Exploration implicite et explicite de l'espace d'´etats atteignables de circuits logiques Esterel

Yannis Bres

Université de Nice – Sophia Antipolis — UFR Sciences

Ecole Doctorale Sciences et Technologies de l'Information et de la Communication

THÈSE

Présentée pour obtenir le titre de

Docteur en Sciences
de l'Université de Nice – Sophia Antipolis

Spécialité Informatique

Préparée au Centre de Mathématiques Appliquées
de l'Ecole des Mines de Paris

par

Yannis BRES

Exploration implicite et explicite de l'espace d'états atteignables de circuits logiques Esterel

Directeur de thèse : Gérard BERRY

Soutenue publiquement le 12 décembre 2002
à l'Institut National de la Recherche en Informatique
et en Automatique de Sophia Antipolis

devant le jury composé de :

MM.	Charles André	Président	UNSA
	Gianpiero Cabodi	Rapporteur	Politecnico di Torino
	Philippe Schnoebelen	Rapporteur	CNRS / ENS Cachan
	Fernand Boéri	Examinateur	UNSA
	Pascal Raymond	Examinateur	CNRS / VERIMAG
	Gérard Berry	Directeur de thèse	Esterel Technologies

Table des matières

1 Introduction **13**

2 Cadre Technique **19**

2.1 Le langage `Esterel` . . . 19
2.2 Automates d'états finis . . . 21
2.3 Circuits logiques et sémantique constructive . . . 23
2.3.1 Circuits logiques . . . 23
2.3.2 La sémantique constructive du langage `Esterel` . . . 25
2.3.3 Les circuits `Esterel` . . . 25
2.3.4 Circuits cycliques . . . 26
2.3.5 L'arbre de sélection des programmes `Esterel` . . . 28
2.4 La chaîne de compilation `Esterel` v5_21 . . . 28
2.5 La chaîne de compilation `Esterel` v5_9x . . . 31
2.6 Autour d'`Esterel` . . . 32
2.6.1 `ECL` . . . 32
2.6.2 Le formalisme graphique `SyncCharts` . . . 34
2.6.3 L'environnement de développement `Esterel Studio` . . . 35
2.7 Machines d'états finis . . . 35
2.8 Diagrammes de décisions binaires (BDDs) . . . 36
2.9 Calcul implicite de l'espace d'états atteignables d'une FSM . . . 38
2.9.1 Principes de base . . . 39
2.9.2 Algorithme de base . . . 40
2.9.3 Raffinements . . . 41
2.9.4 Classes d'équivalences de fonctions de registres . . . 42
2.9.5 Exemple . . . 43
2.9.6 Analyse de complexité . . . 43
2.10 Vérification formelle par observateurs . . . 44
2.10.1 Propriétés de sûreté . . . 44
2.10.2 Propriétés de vivacité . . . 45
2.10.3 Equivalence de machines . . . 45

3 Approche implicite **47**
3.1 L'existant : `Xeve` . . . 47
3.1.1 Améliorations triviales de `Xeve` . . . 48
Vérification incrémentale des propriétés . . . 49
Transitive Network Sweeping automatique . . . 49
3.2 L'outil de vérification formelle `evcl` . . . 50
3.3 Réduction du nombre de variables pour le calcul du RSS . . . 51
3.3.1 Remplacement des variables d'états par des entrées libres . . . 51
Exemple . . . 52
3.3.2 Abstraction des variables à l'aide d'une logique trivaluée . . . 53
Logique trivaluée . . . 53
Application aux calculs d'espaces d'états atteignables . . . 54
Exemple . . . 56
Discussion . . . 56
3.3.3 Implémentation et expérimentations . . . 57
3.3.4 Automatisation de l'abstraction . . . 58
Abstraction basée sur la profondeur des variables d'états . . . 58
Analyse des réfutations erronées . . . 60
3.3.5 Travaux connexes . . . 60
3.4 Utilisation des informations structurelles sur les modèles . . . 61
3.4.1 Approximation syntaxique de l'espace d'états accessibles . . . 61
3.4.2 Renforcement des relations entre variables remplacées par des entrées libres . . . 62
3.4.3 Borne supérieure d'approximation . . . 63
3.4.4 Stratégies d'ordonnancement des variables . . . 63
3.5 Représentation interne des circuits . . . 64
3.5.1 Construction directe de `Tgr_Network`s à partir de circuits au format `BLIF` . . . 65
3.5.2 Construction de `Tgr_Network`s à partir de circuits au format interne `sc` . . . 65
3.5.3 Représentation intermédiaire du circuit par un AIGraph . . . 66
3.6 Vérification formelle en boîte blanche ou noire . . . 68
3.7 *Bounded Model Checking* . . . 68
3.8 Ingénierie d'`evcl` et de la librarie `TiGeREnh` . . . 69
3.8.1 Fournitures d'information supplémentaires à des écouteurs . . . 69
3.8.2 Vérification (informelle) du vérifieur . . . 70
3.9 Génération de séquences de tests exhaustives . . . 71
3.10 Conclusion . . . 73

4 Approche explicite **75**
4.1 Le nouveau moteur d'évaluation explicite de circuits . . . 76
4.1.1 Portes logiques . . . 76
4.1.2 Algorithme de base . . . 76

4.1.3 Complexité de l'algorithme . . . 77
4.1.4 Réduction de l'explosion combinatoire . . . 78
Utilisation des relations entre les entrées du circuit . . . 78
Expansion des compteurs . . . 79
Partitionnement des entrées du circuit . . . 80
Pondération des entrées du circuit . . . 81
Constructivité faible . . . 82
Traitement spécifique des actions *reset* . . . 84
Analyse *a priori* des branches reconvergentes . . . 86
4.1.5 Optimisation des parties linéaires . . . 88
Mise en queue des tests à effectuer (branchement retardé) . . . 88
Préservation de l'état du circuit entre les branchements . . . 88
Mécanisme d'évaluation rapide des portes logiques . . . 90
4.1.6 Analyse des registres redondants . . . 91
4.2 Génération d'automates explicites . . . 92
4.2.1 Complexité en espace des automates . . . 92
4.2.2 Partage en mémoire des sous-arbres isomorphes . . . 93
4.2.3 Suppression *a posteriori* des branchements inutiles et branches reconvergentes . . . 94
4.2.4 Implémentation des tables d'adressage dispersé . . . 96
4.2.5 Gestionnaire de tas *ad hoc* . . . 97
4.3 Vérification formelle explicite . . . 100
4.3.1 Réduction de la consommation mémoire . . . 100
4.3.2 *Sweeping* structurel à la volée . . . 101
4.4 Génération de séquences de tests exhaustives . . . 101
4.5 Conclusion . . . 102

5 Approche hybride implicite/explicite 103
5.1 Algorithme de base . . . 104
5.1.1 Analyse des vecteurs de BDDs des registres . . . 105
5.1.2 Complexité de l'algorithme . . . 105
5.1.3 Optimisations . . . 107
Utilisation des relations entre les entrées du circuit . . . 107
Pondération des entrées du circuit . . . 107
Préservation de l'état du circuit entre les branchements . . . 108
Mécanisme d'évaluation rapide des portes logiques . . . 108
5.2 Génération d'automates . . . 108
5.2.1 Résultats expérimentaux . . . 108
5.3 Vérification formelle . . . 109
5.4 Génération de séquences de tests exhaustives . . . 110
5.4.1 Algorithmes . . . 110
Couverture d'états . . . 111
Couverture de transitions . . . 111

Couverture d'outputs . . . 111
5.4.2 Résultats expérimentaux . . . 111
5.5 Conclusion . . . 113

6 Expérimentations **115**
6.1 Description des modèles . . . 115
6.2 Machine de test . . . 116
6.3 Légendes . . . 117
6.4 Carburant . . . 118
6.4.1 Description . . . 118
6.4.2 Calcul d'espace d'états complet . . . 118
6.4.3 Vérification formelle . . . 118
6.4.4 Abstraction du modèle . . . 120
Propriétés 1, 2, 3 et 4 . . . 120
Propriété 1 . . . 121
Propriété 2 . . . 122
Propriété 3 . . . 122
Propriété 4 . . . 123
6.4.5 Conclusions . . . 124
6.5 `FWS` . . . 125
6.5.1 Description . . . 125
6.5.2 Calcul d'espace d'états complet . . . 125
6.5.3 Vérification formelle . . . 125
6.5.4 Abstraction du modèle . . . 126
Propriété 1 . . . 126
Propriété 2 . . . 126
Propriété 3_1 . . . 127
Propriété 3_2 . . . 128
Propriété 4_1 . . . 128
Propriété 4_2 . . . 129
Propriété 5 . . . 130
Propriété 6 . . . 130
Propriété 7 . . . 131
6.5.5 Conclusions . . . 131
6.6 `TI` . . . 139
6.6.1 Description . . . 139
6.6.2 Vérification formelle . . . 139
Approche implicite . . . 139
Utilisation de l'analyseur de satisfiabilité `Prover` . . . 141
Approche purement explicite et hybride implicite/explicite . . . 142
6.6.3 Génération d'automate . . . 142
6.6.4 Arbre à 10 commutateurs . . . 145
6.6.5 Conclusions . . . 146

6.7 Testbench 147

7 Conclusion **149**

A Outils connexes **153**
A.1 Autour de Lustre : Lesar, Lutess et Lurette 153
A.2 SPIN 153
A.3 Murφ 154

B Sémantique opérationnelle du langage Esterel **157**

Liste des algorithmes **159**

Liste des figures **161**

Liste des programmes Esterel **163**

Liste des tableaux **165**

Bibliographie **167**

Remerciements

Terminer une thèse de doctorat est l'aboutissement d'un long —très long— chemin, pavé d'une succession de faits déterminants, une conjonction de battements d'ailes de papillons qui, s'ils n'avaient pas battu au même moment, au même endroit...

Cet aboutissement est l'occasion rituelle de remercier tous ceux qui ont donc battu des ailes, volontairement ou non, délicatement ou avec insistance, mais de préférence à bon escient. Cet ouvrage leur est donc dédié.

Le plus simple est probablement de procéder par ordre chronologique... Par soucis de concision, nous prendrons comme origine des temps une date arbitraire mais ô combien marquante (enfin surtout pour la première personne remerciée) : celle qui m'a vu naître.

Que ma mère soit donc ici remerciée pour m'avoir donné la vie, etc., mais également pour tous les sacrifices qu'elle a faits pour moi, surtout ceux dont je ne me suis certainement même pas rendu compte. Dans un registre purement informatique, puisqu'il est beaucoup question de cela ici, qu'elle soit notamment remerciée pour m'avoir offert mon premier ordinateur (un super TO-7 8bits de feu cadencé à 1MHz avec lecteur de cassettes et `BASIC` en ROM !), puis le deuxième (un encore plus super Amstrad PC1512 encore plus de feu cadencé à 8MHz avec deux lecteurs de disquettes 5"¼ !). Ce n'était que le début d'une longue histoire...

Merci à Mr Dupuis pour m'avoir donné ma chance et confié, alors que je n'avais pas encore 18 ans, le développement d'un logiciel de la fiabilité duquel des vies dépendaient et, progressivement, pour m'avoir confié le développement d'un système de gestion qui en quelques années de travail en parallèle à mes études (enfin...) a fini par répondre pratiquement à tous les besoins internes des quatre sociétés du groupe qu'il dirigeait.

Merci à tous ces enseignants qui m'ont beaucoup appris, dont notamment :

- Olivier Lecarme, pour m'avoir fait découvrir la Sainte Algorithmique, que je n'avais qu'entraperçue dans mes explorations solitaires, et pour son aide fort précieuse en LaTeX (avec toutes mes excuses pour ne pas avoir du tout accroché à tes histoires de grammaires lalalalalère et je ne sais quoi) ;
- Emmanuel Kounalis pour la seconde couche d'algos si yolis : Braaaaaaaaavvooooo !
- Jean-Paul Roy, pour m'avoir fait découvrir la programmation fonctionnelle et pour le plaisir que j'ai eu à tenter de l'enseigner avec lui par la suite ;

– Françoise Baude, pour m'avoir fait découvrir les automates d'états finis... durant l'examen correspondant ;
– Manuel Serrano pour m'avoir fait découvrir la compilation et pour m'avoir présenté à Gérard Berry ;
– Gilles Menez enfin, qui a été l'ultime personne à me conseiller la voie de la thèse.

Merci à Gérard Berry pour avoir accepté d'encadrer mes recherches durant ces trois années.

Merci à l'équipe des Cadence Berkeley Labs, où j'ai passé quelques mois très riches en découvertes, notamment à Ellen Sentovich, Ken McMillan, Andreas Kuehlmann, Luciano Lavagno et Pablo Argon. Merci encore à Ellen pour avoir rendu ces découvertes possibles.

Merci à tous ceux qui contribuent à faire du rez-de-chaussée du bâtiment Fermat de l'INRIA un cadre de travail que j'aurai particulièrement apprécié : Frédéric Boussinot, Xavier Fornari, Dominique Micollier, Valérie Roy, Robert de Simone, Jean-Ferdinand Susini, Olivier Tardieu et Eric Vecchié (bosse ta thèse !). Merci à Nadia Maizi pour son support.

Merci à Esterel Technologies, notamment pour l'accès aux serveurs de calculs (surtout celui du CMA) et le coin de bureau, à Bruno Pagano et au groupe Core.

Enfin, merci à Charles André d'avoir accepté de présider mon jury de thèse, à Gianpiero Cabodi et Philippe Schnoebelen d'avoir accepté de la rapporter, et à Fernand Boéri et Pascal Raymond d'avoir accepté de l'examiner.

Pour avoir énormément appris dans les livres et pour avoir vu le travail que représentait déjà la rédaction d'une thèse, je tiens à remercier tous les écrivains qui ont fourni un travail encore largement supérieur.

Enfin, globalement, merci à toutes ces personnes mais aussi celles que j'ai oubliées et dont j'implore le pardon, pour toutes les discussions toujours très intéressantes que nous avons eu et, je l'espère, pour toutes celles que nous aurons à l'avenir.

