] R. K. Brayton A. Sangiovanni-Vincetelli S. Malik, A. R. Wang. Logic verification using binary decision diagrams in a logic synthesis environment. In *Proc. of ICCAD'88*, pages 6–9, November 1988.

[SM90] S. Yajima S. Minato, N. Ishiura. Shared binary decision diagrams with attributed edges for efficient boolean function manipulation. In *Proc. of The 27th Design Automation Conference*, pages 52–57, Las Vegas, NA, June 1990.

[Sto36] M. Stone. The theory of representations for boolean algebra. *Transactions AMS*, 40:37–111, 1936.

[Str91] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1991. 2^{nd} Edition.