A Sylvie,

pour tout ce que nous construisons

et tout ce que nous construirons

Chapitre 1

Introduction

Systèmes réactifs

Avec la banalisation des composants électroniques, circuits intégrés et programmes temps réel sont devenus pratiquement omniprésents. Ils font partie intégrante de notre quotidien et on les retrouve de plus en plus en assistance ou en remplacement de l'opérateur humain, souvent dans des situations où des vies humaines sont en jeu : systèmes d'assistance au freinage des voitures, pilotes automatiques d'avions, stimulateurs cardiaques, robots chirurgiens, contrôleurs de centrales nucléaires, etc.

L'approche réactive synchrone permet de modéliser et d'implémenter de manière élégante les circuits intégrés et les programmes temps réels, collectivement appelés systèmes réactifs. L'approche réactive synchrone est basée sur le modèle sémantique des machines d'états finis étendues par des actions de calculs et des dépendances de données. Ce modèle sémantique rigoureux permet en outre l'application de techniques de vérification formelle de ces systèmes.

Un système réactif réagit en permanence à son environnement : les signaux provenant de cet environnement sont analysés et conduisent à des changements de l'état interne du système ainsi qu'à l'émission de signaux diffusés dans l'environnement. L'exécution d'un système réactif est donc décomposée en réactions successives, dont la durée est généralement majorée par des contraintes fortes que le système se doit de respecter. L'approche synchrone simplifie le cadre théorique des systèmes réactifs en considérant que les réactions ont une durée nulle.

Nos travaux s'inscrivent dans le domaine des programmes réactifs synchrones `Esterel`, éventuellement générés à partir d'autres langages de haut-niveau comme les `SyncCharts` ou `ECL`. Nous ne travaillons pas directement sur ces programmes mais sur des circuits logiques qui en sont dérivés. Ces circuits sont des graphes éventuellement cycliques constitués de portes logiques n-aires (`et`, `ou`, etc.) et de registres (*latches*) mémorisant un bit de donnée d'un top d'horloge à un autre.

Exploration d'espaces d'états atteignables

De nombreuses analyses des systèmes réactifs nécessitent d'explorer leur espace d'états atteignables. En fonction des besoins, ces explorations devront se faire selon divers degrés de précision. Nous étudierons dans cette thèse la vérification formelle de propriétés du système, la génération de tests permettant de stimuler de manière exhaustive l'ensemble des comportements possibles du système et la génération d'automate explicite représentant le système.

Vérification formelle

Le *Model Checking* regroupe les techniques visant à vérifier formellement l'adéquation d'un modèle à des propriétés temporelles de sûreté ("quelque chose de mal ne peut jamais arriver") ou de vivacité ("quelque chose de notable arrivera tôt ou tard"). Une des méthodes de *Model Checking* consiste à calculer l'espace d'états atteignables du système à vérifier et de le confronter aux propriétés. L'intersection entre l'ensemble des états atteignables et l'ensemble des états dans lesquels les propriétés pourraient être violées doit être nulle.

Le *Model Checking* n'impose pas de construire une représentation particulièrement détaillée de l'espace d'états atteignables : seule la connaissance des états atteignables à chaque niveau de profondeur du modèle facilite la génération de contre-exemples.

Génération de tests

La génération de séquences de tests stimulant un système permet de contrôler la conformité de deux implémentations à différents niveaux d'abstraction. Encore de nos jours, la génération de ces séquences de tests n'est que parfois automatisée, et les séquences produites sont le plus souvent longues à exécuter et incomplètes. Ce manque d'exhaustivité a d'ailleurs été la cause de nombreuses validations erronées [CGH+93, CYF94, SD95a]. Pour générer des séquences de tests exhaustives, il est nécessaire d'explorer la totalité de l'espace d'états atteignables du système.

Le degré de précision concernant la structure de l'espace d'états atteignables dépend de l'objectif de couverture des tests. Par exemple, la couverture d'états peut se contenter de ne connaître que le graphe des états mais la couverture de transitions nécessite en plus de connaître la structure interne des différentes transitions possibles d'un état à un autre.

Génération d'automates

La génération d'automates permet d'obtenir une représentation complètement explicite du graphe de transitions du système. Ce graphe peut être utilisé à diverses fins : non seulement la représentation qu'il donne du système en permet une exécution extrêmement rapide, mais cette représentation facilite bon nombre d'analyses du sytème en rendant directement accessibles de nombreuses données le concernant.

La génération d'automate explicite se situe à l'extrême du spectre des degrés de précision concernant la structure de l'espace d'états atteignables : les automates explicites fournissent une représentation détaillée ("*micro-step*") de chacune des transitions possibles entre deux états.

Approches implicites et explicites

A leurs débuts, les techniques d'exploration d'espaces d'états atteignables reposaient sur une énumération explicite de ces états, traités individuellemment. Si les techniques explicites permettent une exploration très fine des espaces d'états, elles souffrent intrinsèquement d'importants risques d'explosion en temps lorsque la combinatoire des expressions est trop importante. De plus, le stockage des états atteignables du système sous une forme explicite (en extension) utilisant un bit par variable d'état est souvent très coûteuse en mémoire.

Apparues au début des années 90, les techniques implicites ont permis une nette réduction des coûts d'exploration de l'espace d'états atteignables d'un système. L'idée est d'utiliser des formules décrivant les ensembles en intention au lieu de les énumérer explicitement. Les formules sont exprimées par des diagrammes de décisions binaires (BDDs). Les représentations d'ensembles en intention par des BDDs sont souvent bien plus compactes que leur représentation explicite. Enfin, les opérations de base sur les ensembles sont implémentées par des algorithmes souvent très efficaces en pratique.

Pour autant, les techniques explicites ne sont pas devenues obsolètes et leurs apports demeurent. Par exemple, de nombreux outils de *Model Checking*, comme `SPIN` ou `Mur`φ, sont basés sur des techniques hybrides faisant converger les techniques implicites et explicites.

Nos apports

Cette thèse traite des méthodes implicites et explicites de l'exploration des états atteignables d'un système. Nos travaux visent à réduire les coûts de ces explorations, par exemple *via* des abstractions du système étudié. Nous utilisons les résultats de ces explorations à des fins de vérification formelle de propriétés de sûreté, de génération d'automates explicites ou de génération de séquences de tests exhaustives.

Le Chapitre 2 détaille le cadre technique de cette thèse. Nous introduisons le langage `Esterel`, les automates explicites et les circuits logiques ainsi que la chaîne de compilation `Esterel`, dans laquelle s'inscrivent les outils que nous avons développés. Nous présentons ensuite une brêve introduction aux diagrammes de décisions binaires (BDDs) ainsi que les principes de base du calcul d'espace d'états atteignables d'un système, notamment à l'aide de BDDs. Enfin, nous présentons les différents concepts relatifs à la vérification formelle de programmes.

Le Chapitre 3 présente les résultats de nos travaux dans le cadre des techniques implicites. Nous avons développé un outil de vérification formelle proposant diverses fonctionnalités visant à réduire le nombre de variables impliquées dans les calculs d'espaces d'états.

Les fonctionnalités proposées intègrent la technique usuelle de remplacement de variables d'états par des entrées libres. Nous proposons d'étendre cette technique par une nouvelle méthode d'abstraction de variables basée sur une logique trivaluée. Ces deux méthodes calculant des sur-approximations de l'espace d'états atteignables, nous proposons également différentes stratégies utilisant les informations à notre disposition concernant la structure du modèle afin de réduire la sur-approximation.

Nous proposons aussi de vérifier les modèles en boîte noire, autrement dit en gérant au niveau du vérifieur l'édition de lien du modèle à vérifier avec les observateurs contenant les propriétés, ce qui permet de s'affranchir de liaisons excessives et pénalisantes générées par le compilateur `Esterel` version 5.

Le Chapitre 4 présente les résultats de nos travaux dans le cadre des techniques explicites. Nous avons développé un nouveau moteur d'exploration de l'espace d'états atteignables d'un circuit. Ce moteur est basé sur la simulation de la propagation des informations dans dans ses portes. La stabilisation du circuit est atteinte par des branchements récursifs sur ses entrées.

L'approche explicite présente intrinsèquement un risque d'explosion exponentielle. Nous présentons différentes techniques exactes ou heuristiques permettant d'éviter autant que possible ce risque. Nous présentons aussi différents choix d'implémentation, aux conséquences plus linéaires sur les performances du moteur, mais aux apports néanmoins non négligeables.

Ce moteur a initialement été utilisé à des fins de génération d'automates explicites. Ce générateur d'automates est intégré au compilateur `Esterel` version 5 et commercialisé au sein de l'environnement de développement intégré `Esterel Studio`. Il remplace l'ancien générateur d'automates du compilateur `Esterel` version 4, qu'il a très tôt surpassé en performances de plusieurs ordres de grandeur.

Ce moteur a par la suite été utilisé à des fins de vérification formelle. Il a rendu possible ou notablement accéléré la vérification formelle de plusieurs modèles que les technologies implicites ou les analyseurs de satisfiabilité ne parvenaient pas ou peinaient à traiter.

Le Chapitre 5 présente les résultats de nos travaux sur la convergence des techniques implicites et explicites. Nous avons développé un moteur hybride implicite/explicite de l'espace d'états atteignables d'un programme. Ce moteur hybride reprend le principe d'une simulation de la propagation des informations à travers le circuit, mais cette fois-ci en propageant des BDDs, ce qui ne nécessite plus de branchements récursifs sur les entrées. Cela permet d'améliorer les performances du moteur purement explicite sur les modèles peu linéaires.

Ce moteur hybride est moins adapté à la génération d'automates, les contraintes d'ordonnancement des actions étant assez pénalisantes. Nous destinons donc plus ce moteur hybride à des fins de vérification formelle ou de génération de séquences de tests couvrantes. Sur ce dernier point, nous présentons un prototype d'outil permettant de générer des séquences de tests exhaustives selon divers objectifs de couverture. Ce prototype présente des performances largement supérieures à un outil comparable mais basé sur des techniques purement implicites, développé au sein d'Esterel Technologies.

Le Chapitre 6 présente les différents modèles utilisés pour les mesures expérimentales, et détaille quelques expérimentations significatives. La plupart de ces modèles proviennent de sources industrielles.

Enfin, le Chapitre 7 conclut cette thèse, l'Annexe A présente différents outils connexes aux nôtres et l'Annexe B détaille les règles de la sémantique opérationnelle du langage `Esterel`.

Chapitre 2

Cadre Technique

Ce chapitre présente le cadre technique de cette thèse.

La section 2.1 propose une brève description du langage `Esterel`. La section 2.2 présente les automates d'états finis, en lesquels les programmes `Esterel` étaient historiquement traduits. La section 2.3 présente les circuits logiques, désormais représentation pivot du compilateur `Esterel`, ainsi que la sémantique constructive du langage, sur laquelle s'appuie la traduction en circuits. La section 2.4 présente la chaîne de compilation du compilateur `Esterel` avant nos interventions. La section 2.5 présente l'intégration des outils que nous avons développés à cette chaîne de compilation.

La section 2.6 présente deux langages fortement liés à `Esterel` —`ECL` et le formalisme graphique `SyncCharts`— ainsi que l'environnement de développement intégré `Esterel Studio`, qui permet l'utilisation conjointe de ces trois formalismes.

La section 2.7 présente le modèle sémantique des machines d'états finis, sur lequel est basé l'approche synchrone réactive. La section 2.8 propose une introduction aux Diagrammes de Décisions Binaires (BDDs). La section 2.9 propose une introduction au calcul d'espace d'états atteignables de machines d'états finis par des méthodes implicites. Enfin, la section 2.10 présente le paradigme des observateurs synchrones, que nos outils mettent en œuvre pour la vérification formelle de propriétés de sûreté.

2.1 Le langage `Esterel`

Le langage `Esterel` a été conçu en 1982 par deux automaticiens, Jean-Paul Marmorat et Jean-Paul Rigault, au Centre de Mathématiques Appliquées de l'Ecole des Mines de Paris, à Sophia-Antipolis. Alors qu'ils devaient construire une mini-voiture automatique pour un concours organisé par un journal d'électronique, ces chercheurs ont été confrontés au manque d'expressivité des formalismes alors disponibles. Ceux-ci ne permettaient pas d'exprimer aisément les algorithmes de contrôles et ne facilitaient pas la gestion des contraintes temporelles propres à ce type d'applications. Le langage `Esterel` a donc été conçu avec pour objectif de combler ces manques tout en demeurant simple et intuitif.

Peu après, Gérard Berry a rejoint l'équipe et s'est tout d'abord attaché à donner une

sémantique rigoureuse au langage. Il a par la suite pris la tête de l'équipe et a dirigé l'évolution du langage et du compilateur, partiellement décrite dans ce chapitre.

`Esterel` est un langage
- *réactif* : l'exécution d'un programme `Esterel` correspond à une succession de réactions délimitées par des tops d'horloge (le temps est discrétisé), chaque réaction correspondant à une analyse de l'état interne du système et des signaux d'entrées présents ainsi qu'à l'emission en conséquence de signaux de sortie ;
- *synchrone* : les réactions sont considérées comme prenant un temps nul ; en pratique, les émissions de signaux de sorties se font dans la même réaction que la lecture des signaux d'entrée.

L'approche synchrone réactive est introduite dans [Hal98] et plus complètement décrite dans [Hal93].

Les communications entre différents modules se font par diffusion instantanée de signaux (*instantaneous broadcast*). Le langage `Esterel` est impératif et il comprend les instructions de base suivantes [Ber] :
- le délai unitaire : **`pause`**, la seule instruction qui délimite et sépare les réactions
- le séquencement : p `;` q
- la composition parallèle : p `||` q
- les boucles : **`loop`** p **`end`**
- l'émission de signal : **`emit`** `S`
- le test de présence ou d'absence de signal : **`present`** `S` **`then`** p **`else`** q **`end`**
- la déclaration de signaux locaux : **`signal`** `S` **`in`** p **`end`**
- la gestion d'exceptions : **`trap`** `T` **`in`** p **`end`** et **`exit`** `T`
- la suspension : **`suspend`** p **`when`** `S`
- la préemption forte : **`abort`** p **`when`** `S`
- l'instruction vide : **`nothing`**

A partir de ces instructions de base sont dérivées d'autres instructions de plus haut niveau, plus pratiques pour le programmeur (cf. Programmes 2.1). `Esterel` permet aussi la manipulation de données (variables et signaux valuées), ce dont nous ne traiterons pas.

Le Programme 2.2 présente un aperçu du langage `Esterel` [Ber]. Ce programme modélise une séance d'entraînement d'un coureur.

Chaque matin (lignes 12+34), le coureur fait `NumberOfLaps` tours de stade (ligne 14+30). A chaque tour (lignes 15+29), il commence par marcher (ligne 17) jusqu'à avoir parcouru 100 mètres (lignes 16+18). Puis il saute (ligne 22) à chaque pas (lignes 21+23) durant 15 secondes (lignes 20+24). Enfin, il court (ligne 25) jusqu'à la fin du tour (ligne 29). En parallèle (ligne 26) des phases intenses (lignes 20-25), le coureur vérifie son état cardiaque (ligne 27), ce qui pourrait lancer l'exception `HeartAttack` (**`exit`** `HeartAttack`) en cas de défaillance, terminerait alors immédiatement l'entraînement (lignes 13-31) et dirigerait le coureur vers l'hôpital (lignes 31-33)... jusqu'au lendemain (lignes 12+34), où l'entraînement recommence.

```
1 loop
2   pause
3 end
```
(a) `halt`

```
1 abort
2   halt
3 when I
```
(b) `await I`

```
1 loop
2   abort
3     p ; halt
4   when S
5 end
```
(c) `loop` p `each S`

```
1 await S ;
2 loop
3   abort
4     p ; halt
5   when S
6 end
```
(d) `every S do` p

```
1 every tick
2   emit I
3 end
```
(e) `sustain I`

Programme `Esterel` 2.1: Quelques instructions `Esterel` dérivées

Ce programme utilise des *signaux d'entrées* (*inputs*) pour démarrer/redémarrer (instruction **`every`**), préempter (instructions **`every`**, **`abort`**+**`when`** ou **`trap`**+**`exit`**), etc., des comportements. Ces comportements sont modélisés par l'émission de *signaux de sortie* (*outputs*), dont l'émission est ici maintenue à chaque instant par l'instruction **`sustain`**, jusqu'à ce que cette instruction soit préemptée.

2.2 Automates d'états finis

Dans les versions v1, v2 et v3 du compilateur, les programmes `Esterel` étaient directement traduits en automates d'états finis [BG92], par expansion puis applatissement des comportements. L'utilisation d'une représentation complètement explicite apporte notamment les avantages suivants :

- toutes les configurations d'états accessibles sont directement connues ;
- un automate explicite est très facilement interprétable ou traduisible en un langage générique (`ADA`, `C`, `Java`, etc.) ;
- l'évaluation, interprétée ou compilée, d'un automate explicite est extrêmement rapide ;
- jusqu'à une certaine taille, les automates explicites sont très lisibles pour un humain.

Malgré cela, les automates explicites présentent le double désavantage de nécessiter un temps de génération et un espace de stockage exponentiels dans la taille du code dans le pire des cas : le parallélisme, construction de base des langages synchrones, conduit à la duplication des sous-composantes parallélisées.

Cela s'illustre typiquement à l'aide d'un programme tel que `AB_O` (Programme 2.3). Ce programme attend la réception des signaux d'entrées `A` et `B` (dans le même instant ou non)

```
1  module Runner:
2
3    constant   NumberOfLaps: integer;
4
5    input      Morning, Second, Meter, Step, Lap;
6
7    relation   Morning => Second,
8               Lap => Meter;
9
10   output     Walk, Jump, Run;
11
12   every Morning do
13     trap HeartAttack in
14       repeat NumberOfLaps times
15         abort
16           abort
17             sustain Walk
18           when 100 Meter;
19           [
20                abort
21                  every Step do
22                    emit Jump
23                  end
24                when 15 Second;
25                sustain Run
26             ||
27                % monitor heart condition
28           ]
29         when Lap
30       end
31     handle HeartAttack do
32       % rush to hospital
33     end
34   end
35
36 end module
```

Programme Esterel 2.2: Runner

pour émettre le signal de sortie O.

L'automate correspondant est représenté par la Figure 2.1. Les transitions peuvent être étiquetées par un couple *condition/action*. Si cet automate demeure assez lisible, on notera que les tests de présence des signaux A et B ainsi que que l'émission du signal O sont répétés à plusieurs reprises, alors qu'il n'est jamais fait plus d'une fois mention de ces signaux dans le programme source.

```
1 module AB_O:
2
3   input   A, B;
4   output  O;
5
6   [ await A || await B ] ;
7   emit O
8
9 end module
```

Programme Esterel 2.3: AB_O

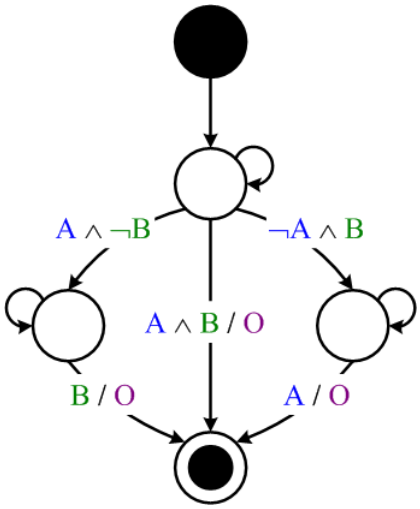


Figure 2.1: L'automate AB_O

Si l'on poursuit cet exemple en rajoutant l'attente d'un troisième signal C à la ligne 6, l'automate devrait être représenté en 3 dimensions et aurait alors la forme d'un cube unitaire, l'avancement sur l'un des axes correspondant à la réception d'un des trois signaux A, B ou C. Avec un quatrième signal, l'automate serait un hypercube de dimension 4, etc. : pour un nombre n de signaux attendus en parallèle, l'automate correspondant aura 2^n états (à l'état initial près).

Enfin, les automates ordonnancent en séquence toutes les actions à effectuer dans l'instant. Ils ont donc tendance à entrelacer des comportements initialement distincts et mis en parallèle dans un programme, ce qui en réduit encore la lisibilité.

2.3 Circuits logiques et sémantique constructive

2.3.1 Circuits logiques

Dans la version 4 du compilateur Esterel, les automates explicites ont été délaissés au profit des circuits logiques, à partir desquels pouvaient encore être générés des automates, mais de manière optionnelle.

A l'inverse des automates explicites, les circuits sont à la fois générables en un temps

linéaire et ont une taille linéaire par rapport aux programmes sources[1] : les circuits permettent donc de s'affranchir des limitations qu'imposent les automates et de pouvoir utiliser le parallélisme sans aucune contrainte. Avec les circuits logiques, l'ensemble des états atteignables est désormais implicite, à l'inverse des automates qui le spécifient explicitement. D'autres informations sont néanmoins plus aisément accessibles, comme par exemple les dépendances de données, qui sont éparpillées dans un automate.

La Figure 2.2 présente le circuit correspondant au programme précédent, tel qu'il pourrait être généré par le compilateur `Esterel`. Si de prime abord le circuit semble beaucoup moins lisible que l'automate, on constate déjà que les références aux signaux `A`, `B` ou `O` sont désormais uniques. Nous avons encadré par des pointillés rouges les parties de circuit correspondant aux boucles d'attente des signaux `A` et `B` : ces blocs sont identiques au signal testé près et sans redondance.

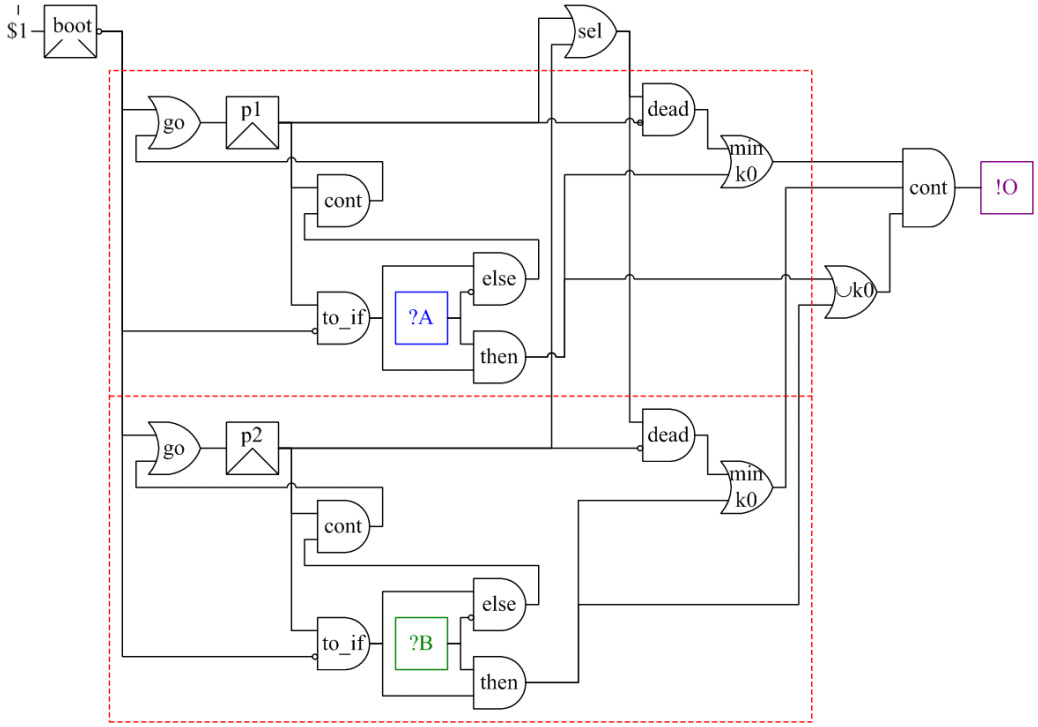


Figure 2.2: Le circuit `AB_O`

L'ajout d'attentes de signaux supplémentaires dans le programme source génèrera alors autant de blocs supplémentaires. Il en va de même pour les autres constructions du langage telles que les boucles, les préemptions, les exceptions, etc. : leur utilisation n'aura qu'un impact linéaire sur le circuit généré, alors qu'il pourrait avoir un impact exponentiel sur un automate.

[1] La schizophrénie de certains débuts (les surfaces) de boucles, soit la possibilité que plusieurs incarnations de ces débuts de boucles soient actives simultanément, implique de dupliquer certaines parties du programme [Ber99]. Cette duplication est généralement négligeable, et ne devient notable que sur des programmes de psychopathes.

2.3.2 La sémantique constructive du langage `Esterel`

A l'inverse d'un trop grand nombre de langages de programmation, le langage `Esterel` dispose d'une sémantique mathématique rigoureuse, la *sémantique constructive* [Ber98, Ber99]. Cette sémantique est constituée de règles logiques *SOS* (*Structural Operational Semantics*). Pour chaque instruction du langage, un ensemble de règles de réécriture décrit leur comportement pour un environnement spécifié. Les règles sont de la forme

$$p \xrightarrow[E]{E', k} p'$$

où

- p est l'instruction réécrite en p', l'instruction à exécuter à l'instant suivant (la dérivation) ;
- E est l'environnement d'entrée, soit l'ensemble des signaux présents ;
- E' est l'environnement résultant, soit l'ensemble des signaux émis par la règle ;
- k est le code numérique de complétion de l'instruction : 0 pour la terminaison, 1 pour la pause, $k+2$ pour un lancement d'exception (**`exit`**) de niveau k.

Par exemple, la règle sémantique associée à la parallélisation est la suivante :

$$\frac{p \xrightarrow[E]{E', k} p' \qquad q \xrightarrow[E]{E'', l} q'}{p||q \xrightarrow[E]{E' \cup E'' \,,\, \max(k,l)} p'||q'}$$

Les règles sémantiques correspondant aux instructions de base du langage sont indiquées en Annexe B (p. 157).

Enfin, la sémantique constructive du langage `Esterel` stipule qu'un circuit est *constructif* si, pour peu que l'on maintienne ses entrées électriquement stables durant un temps suffisant, toutes les portes qui le constituent se stabilisent à leur tour en un temps fini. Par construction, les circuits acycliques sont nécessairement constructifs. Les circuits cycliques nécessiteront un traitement spécifique et potentiellement très coûteux afin de vérifier leur constructivité.

2.3.3 Les circuits `Esterel`

La sémantique constructive du langage permet la traduction des programmes `Esterel` en circuits logiques :

- E devient l'ensemble des fils transportant le statut de présence des signaux d'entrée ;
- E' devient l'ensemble des fils transportant le statut de présence des signaux de sortie ($E \subseteq E'$) ;
- k devient l'ensemble des fils indiquant les codes de complétion ; ces fils activent à leur tour d'autres parties du circuit.

Une génération naïve de circuits consiste à générer récursivement, pour chaque instruction du langage et en respectant les règles de la sémantique constructive, un sous-circuit dont l'interface est présentée par la Figure 2.3, où :

- la porte *go* démarre l'instruction (à l'instant initial) ;
- la porte *res* poursuit l'exécution de l'instruction (aux instants ultérieurs) ;
- la porte *susp* suspend l'instruction ;
- la porte *kill* termine une instruction à la fin de l'instant ;
- la porte *sel* indique que l'instruction est sélectionnée (autrement dit, elle contient au moins une instruction active).

Les points d'arrêt du programme —les instructions **`pause`** explicites ou implicites— génèrent des registres permettant de mémoriser l'état du programme, soit les points d'arrêts actifs, d'un instant à l'autre.

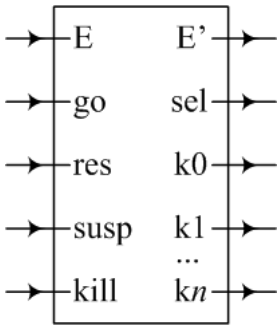


Figure 2.3: Bloc de base des circuits `Esterel`

2.3.4 Circuits cycliques

Certains programmes `Esterel` génèrent des circuits contenant des cycles dans leur logique combinatoire, autrement dit dans le même instant.

Le Programme 2.4 présente un tel cycle : dans le même instant, l'émission de `O2` est conditionnée par l'émission de `O1`, alors que l'émission de `O1` est elle-même conditionnée par l'émission de `O2`. Un tel programme est rejeté par le compilateur `Esterel` car il n'est pas *réactif* [Ber99] : les signaux de sortie ne peuvent être correctement déterminés puisqu'il existe deux solutions aux équations :

$$\texttt{O1} \wedge \texttt{O2} \qquad \text{et} \qquad \neg\texttt{O1} \wedge \neg\texttt{O2}$$

Le Programme 2.5 est aussi cyclique, mais la dépendance entre `O1` et `O2` est brisée par la présence ou l'absence du signal d'entrée `I`. Un tel programme est donc accepté par le compilateur `Esterel`.

Enfin, un exemple typique de circuit non constructif est donné par l'expression

$$X = X \vee \neg X$$

dont la seule solution logique est $X = 1$ selon la règle du tiers exclu. La sémantique constructive est basée sur les règles de propagation du courant électrique, qui ne connait rien au principe du tiers exclu. Ainsi, dans un modèle de circuits où l'on prend en compte les délais de propagation du courant imposés par les portes, il existe des délais envisageables pour lesquels ce circuit ne se stabilise pas. La Figure 2.4 présente un exemple de tels délais :

```
1  module BadCycle:
2
3    output  O1, O2;
4
5       present O1 then
6         emit O2
7       end
8    ||
9       present O2 then
10        emit O1
11      end
12
13 end module
```

Programme Esterel 2.4: BadCycle

```
1  module GoodCycle:
2
3    input   I;
4    output  O1, O2;
5
6    present I then
7      present O1 then
8        emit O2
9      end
10   else
11     present O2 then
12       emit O1
13     end
14   end
15
16 end module
```

Programme Esterel 2.5: GoodCycle

avec une valeur initiale des fils à 0, un délai unitaire pour les portes non et ou et un délai de 3 pour la porte identité, le circuit ne se stabilise pas et la valeur X oscille indéfiniment entre les valeurs 0 et 1.

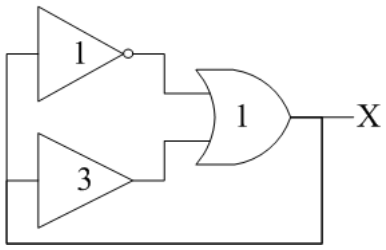


Figure 2.4: Le circuit “Hamlet”

L’analyse de la constructivité d’un programme fait appel à des techniques mathématiques à base de BDDs [SBT96], ce qui peut être extrêmement coûteux à la fois en temps et en mémoire, voire prohibitif sur certains types de programmes. De plus, les circuits décyclisés sont généralement notablement plus gros que les circuits initiaux. Néanmoins, les circuits cycliques permettent parfois de résoudre avec élégance certains problèmes (voir par exemple l’arbitre de bus de Robert de Simone, présenté en 4.1.4, p. 81). Eviter d’avoir à contrôler la constructivité d’un circuit par des méthodes coûteuse lorsque cela est possible est donc très souhaitable.

2.3.5 L'arbre de sélection des programmes `Esterel`

A chaque instruction d'un programme `Esterel` correspondent donc des portes (*go* et *res*) qui déterminent si une instruction doit être exécutée ou non. Les valeurs de ces portes sont calculées par combinaisons :

- de variables d'états, ou *registres de pause* ;
- de sélection ;
- de tests sur les signaux ou les variables.

Ces portes alimentent à leur tour les registres de pauses, qui détermineront les instructions actives à l'instant suivant.

La partie contrôle des programmes `Esterel` a une structure très hiérarchique, qui reflète l'emboîtement des constructions du programme initial[2]. Cette structure hiérarchique se retrouve à son tour dans l'*arbre de sélection*, arbre dont les feuilles sont les registres de pauses et les nœuds des portes `ou` [Ber99, PB01]. Ces portes `ou` indiquent également si les sous-arbres sont *compatibles* ou *exclusifs*, autrement dit s'ils peuvent s'exécuter en même temps (parallélisation) ou non (séquencement).

Le Programme 2.6 présente en partie droite l'arbre de sélection correspondant à ce programme. Les nœuds exclusifs sont symbolisés par des dièses. L'instruction **`await`** implique une pause implicite : une fois que le premier instant auquel une instruction **`await`** était activée est terminé, l'instruction suivante est activée dès que le signal attendu est reçu. De ce fait, chaque instruction **`await`** génère un registre de pause dans le circuit. Parce que les instructions **`await`** des lignes 6 et 8 sont exécutées en séquence, leurs registres sont *exclusifs*. De manière similaire, parce que le bloc des lignes 5 à 13 est exécuté en séquence avec l'instruction **`await`** de la ligne 14, les registres de pause générés par le bloc des lignes 5 à 13 sont *exclusifs* avec le registre de pause de la ligne 14. Inversement, les blocs des lignes 6 à 9 et 11 à 12 sont *compatibles* : aucune relation liant les registres de pause ne peut être inférée.

Les exclusions indiquées par l'arbre de sélection ont par exemple été utilisées pour diminuer le nombre de registres d'un circuit `Esterel` [STB97]. L'idée était, en remontée d'un parcours récursif de l'arbre de sélection, de factoriser les registres présents dans des sous-arbres exclusifs et donc réutilisables.

2.4 La chaîne de compilation `Esterel` v5_21

La Figure 2.5 présente un aperçu de la chaîne de compilation `Esterel` v5_21, soit avant nos interventions.

Les fichiers sources `Esterel` (`.strl`) sont transformés en code intermédiaire (`.ic`) après une passe d'analyses lexicale, syntaxique et sémantique. Le code intermédiaire est ensuite lié (`.lc`), puis compilé en circuits logiques (`.sc`).

[2] Alors que la partie communication, qui s'effectue par envoi de signaux (*broadcast* instantané), est plus transversale.

```
1  module SelectionTree:
2
3    input   I1, I2, I3;
4
5    [
6         await I1;          % pause 1 --- #
7         do something;      %              # --- /
8         await I2;          % pause 2 --- #     /
9         do something       %                   / --- #
10     ||                    %                   /     #
11        await I3;          % pause 3 --------- /     # ----
12        do something       %                         #
13   ];                      %                         #
14   await I4;               % pause 4 --------------- #
15   do something
16
17 end module
```

Programme Esterel 2.6: SelectionTree

Les circuits doivent ensuite être triés (.ssc). Les circuits acycliques ne nécessitent qu'un simple tri topologique effectué par le processeur scssc. Les circuits cycliques nécessitent que leur constructivité soit contrôlée avant d'être décyclisés, tâches réalisées par le processeur sccausal. Les circuits non constructifs sont rejetés et un des cycles est exhibé afin d'aider le développeur à corriger son programme.

Les circuits triés (.ssc) peuvent être convertis au format public BLIF [BLI98]. Les circuits BLIF sont optimisables par blifopt, un ensemble de scripts combinant des outils d'optimisation génériques provenant de SIS [SSL+92] et remlatch [STB96], un optimiseur développé par l'équipe Esterel. Les circuits BLIF optimisés peuvent alors être reconvertis en circuits .scc par blifssc. Les circuits BLIF peuvent être vérifiés formellement par Xeve [Bou97, Bou98] (cf. 3.1, p. 47). L'équivalence de deux circuits BLIF peut être vérifiée formellement par le processeur fsm_verify (non représenté sur la Figure 2.5).

Les circuits triés (.ssc) peuvent être compilés en automates explicites par le processeur sscoc (cf. 2.5, p. 31).

Les programmes Esterel peuvent être traduits en programmes C (les *backends* ne sont pas représentés sur la Figure 2.5) à partir :

- des circuits non triés (.sc), auquel cas le circuit est évalué par simulation de la propagation du courant électrique ;
- des circuits triés (.ssc), auquel cas les équations du circuit sont résolues linéairement ;
- des automates (.oc), auquel cas les arbres de décisions formant les transitions entre états sont parcourus en fonction des positionnement des entrées.

Ces trois implémentations du modèle théorique des machines d'états finis correspondent

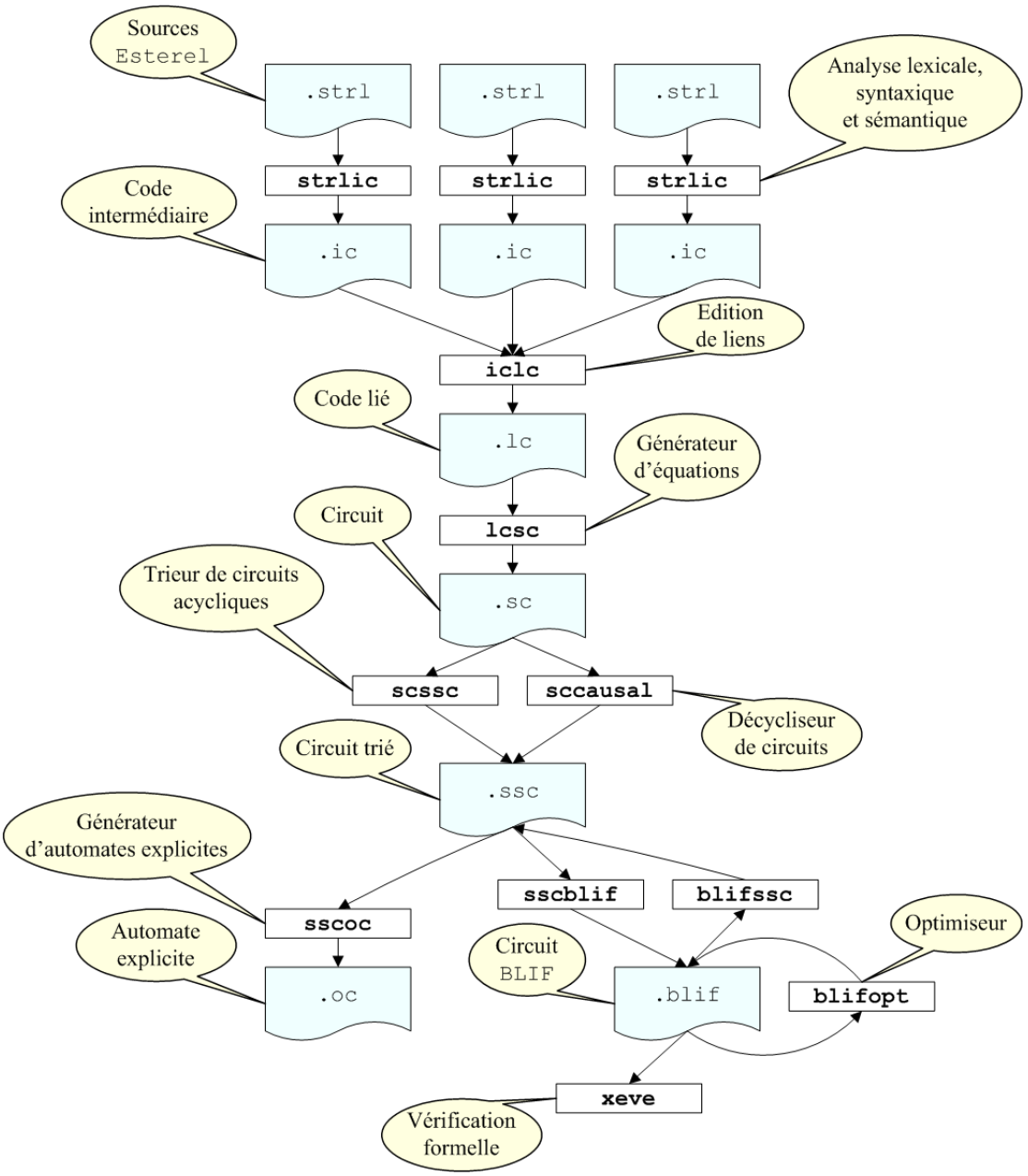


Figure 2.5: La chaîne de compilation d'`Esterel` v5_21

à des compromis taille/vitesse d'exécution différents : les circuits sont très compacts mais nécessitent d'évaluer la totalité des équations —qu'elles soient actives ou non— à chaque instant, alors que les automates offrent des transitions minimales mais une taille potentiellement exponentielle.

Il est aussi possible de traduire les programmes `Esterel` dans d'autres langages comme `Java` ou `ADA`, mais la compilation en `C` correspond au flot principal.

Les programmes `C` ainsi produits peuvent alors être compilés pour n'importe quelle plate-forme disposant du compilateur adéquat (machine standard Unix/Windows ou système embarqué, LegOS, etc.). Les programmes `C` peuvent également être liés avec la bibliothèque `libcsimul`, afin d'être simulés par le *debugger* graphique `xes`, ainsi que son évolution intégrée à `Esterel Studio`.

2.5 La chaîne de compilation `Esterel` v5_9x

Depuis la version 4 du compilateur `Esterel`, la génération d'automates explicites se faisait donc à partir des circuits, en deux passes :

1. Une première passe triait d'abord les équations composant le circuit : cela était fait par un simple tri topologique par le processeur `scssc` pour les circuits acycliques, ou par contrôle de la constructivité et décyclisation du circuit par le processeur `sccausal` [SBT96] pour les circuits cycliques.
2. Une seconde passe, le processeur `sscoc` [Mig, Mig94], générait alors l'automate en effectuant une exécution symbolique de toutes les transitions possibles jusqu'à saturer l'espace d'états atteignables.

L'outil `sccausal` fait appel à des BDDs pour calculer l'espace d'états accessibles des composantes cycliques du circuit, vérifier la constructivité et rendre des équations triées. Ce processus peut être extrêmement coûteux à la fois en temps et en mémoire, voire prohibitif sur certains types de programmes.

En sus, le processeur `sscoc`, peu évolutif parce que peu documenté, présentait de mauvaises performances sur des programmes assez simples : le couple `sccausal`+`sscoc` réduisait encore plus le champs d'application des automates explicites, déjà limité par leur caractère exponentiel dans le pire des cas. Le processeur `sscoc` considérait les circuits comme une liste triée d'équations booléennes. L'algorithme de résolution des équations était donc linéaire et procédait à la substitution des variables par les résultats d'équations précédemment résolues, l'automate étant dérivé des résultats de ces équations. Par nature, cet algorithme ne s'applique pas aux circuits cycliques.

En fait, travailler sur une liste d'équations triées, et donc décycliser au préalable les circuits cycliques n'est pas nécessaire : comme le stipule la sémantique constructive [Ber99] du langage `Esterel`, un circuit est *constructif* si, pour peu que l'on maintienne ses entrées électriquement stables durant un temps suffisant, toutes les portes qui le constituent se stabilisent à leur tour en un temps fini. Cette définition est indifférente à d'éventuels cycles.

Nous avons donc développé un nouveau moteur d'évaluation explicite de circuits, complètement conforme avec la sémantique constructive du langage. Ce moteur a tout d'abord été utilisé pour la génération d'automates explicites au sein du processeur `scoc`. Ce processeur travaille directement à partir de circuits `.sc` et remplace le processeur `sscoc` ainsi que la nécessité de trier le circuit au préalable (*via* `scssc` ou `sccausal`). Notre processeur `scoc` est intégré au compilateur `Esterel` depuis la version v5_91 et fait partie de l'environnement de développement intégré `Esterel Studio`, commercialisé par la société Esterel Technologies. Le nouveau moteur d'évaluation explicite de circuits a par la suite été étendu à des fins de vérification formelle ou de génération de séquences de tests exhaustives. Ce moteur explicite est présenté dans le Chapitre 4. Ce moteur a par la suite évolué en une version hybride implicite/explicite, décrite au Chapitre 5

En parallèle, nous avons développé un nouveau vérifieur formel basé sur des techniques implicites, `evcl`, décrit au Chapitre 3.

La Figure 2.6 présente la chaîne de compilation v5_9x, avec nos apports.

Outre nos travaux, un nouvel outil, `scsimplify`, développé par Dumitru Potop-Butucaru, permet de simplifier la partie contrôle des circuits `.sc` [PB01]. Cet outil n'est pas distribué avec le compilateur `Esterel` v5_9x. La plupart des améliorations qu'il propose ont été ou seront intégrées au générateur de circuits d'`Esterel` v7.

2.6 Autour d'`Esterel`

`Esterel` est utilisé comme langage cible d'un certain nombre d'outils ou de formalismes réactifs synchrones. Nous noterons particulièrement le langage `ECL`, un `C` augmenté de constructions réactives synchrones, ainsi que le formalisme graphique `SyncCharts` et sa version industrielle `Esterel Studio`.

2.6.1 `ECL`

Avant la version v7 du compilateur `Esterel`, la séparation contrôle/données était très nette : la partie contrôle d'un programme était rédigée en `Esterel` alors que la partie traitant des données était rédigée dans le langage cible de la compilation, sous la forme de fonctions invoquées par la partie contrôle, généralement en `C`. La compilation séparée des modules `Esterel` et des fonctions `C` était à la charge de l'utilisateur.

Dans le but de masquer cette dualité, Luciano Lavagno et Ellen M. Sentovitch ont proposé le langage `ECL` [LS99], que l'on peut voir comme un enrichissement du langage `C` par des constructions synchrones provenant d'`Esterel`. Le compilateur `ECL` s'appuie sur le compilateur `Esterel` pour compiler la partie synchrone mais se charge de l'extraction contrôle/données. Le développeur y gagne donc un cadre de travail plus unifié et un processus de compilation simplifié.

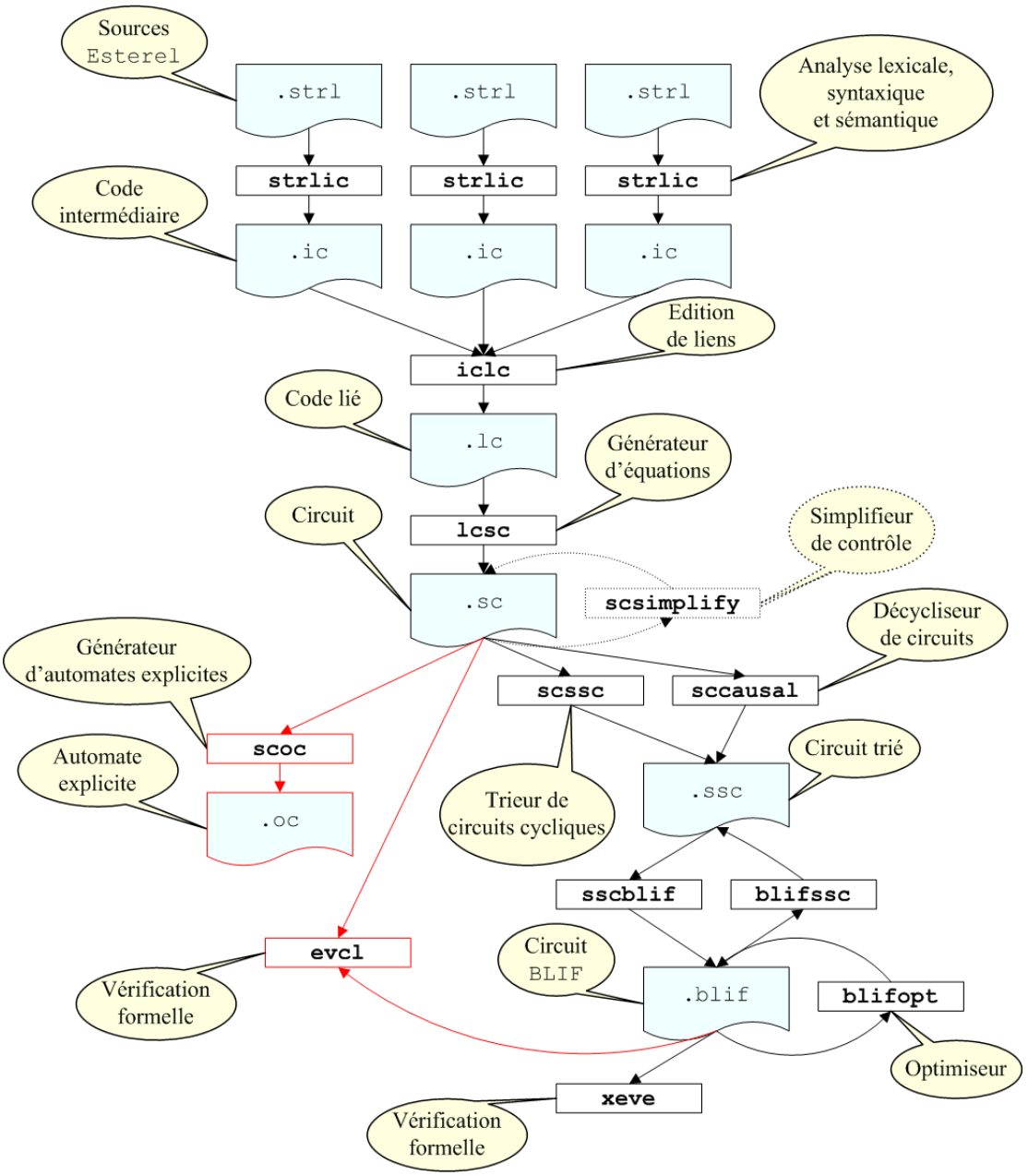


Figure 2.6: La chaîne de compilation d'`Esterel` v5_9x

2.6.2 Le formalisme graphique SyncCharts

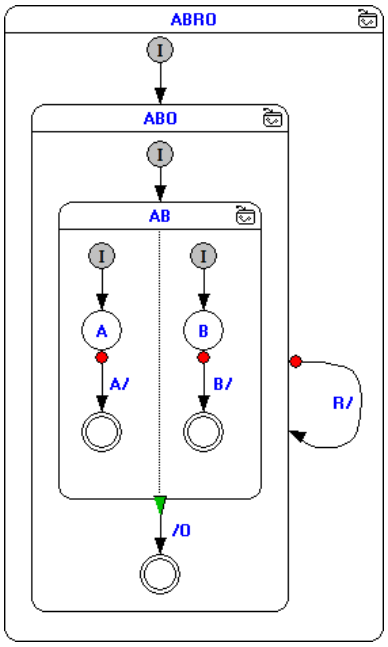
Le formalisme SyncCharts [And96a, And96b, ABD98], proposé par Charles André, peut être vu comme la "version graphique" du langage Esterel. Le formalisme graphique SyncCharts s'inspire d'Argos [Mar91] de Florence Maraninchi, la version synchrone des StateCharts [Har87] de David Harel. Le formalisme SyncCharts a le même pouvoir d'expressivité que le langage Esterel, dans lequel les programmes SyncCharts sont traduits. Selon les programmes et les personnes, l'approche graphique permet parfois une compréhension plus rapide du programme Esterel correspondant.

Le Programme Esterel 2.7 est présenté avec sa traduction en SyncCharts. Ce programme attend que les signaux A et B aient été présents au moins une fois pour émettre le signal O ; dès que le signal R apparaît, le comportement est réinitialisé.

Un programme SyncCharts est formé de macro-états (boîtes à coins arrondis) et d'états simples (cercles). Les cercles gris et pleins mentionnant I correspondent au démarrage d'un macro-état. Les transitions entre états sont représentées par des flèches. Les transitions suite à une terminaison normale démarrent par une flèche verte, la préemption forte (**abort**) est représentée par une flèche démarrant par une boule rouge, la préemption faible (**weak abort**) est représentée par une flèche simple. Les transitions peuvent éventuellement être gardées par une condition et effectuer une action, séparées par une barre oblique. Enfin, le parallélisme est représenté par un découpage vertical ou horizontal des macro-états par des lignes en pointillés.

Le programme SyncCharts présente un emboîtement de *macro-états* (ABRO, ABO et AB) similaire à l'emboîtement des constructions Esterel.

```
1  module ABRO:
2
3    input    A, B, R;
4    output   O;
5
6    loop
7      [
8            await A
9        ||
10           await B
11     ];
12     emit O
13   each R
14
15 end module
```



Programme Esterel 2.7: ABRO

2.6.3 L'environnement de développement `Esterel Studio`

`Esterel Studio` est un environnement de développement intégré dédié aux applications réactives synchrones. Le formalisme de spécification et de développement proposé est le formalisme graphique `SyncCharts`, dont les macro-états peuvent aussi être exprimés textuellement, en `ECL` ou en `Esterel`.

En interne, les applications sont traduites en `Esterel`. Le compilateur `Esterel`, intégré à `Esterel Studio`, les traduit alors en `C` ou `C++` afin d'être embarquées, ou en `VHDL` pour être produites en circuits intégrés.

Comme la plupart des environnements de développements intégrés, `Esterel Studio` fournit un gestionnaire de projet hiérarchique, un déboggeur/simulateur pas-à-pas évolué, etc. `Esterel Studio` intègre en outre deux vérificateurs formels, l'un basé sur des technologies BDDs (une évolution de `Xeve`, cf. 3.1, p. 47), l'autre sur des analyses de satisfiabilité (*SAT Solver*). Enfin, `Esterel Studio` permet la génération de séquences de tests exhaustives selon différents critères (cf. 3.9, p. 71).

2.7 Machines d'états finis

Les programmes que nous analysons correspondent au modèle sémantique des machines d'états finis (*Finite State Machine*, FSM). Ces machines sont des systèmes (logiciels, électroniques, etc.) possédant un état interne, le nombre total d'états atteignables étant donc fini. A partir de leur état initial, ces systèmes peuvent changer d'état selon des règles de transitions qui les caractérisent, en fonction à la fois de leur état courant et de stimuli externes. Plus précisemment, les programmes que nous analysons sont des machines de Mealy [Mea55], qui ajoutent aux machines d'états finis la notion d'actions à effectuer lors des transitions d'états.

Dans le cadre des programmes `Esterel`, `ECL` et `SyncCharts`, l'état interne est encodé dans un vecteur de variables d'états, ou registres, les stimuli extérieurs et les réponses du système à ces stimuli étant concrétisés par des signaux d'entrées/sorties.

Plus formellement, une FSM peut être définie par un tuple $(m, n, p, \delta, \omega, \mathcal{I}, \mathcal{J})$ où ($\mathbf{B} = \{0, 1\}$ est l'ensemble des booléens) :

- m est le nombre de signaux d'entrée (*inputs*) ;
- n est le nombre de variables d'états, ou registres ;
- p est le nombre de signaux de sortie (*outputs*) ;
- $\delta : \mathbf{B}^m \times \mathbf{B}^n \rightarrow \mathbf{B}^n$ est le vecteur des fonctions des registres, ou fonction de transition ;
- $\omega : \mathbf{B}^m \times \mathbf{B}^n \rightarrow \mathbf{B}^p$ est le vecteur des fonctions des sorties ;
- $\mathcal{I} : \mathbf{B}^n \rightarrow \mathbf{B}$ est la fonction caractéristique de l'espace d'état initial ;
- $\mathcal{J} : \mathbf{B}^m \rightarrow \mathbf{B}$ est la fonction caractéristique des combinaisons d'entrées valides (*combinational input care set*) : si le modèle spécifie des implications ou des exclusions entre signaux d'entrée, seules certaines combinaisons d'entrées sont valides.

Nous identifierons par la suite un ensemble et sa fonction caractéristique. De ce fait, $\mathcal{J}(x) = 1$ signifie $x \in \mathcal{J}$. De même, par souci de simplification, nous omettrons les flèches

sur les fonctions vectorielles ou les vecteurs de variables.

2.8 Diagrammes de décisions binaires (BDDs)

Puisque nous identifions les ensembles et leur fonction caractéristique, toutes nos manipulations d'ensembles se ramènent à des manipulations de formules logiques booléennes. Le choix de la structure de données pour représenter et manipuler ces formules a donc un impact majeur sur les coûts des calculs.

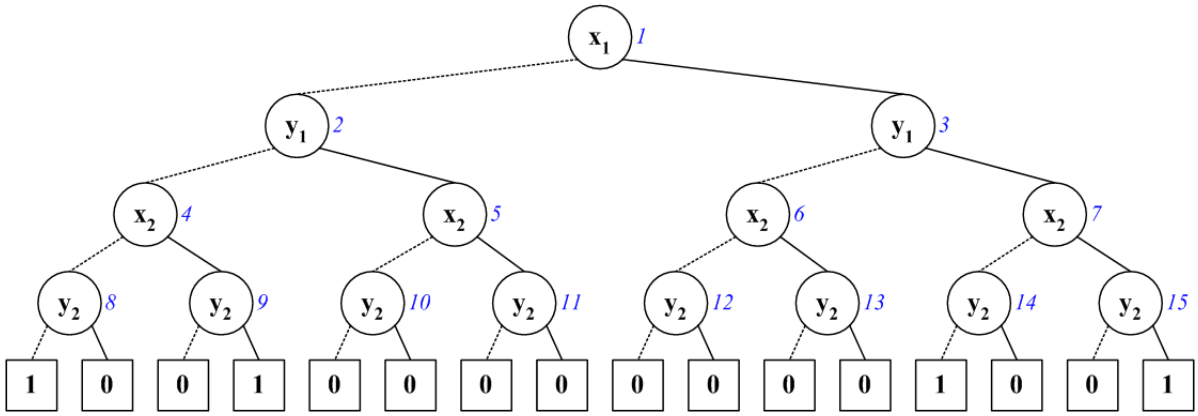
L'idée d'une représentation des fonctions booléennes par des diagrammes de décisions est loin d'être récente [Lee59, Ake78]. La Figure 2.7(a) présente un diagramme de décisions possible pour la formule $(x_1 \Leftrightarrow y_1) \wedge (x_2 \Leftrightarrow y_2)$. A chaque nœud est associé une variable de la formule. Les arcs continus (resp. en pointillés) correspondent au cas où la variable est vraie (resp. fausse). L'évaluation de la fonction pour une valuation donnée des variables revient à suivre dans l'arbre le chemin correspondant à cette valuation jusqu'à la feuille indiquant la valeur de la fonction. Nous avons numéroté les nœuds afin de pouvoir les référencer.

A partir de tels diagrammes de décisions, Bryant [Bry86] a énoncé un ensemble de règles simples pour la structuration de ces diagrammes. Ces règles permettent tout d'abord d'obtenir, modulo un choix d'ordonnancement de variables, une représentation canonique et généralement très compacte de formules booléennes complexes. De plus, ces règles permettent l'application d'algorithmes très efficaces pour manipuler ces formules. La canonicité des représentations permet d'assimiler la notion sémantique d'équivalence de fonctions à la notion syntaxique d'isomorphisme de leur représentation. La canonicité des représentations permet également de mettre en œuvre des techniques de mémoisation, autrement dit de stocker dans des structures de mémoire cache les résultats de calculs coûteux une fois effectués, afin d'éviter de les répéter à l'identique.

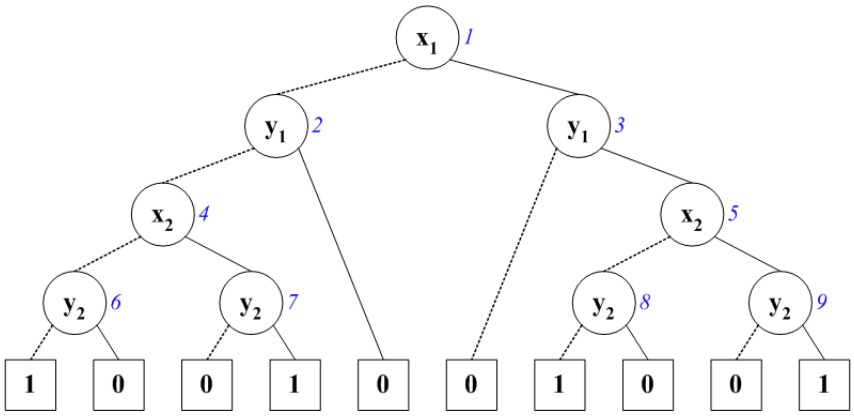
L'idée majeure conduisant à la canonicité des représentations est de stocker tous les nœuds dans une unique table d'adressage dispersé, afin de partager les nœuds identiques. Le code de hachage des nœuds est calculé à partir de l'index de la variable du nœud ainsi que de l'adresse de ses deux nœuds fils.

Les règles permettant d'assurer la canonicité des représentations tout en compactant les diagrammes de décisions binaires (*Binary Decision Diagram*, BDD) sont les suivantes :

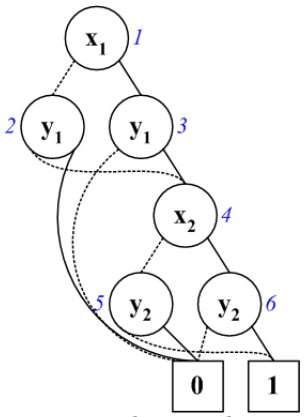
- Ordonnancement des variables : dans un BDD, les variables doivent apparaître dans le même ordre dans tous les chemins possibles de la racine vers les feuilles. Nous avons déjà appliqué cette règle pour la Figure 2.7(a) avec l'ordre $x_1 < y_1 < x_2 < y_2$.
- Suppression des tests inutiles : les nœuds dont les deux fils sont identiques sont remplacés par ce fils. Ainsi, dans la Figure 2.7(a), les nœuds 10, 11, 12 et 13 peuvent être remplacés par 0. Dès lors, les nœuds 5 et 6 peuvent de même être remplacés par 0 (Figure 2.7(b)).
- Unicité des nœuds isomorphes : les nœuds terminaux, autrement dit les constantes booléennes 0 (faux) et 1 (vrai), sont partagés. De même, grâce à la table d'adressage dispersé, les nœuds isomorphes sont partagés et n'existent qu'en instance unique.



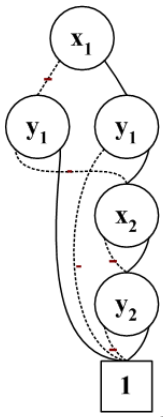
(a) Diagramme expansé



(b) Sans tests inutiles



(c) Avec partage des nœuds isomorphes



(d) Avec marquage des arcs

Figure 2.7: Diagrammes de décisions binaires de la formule $(x_1 \Leftrightarrow y_1) \wedge (x_2 \Leftrightarrow y_2)$

Ainsi, dans la Figure 2.7(b), les nœuds 6 et 8, et 7 et 9 peuvent être fusionnés. Dès lors, les nœuds 4 et 5 peuvent l'être également (Figure 2.7(c)).

– Enfin, les arcs peuvent porter une indication de négation du nœud vers lequel ils pointent. Cela permet de n'avoir qu'un seul nœud terminal, par exemple la constante 1 (vrai). Il est nécessaire, pour garantir la canonicité des représentations, que cette indication de négation n'apparaisse que sur les arcs empruntés lorsque la variable est fausse. Ainsi, une fois les nœuds terminaux unifiés, les nœuds 5 et 6 de la Figure 2.7(c), qui sont opposés, peuvent être partagés, ainsi que les nœuds 2 et 4 (Figure 2.7(d)). Cette dernière règle permet de reconnaître en temps constant si deux fonctions sont opposées. Cette règle peut permettre également, dans le meilleur des cas, de diviser par 2 le nombre de nœuds dans le système. Cette règle s'implémente généralement sans le moindre surcoût en mémoire en utilisant le bit de poids faible des pointeurs vers les nœuds, ce bit étant sinon toujours à 0.

Les BDDs permettent l'implémentation d'algorithmes souvent très efficaces pour la manipulation de fonctions booléennes, que nous ne détaillerons pas [Bry86, Bry92, Som99]. Hormis la négation et le test d'équivalence de fonctions qui se font en temps constant et sans consommation de mémoire, les algorithmes impliquant des BDDs ont une complexité en temps et en espace qui est fonction du nombre de nœuds, voire également du nombre de variables. Le nombre de nœuds nécessaires pour représenter une fonction dépend très fortement de l'ordonnancement des variables choisi. (La détermination d'un ordonnancement de variable optimal est un problème NP-complet [THY93]. Pour nos problèmes d'analyse d'espaces d'états atteignables de circuits logiques, il existe différentes heuristiques de calcul de cet ordonnancement [MWBSV88, TSLaASV90, ATB94]. Une fois l'ordonnancement des variables choisi, il est possible de l'améliorer légèrement par permutations localisées (*sifting*) de variables [Rud93, KF98].)

Enfin, tout au long des calculs impliquant des BDDs, il est majeur de s'assurer que les BDDs manipulés n'ont pas plus de nœuds que nécessaire. Par exemple, il est inutile de conserver les nœuds donnant la valeur d'une fonction à l'extérieur du domaine sur laquelle on va l'appliquer. Il est donc primordial d'éliminer ces nœuds inutiles, par exemple *via* l'opérateur `Restrict` [CM90], qui spécialise une fonction sur un domaine précis.

2.9 Calcul implicite de l'espace d'états atteignables d'une FSM

De nombreuses analyses des systèmes réactifs nécessitent de calculer leur espace d'états atteignables. Nous introduisons les principes de bases de ce calcul à l'aide de techniques implicites.

2.9.1 Principes de base

Construire l'espace d'états atteignables (*Reachable State Space*, RSS) d'une machine d'états finis revient à calculer la limite de la séquence convergente d'ensembles finis d'états définie par les équations suivantes :

$$\begin{aligned} \mathrm{RSS}_0 &= \mathcal{I} \\ \mathrm{RSS}_{k+1} &= \mathrm{RSS}_k \cup \delta(\mathcal{J}, \mathrm{RSS}_k) \end{aligned} \tag{2.1}$$

où nous utilisons l'extension standard des fonctions vers des ensembles :

$$\delta(X, Y) = \{\delta(x, y) \mid x \in X, y \in Y\}$$

L'ensemble RSS_k décrit l'espace d'états atteignables pour une profondeur —autrement dit un nombre de tops d'horloge depuis l'instant initial— inférieure ou égale à k. L'espace d'états atteignables d'un modèle tel que calculé par la séquence RSS a donc une structure en "oignon" où à chaque écaille correspond une profondeur de l'espace d'états (Figure 2.8). On appelle *diamètre* d'un modèle la profondeur maximale des états, autrement dit le nombre maximal d'instants (tops d'horloge) nécessaires pour atteindre n'importe quel état du modèle.

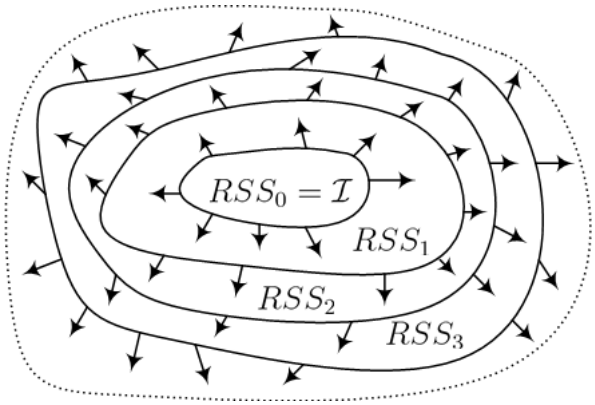


Figure 2.8: Structure en oignon de l'espace d'états atteignables

Mentionnons que si la séquence RSS présente un calcul d'espace d'états atteignable profondeur par profondeur (*breadth-first*), cette stratégie de calcul n'est pas unique. Une stratégie de calcul en profondeur d'abord (*depth-first*) est envisageable, tout comme un mélange de ces deux stratégies [RS95]. Nous verrons dans les Chapitre 4 et 5 des stratégies de calcul énumératives, avec lesquelles les états atteignables sont analysés un par un. Bien que ces stratégies énumératives explorent les états atteignables selon une profondeur croissante, il est tout à fait envisageable d'établir des règles de priorités déterminant l'ordre d'analyse des états [YSAA97, Yan98].

2.9.2 Algorithme de base

L'Algorithme 2.1 présente la méthode de base de calcul de la suite RSS. La boucle principale, qui calcule les ensembles RSS_k successifs s'étend de la ligne 4 à la ligne 17. La ligne 5 construit le domaine spécifique à l'itération courante, comme conjonction de l'ensemble d'états découverts à l'itération précédente et de l'ensemble des inputs valides $\mathcal{J}$. La boucle de la ligne 8 à la ligne 11 construit la fonction de transition spécialisée pour le domaine courant. (Rappelons que la restriction des BDDs au domaine sur lequel ils seront utilisés est une source majeure de réduction du nombre de nœuds.) La ligne 9 construit la fonction associée à un registre particulier, restreint donc au domaine courant. La ligne 10 associe cette fonction à la variable de registre correspondante pour l'ensemble d'états suivants et la combine avec la fonction de transition finale. La ligne 12 applique la fonction de transition à l'ensemble d'états découverts à l'itération précédente. La ligne 13 quantifie existentiellement les variables associées aux anciennes valeurs des registres ainsi que les variables d'inputs. La ligne 14 substitue les variables associées aux nouvelles valeurs des registres par celles associées aux anciennes valeurs, de manière à obtenir une fonction qui ne référence de nouveau que les variables associées aux anciennes valeurs des registres, pour l'itération suivante. Enfin, la ligne 15 calcule l'ensemble des états réellement nouveaux, et la ligne 16 ajoute ces nouveaux états à l'espace d'états atteignables final. L'algorithme termine lorsqu'il n'y a plus de nouveaux états.

```
1   function RSS( FSM )
2     Result ← 𝓘
3     NewStates ← 𝓘
4     repeat
5       Domain ← 𝓙 ∧ NewStates
7       δ ← 1
8       for i ∈ [1..n]
9         δ_i ← BuildRestrictedRegisterFunction( i, Domain )
10        δ ← δ ∧ (NewRegVariable(i) = δ_i)
11      end
12      Image ← δ ∧ NewStates
13      Image ← Quantify( Image, OldRegVariables+InputVariables )
14      Image ← Substitute( Image, NewRegVariables, OldRegVariables )
15      NewStates ← Image ∧ ¬Result
16      Result ← Result ∨ Image
17    until NewStates = 0
```

Algorithme 2.1: Calcul du point fixe de la suite RSS

A notre connaissance, il n'existe pas de preuve formelle de la validité de cette algorithme. Intuitivement, la terminaison de cet algorithme est garantie par le fait que l'on ne calcule jamais deux fois l'image d'un même état (ligne 15) et que le nombre d'états atteignables

est fini ($\leq 2^n$). La prise en compte de la totalité des états atteignables est garantie par l'itération des lignes 1 à 4, qui ne termine que lorsque l'intersection entre l'ensemble d'états résultant du calcul de l'itération courante et l'ensemble des états connus jusqu'à présent (ligne 15) est vide. Une analyse plus fine de cet algorithme est disponible dans [CM91].

2.9.3 Raffinements

Si cet algorithme découpe clairement les différentes phases du processsus, ce découpage est loin d'être aussi effectif dans les implémentations réelles :

- La fonction de transition (construite aux lignes 8 à 10) ne combine en fait que des représentants des classes d'équivalences des fonctions de registres (cf. sous-section suivante).
- Une fois les fonctions redondantes éliminées, la fonction de transition n'est pas construite en un unique BDD. Les calculs d'images se font en partitionnant la fonction de transition [BCL91]. Le partitionnement se fait généralement selon des heuristiques basées sur les supports des fonctions [BCL+94, GB94, RAB+95, MS00b, CCJ+01a, CCJ+01b] ou selon des informations de haut niveau sur la structure des modèles [SHM00, MS00a, MS01]. L'utilisation d'informations structurelles spécifiques aux programmes `Esterel` pour guider le partitionnement est actuellement étudiée par Eric Vecchié.
- Les quantifications existentielles et les substitutions de variables (lignes 13 et 14) ne se font pas en une seule passe mais à la volée, durant le calcul d'image [HKB96]. Plus précisément, les quantifications existentielles sont faites au plus tôt (*early quantification*), dès que les variables à quantifier n'apparaissent plus dans les expressions demeurant à traiter. Cela est fait avec pour objectif de réduire la taille des BDDs intermédiaires, les quantifications existentielles produisant généralement des BDDs de moindre taille.

De telles optimisations ont essentiellement pour objectif de réduire la consommation mémoire. En effet, les pics de consommation mémoire interviennent généralement lors des calculs d'image [RS95]. Les optimisations que nous venons de mentionner ne remettent toutefois pas fondamentalement en cause les analyses de complexité de la section 2.9.6 (p. 43).

Des axes de recherches similaires ont été étudiés pour réduire le coût des calculs d'espaces d'états atteignables, dont notamment :

- En partitionnant les ensembles dont on calcule les images [CCQ96, CCLQ97].
- En abandonnant les calculs d'espace d'états atteignables profondeur par profondeur (*breadth-first*) en ne travaillant que sur des BDDs "élagués" pour en renforcer la densité [RS95].
- En partitionnant le modèle analysé (cf. 3.3.5, p. 60).

2.9.4 Classes d'équivalences de fonctions de registres

Les BDDs offrent, modulo l'ordre des variables choisi, une forme *canonique* des fonctions qu'ils décrivent. Ainsi, deux fonctions égales ont le même pointeur et deux fonctions opposées ne diffèrent que par le bit de poids faible de leur pointeur. Des fonctions égales ou opposées sont dites *équivalentes*. Il arrive fréquemment que des modèles comportent de nombreux registres redondants, *a fortiori* lorsque le code est généré.

Une fois les fonctions des registres construites, celles-ci sont triées par *classes d'équivalences*, et un seul représentant de chaque classe est conservé. Les calculs d'images ne se font donc que sur un vecteur de fonctions uniques. De ce fait, les résultats des calculs d'images successifs ne sont pas représentés par des BDDs purs, mais par des *fonctions booléennes compactées* (CBFs, *Compacted Boolean Functions*). Ces CBFs sont des couples formés du BDD résultat, dépourvu de variables redondantes, et d'une liste d'équivalence de variables.

Dès lors, tous les ensembles d'états manipulés par le système (ensembles de nouveaux états de chaque profondeur, ensembles des états atteignables, etc.) sont en fait représentés par de telles CBFs. La représentation des ensembles d'états par des fonctions booléennes compactes permet de considérables réductions du nombre de nœuds des BDDs manipulés et donc un gain à la fois en temps et en consommation mémoire.

A différentes étapes du calcul de l'espace d'états atteignables, il est nécessaire de combiner des CBFs entre eux. Ces combinaisons surviennent par exemple à la fin de chaque profondeur, lorsque les états images doivent être filtrés pour ne conserver que les nouveaux états, ou lorsque ces nouveaux états doivent être ajoutés à l'espace d'états atteignables total.

Pour combiner deux CBFs entre eux, il est nécessaire d'expanser partiellement les BDDs et de réintroduire les variables qui ne sont pas communément redondantes. Cette expansion revient à substituer des variables, ce qui peut avoir une complexité exponentielle dans le pire des cas.

Le coût de ces substitutions peut devenir critique lors de l'analyse de circuits à forte proportion de registres redondants. La proportion de registres redondants atteint par exemple son paroxysme dans les circuits testeurs de circuits (*testbenches*), au comportement totalement linéaire. Ces circuits se ramènent à des successions d'émissions de signaux entrecoupées d'instructions de pause : très peu de registres, voire un seul, sont actifs à la fois (cf. Expérimentation 6.7, p. 147). Moins anecdotiques, les circuits implémentant des transactions comportent aussi fréquemment un grand nombre de registres redondants (cf. Expérimentation 6.6, p. 139). Les circuits implémentant des transactions se retrouvent souvent dans les approches de vérification de SoC (*Top Level Validation*).

Les circuits à forte proportion de registres redondants se prêtent donc peu aux analyses par BDDs, du fait d'explosions combinatoires lors des combinaisons de CBFs. Il est alors préférable d'envisager une analyse explicite ou hybride implicite/explicite, respectivement décrites aux chapitres 4 (p. 75) et 5 (p. 103).

2.9.5 Exemple

Nous pouvons utiliser l'Algorithme 2.1 pour calculer l'espace d'états atteignables du circuit séquentiel de la Figure 2.9. L'état initial $(1, 0, 0, 0)$ du circuit est indiqué par les valeurs apparaissant sur la droite des registres. La première itération découvre le nouvel état $(0, 1, 0, 0)$; la deuxième itération découvre le nouvel état $(0, 0, 1, 0)$; la troisième itération atteint le point fixe. Les trois registres r_1, r_2 et r_3 sont exclusifs et le registre r_4 est toujours à 0.

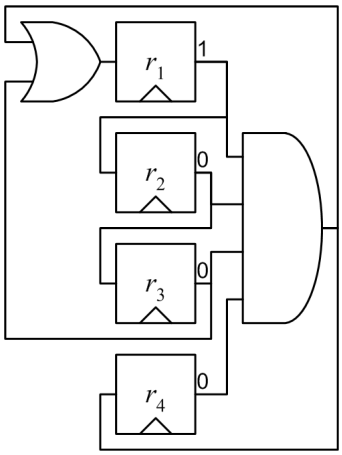


Figure 2.9: Exemple de circuit séquentiel

Nous utiliserons de nouveau cet exemple dans le chapitre suivant, afin d'illustrer des algorithmes de calculs d'espaces d'états atteignables approchés par excès.

2.9.6 Analyse de complexité

La complexité des opérations impliquant des BDDs s'exprime par rapport au nombre de nœuds des BDDs. La négation a un coût constant, la conjonction et la disjonction ont un coût polynomial [CBM89, CM91], les quantifications existentielles et les substitutions ont un coût potentiellement exponentiel par rapport au nombre de variables à traiter. (De manière informelle, $\exists x \,.\, f(x)$ revient à calculer $f(0) + f(1)$, $\exists x,y \,.\, f(x, y)$ revient à calculer $f(0, 0) + f(0, 1) + f(1, 0) + f(1, 1)$, etc.)

Pour la plupart des modèles, on peut négliger les phases de construction du domaine courant (ligne 5) et de construction des fonctions des registres (ligne 9). De même, les phases de tris entre nouveaux états et états connus (ligne 15) et d'agrégation des nouveaux états aux états connus (ligne 16) peuvent être négligées, hormis pour les modèles à forte proportion de registres redondants (cf. 2.9.4, p. 42).

En général, la phase la plus coûteuse du calcul de l'espace d'états atteignables est donc le calcul d'image, qui a un coût potentiellement exponentiel par rapport au nombre de variables d'état et d'inputs, du fait des quantifications existentielles et des substitutions de variables. Les variables d'inputs n'interviennent que dans les calculs intermédiaires : elles

contribuent donc à la consommation de mémoire lors des calculs intermédiaires, mais pas à la consommation de mémoire nécessaire pour représenter l'espace d'états atteignables du modèle. Les variables d'états, elles, interviennent à la fois dans la consommation mémoire nécessaire aux calculs intermédiaires et pour représenter l'espace d'états atteignables du modèle.

Enfin, le diamètre d'un modèle (cf. 2.9, p. 38) est aussi une métrique à prendre en compte pour évaluer la complexité du calcul de l'espace d'états atteignables, puisqu'il détermine le nombre d'itérations nécessaires (lignes 4-17 de l'Algorithme 2.1).

2.10 Vérification formelle par observateurs

Nous proposons dans nos différents outils la vérification formelle de programmes réactifs synchrones selon le paradigme des observateurs [HLR93]. Cela consiste à exécuter, en parallèle du modèle à vérifier, un ou des modules dont les entrées sont celles du modèle observé ainsi que ses sorties (Figure 2.10). En fonction des entrées reçues et de l'état interne de l'observateur, celui-ci contrôle la correction des sorties du modèle observé et émet des signaux spécifiques en cas d'erreur. La vérification formelle consiste alors à vérifier que ces signaux ne puissent jamais être émis.

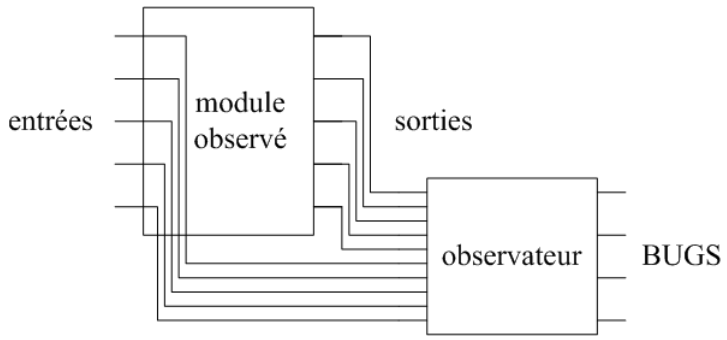


Figure 2.10: Observateur exécuté en parallèle du modèle à observer

Par exemple, en reprenant l'exemple du programme `AB_O` (Programme 2.3), si l'on voulait vérifier que le signal `O` ne pouvait jamais avoir été émis sans réception préalable du signal `A`, il serait possible d'écrire un observateur comme celui du programme 2.8.

2.10.1 Propriétés de sûreté

Cette technique permet aisément la vérification de propriétés de *sûreté* (*safety properties*). Ces propriétés traitent du passé ou du présent et expriment par exemple des invariants du genre "quelque chose de mal n'arrive jamais". Pour un système de contrôle d'ascenseurs, une propriété de sûreté typique est que les ascenseurs ne peuvent être en mouvement avec la porte ouverte.

Les propriétés de sûreté peuvent être directement rédigées sous la forme de modules `Esterel`, `SyncCharts`, `ECL` ou de n'importe quel formalisme permettant une traduction

```
1 module AB_O:
2
3   input   A, B;
4   output  O;
5                                          ||   abort
6   [ await A || await B ] ;                      await O ;
7   emit O                                       emit BUG
8                                          ||   when A
9 end module
```

Programme `Esterel` 2.8: `AB_O` avec observateur

vers ces langages. Les systèmes `TempEst` [JPO95] et `Hurricane` permettent par exemple la traduction en modules `Esterel` de propriétés de sûreté exprimées en logique `LTL` (*Linear-time Temporal Logic*) [MP92].

Le contrôle des propriétés de sûreté consiste à vérifier que l'intersection entre l'espace d'états atteignables du modèle et l'ensemble des états violant les propriétés est réduit à l'ensemble vide.

2.10.2 Propriétés de vivacité

Les propriétés de *vivacité* (*liveness properties*) traitent du futur et expriment des propriétés de la forme "quelque chose de notable arrivera tôt ou tard". De telles propriétés ne peuvent être exprimées par des modules `Esterel` : la vérification de propriétés de vivacité nécessite de mettre en œuvre des techniques spécifiques.

Dans le cadre particulier du langage `Esterel`, les récents travaux de Xavier Thirioux [Thi02] permettent une traduction efficace de propriétés `LTL` arbitraires en automates de Büchi. Ces automates doivent ensuite être confrontés avec l'espace d'états atteignables du modèle afin de vérifier qu'il n'existe pas de chemin menant à une boucle infinie violant les propriétés.

Les propriétés de vivacité étant moins utilisées en pratique que les propriétés de sûreté, nous n'avons pas exploré cette voie. Notons toutefois que le paradigme des observateurs permet l'expression de propriétés de vivacité explicitement *bornées*, de la forme "quelque chose de notable arrivera avant un certain temps".

2.10.3 Equivalence de machines

Enfin, notons que le paradigme des observateurs permet aussi la vérification formelle d'*équivalence* de machines (*equivalence checking*). Il faut alors exécuter en parallèle les machines dont on veut vérifier l'équivalence avec un observateur qui compare chaque sorties deux à deux (avec un *miter* de la forme $\neg(O_1 \oplus O_2)$ [Bra93]).

Chapitre 3

Approche implicite

Apparues au début des années 90, les méthodes implicites sont caractérisées par l'utilisation de Diagrammes de Décisions Binaires (*Binary Decision Diagrames*, BDDs) [Bry86, Som99] pour décrire et manipuler des ensembles d'états *en intention*.

Par la canonicité des fonctions représentées et le partage automatique de leurs nœuds, les BDDs permettent de décorréler tailles de BDDs et cardinaux des ensembles représentés. A l'inverse des méthodes explicites, où l'espace mémoire nécessaire à la représentation d'un ensemble donné est prédictible et généralement important, les BDDs peuvent permettre une représentation d'ensembles de cardinaux arbitraires bien plus économe en mémoire. Néanmoins les opérations cruciales sur les BDDs peuvent être de complexité exponentielle, dans le pire des cas, en fonction du nombre de variables impliquées.

Nous présentons dans ce chapitre plusieurs techniques visant à réduire le nombre de variables impliquées dans les différentes manipulations de BDDs nécessaires aux calculs d'espaces d'états de circuits.

La section 3.1 présente `Xeve`, l'outil de vérification formelle à base de BDDs proposé pour `Esterel` à partir de 1997. La section 3.2 présente `evcl`, l'outil de vérification formelle à base de BDDs que nous avons développé, et dont les fonctionnalités sont décrites plus en détail dans les sections 3.3 à 3.8. La section 3.9 présente l'utilisation de techniques implicites mises en œuvre pour la génération de test au sein d'`Esterel Studio`. Enfin, la section 3.10 présente une conclusion sur les approches implicites.

3.1 L'existant : `Xeve`

En 1997, un outil graphique dédié à la vérification formelle de programmes `Esterel` a été développé par Amar Bouali : `Xeve`, "*an Esterel Verification Environment*" [Bou97, Bou98].

`Xeve` est une interface graphique permettant de faciliter l'utilisation de deux outils précédemment disponibles en ligne de commande, `bliffc2` et `checkblif`, permettant respectivement :

– la minimisation de machines d'états finis par *bisimulation* [BdS92] ;

– la vérification formelle du statut de signaux de sortie.

Pour ce qui est de la vérification formelle, ce que permet `Xeve` se résume essentiellement à une interface graphique (Figure 3.1) permettant de piloter le moteur d'analyse par BDDs sous-jacent, `TiGeR`, développé par Madre, Coudert et Touati [CMT93]. Les seules options disponibles étaient les possibilités de contraindre des entrées à être toujours ou jamais présentes et de sélectionner les signaux de sortie à observer.

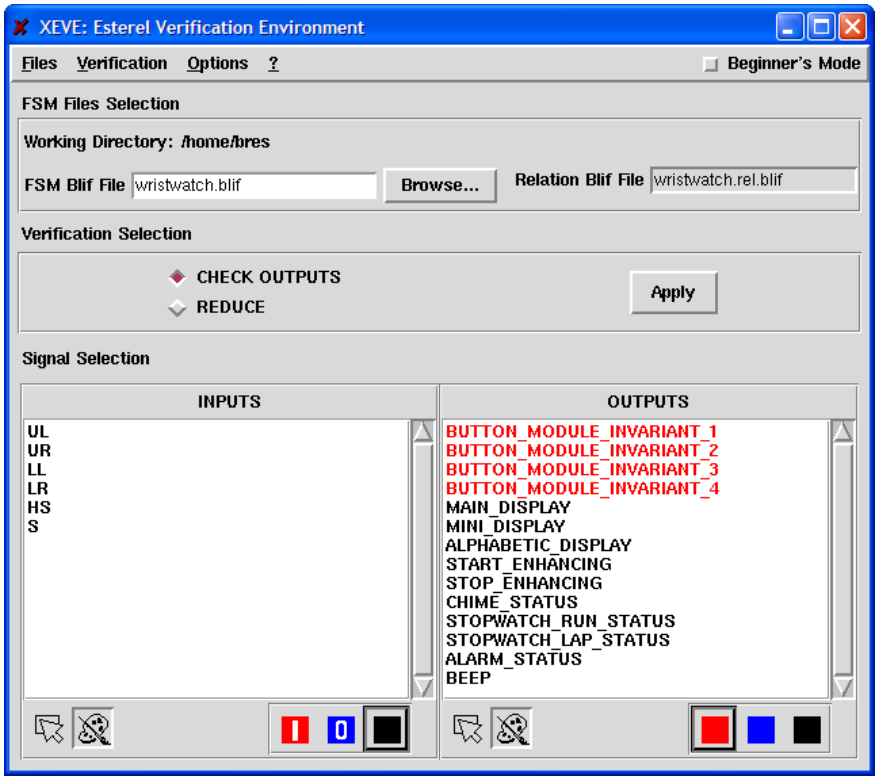


Figure 3.1: L'interface graphique `Xeve`

Par la suite, le compilateur `Esterel` a été industrialisé par la société Esterel Technologies et `Xeve`, désormais intégré à l'environnement de développement `Esterel Studio`, a été amélioré. Les améliorations triviales suivantes, que nous avions suggérées, ont notamment été apportées à la version v5_94.

3.1.1 Améliorations triviales de `Xeve`

Un des principes de base de `Xeve` était de ne calculer l'espace d'états accessibles d'un modèle qu'une seule fois. L'espace d'états accessibles ainsi calculé pouvait alors être utilisé pour différentes vérifications successives. Si cette approche se justifie pour les modèles dont le calcul d'espace d'états accessibles complet est aisé, elle ne l'est pas pour les modèles plus complexes. Il est alors nécessaire, au minimum :

– de contrôler les propriétés à chaque étape du calcul d'espace d'états atteignables ;

– de tenir compte des signaux à vérifier et de simplifier le circuit à analyser en conséquence.

Vérification incrémentale des propriétés

Parce que `Xeve` dissociait le processus de calcul de l'espace d'états atteignables de celui de vérification des propriétés, les propriétés n'étaient vérifiées qu'une fois avoir complètement terminé, avec succès, le calcul d'espace d'états atteignables. Cette approche présente plusieurs inconvénients :

– si le calcul d'espace d'états échoue, le vérifieur ne fournit aucune information, alors qu'il pourrait au moins indiquer jusqu'à quelle profondeur le modèle est correct ;
– si toutes les propriétés sont trouvées violables à une certaine profondeur, avoir poursuivi le calcul de l'espace d'états atteignables était sans intérêt.

Transitive Network Sweeping automatique

De même, `Xeve` calculait l'espace d'états atteignables du circuit complet, quels que soient les signaux de sortie à observer. Or, lorsqu'on ne se focalise que sur un petit ensemble de signaux, il se peut que certaines variables d'états ne puissent pas du tout influencer ces signaux : ces variables d'états ne font pas partie du *cône d'influence* des signaux à observer.

Le cône d'influence de signaux se calcule par un simple parcours du graphe de dépendance de données du circuit, en arrière à partir de ces signaux (et généralement en profondeur). Une fois le parcours terminé, les variables d'état non visitées ne font donc pas partie du cône d'influence des signaux à observer et peuvent être supprimées du circuit.

La Figure 3.2 présente les *supports* des différentes fonctions d'un circuit. Le support d'une fonction est l'ensemble des variables qu'elle référence. Imaginons que l'on ne s'intéresse qu'à la sortie `O3`. Le support de celle-ci est constituée des entrées `I2` et `I3` et du registre `R1`. Le support du registre `R1` est constitué de l'entrée `I3` ainsi que des registres `R1` et `R2`. Le support du registre `R2` est constitué des registres `R1`, `R2` et `R3`. Le support du registre `R3` est constitué des entrées `I2` et `I3` et des registres `R1` et `R2`. L'entrée `I1` et le regitre `R4` n'ont donc aucune influence sur la sortie `O3`. Toute la logique et les registres ne servant qu'aux sorties `O1` et `O2` et au registre `R4` peuvent être supprimés.

Cette amélioration triviale permet des gains décisifs sur certains types de circuits. Par exemple, dans un cas d'étude en vraie grandeur développé chez Dassault Aviation, les propriétés du système de gestion du carburant de l'avion Mirage [BNLD00] n'avaient pas pu être prouvées avec 1 Go de mémoire. (Le calcul d'espace d'états atteignables du système complet n'est pas non plus possible avec 2 Go de mémoire.) En décomposant la vérification en différentes sessions, chacune focalisée sur une propriété et en ne considérant que les variables d'états présentes dans le cône d'influence de la propriété, l'ensemble des propriétés sont alors vérifiables en un peu plus d'un quart d'heure et en consommant moins de 75 Mo de mémoire (cf. 6.4, p. 118). Nous verrons dans la section 3.3 des techniques permettant de réduire encore le temps de vérification formelle de ce modèle à quelques secondes et en consommant moins de 5 Mo de mémoire.

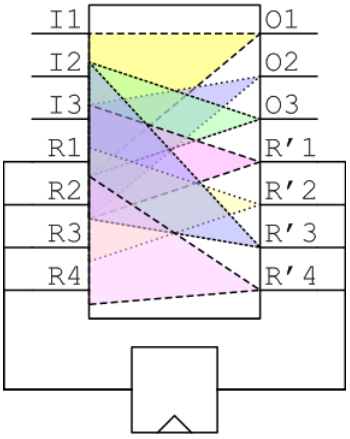


Figure 3.2: Cône d'influence des fonctions d'un circuit

3.2 L'outil de vérification formelle `evcl`

Nous avons estimé que `Xeve` n'était pas un cadre de développement idéal pour nos expérimentations. Maintenir une interface graphique permettant uniquement de faciliter l'accès à un outil expérimental aurait été inutilement fastidieux et d'intérêt limité, du moins dans les premières phases de développement. Nous avons donc développé un nouvel outil de vérification formelle à base de techniques implicites : `evcl` (*Esterel Verification Command Line*)

Afin de faciliter une éventuelle réutilisation de nos développements, la majorité de nos travaux sur ce sujet a en fait été implémentée dans une librairie qui étend le package de BDDs `TiGeR` et remplace certaines de ses fonctionnalités : la librairie `TiGeREnh` (*`TiGeR` Enhancements*). L'outil `evcl` se résume donc essentiellement à fournir une interface textuelle, d'usage le plus aisé possible, permettant le pilotage de la librairie `TiGeREnh`.

L'apport majeur d'`evcl`, décrit à la section 3.3 (p. 51), est de réduire le coût de l'analyse de certains types de programmes, par réduction du nombre de variables à prendre en considération dans les calculs.

`Xeve` ne travaille que sur les circuits au format `BLIF`, soit des équations booléennes dépourvues de toute information supplémentaire. Dès le départ nous avons voulu qu'`evcl`, outre le support des circuits au format `BLIF`, propose l'analyse directe des circuits tels que générés par le compilateur `Esterel` (cf. Figure 2.6, p. 33). Cela permet de tirer partie d'informations structurelles sur le modèle analysé, comme nous le verrons en 3.4 (p. 61). Par exemple, cela permet à `evcl` de tenter de vérifier les propriétés sans le moindre calcul d'espace d'états atteignables, par confrontation des propriétés avec une approximation *par construction* de l'espace d'états atteignables (cf. 3.4.1, p. 61).

Comme représentation interne des circuits logiques, `evcl` propose l'utilisation d'un graphe exclusivement composé de portes `et` binaires et d'inverseurs (cf. 3.5.3, p. 66). Cette représentation, plus simplifiée que la représentation interne du package de BDDs, facilite les analyses du circuit. Elle permet aussi de *compacter* le circuit à la volée, par hachage structurel.

Enfin, `evcl` propose l'analyse de programmes avec des observateurs séparés (vérification en boîte noire, cf. 3.6, p. 68). Cela permet de vérifier des modèles sans avoir à y intégrer des observateurs, modules uniquement utiles à la vérification. De plus, cela permet d'éviter le surcoût engendré par le câblage produit par le compilateur `Esterel` v5, qui avait tendance à trop lier entre elles les différentes parties du circuit, limitant ainsi l'impact du *sweeping*.

3.3 Réduction du nombre de variables pour le calcul du RSS

Le problème principal du calcul d'espace d'états atteignables d'un modèle est lié au nombre de variables —d'états ou intermédiaires— impliquées dans ces calculs (cf. 2.9.6, p. 43).

Une technique triviale de réduction du nombre de variables à manipuler consiste à ne conserver que celles réellement présentes dans le cône d'influence des propriétés à vérifier, par *sweeping*. Si cette technique permet de ne calculer que l'espace d'états atteignables d'un sous-ensemble du circuit, les résultats demeurent *exacts* : il n'y a aucune approximation, donc aucune fausse réponse.

Les sections suivantes présentent d'autres techniques de réduction du nombre de variables dans les calculs d'espaces d'états atteignables, au prix d'*approximations* de ces espaces d'états. Ces techniques sont basées sur des *abstractions* du modèle concret, ces abstractions ayant pour objectif de simplifier les calculs.

Ces techniques demeurent *conservatives* vis-à-vis des états atteignables : aucun état réellement atteignable du modèle concret ne peut être déterminé comme non atteignable sur le modèle abstrait. Pour la vérification formelle de propriétés, cela signifie qu'aucune propriété ne pourra être validée à tort (*false positive*). A l'inverse, des états non atteignables dans le modèle concret pourront être déterminés atteignables dans le modèle abstrait. Pour la vérification formelle de propriétés, cela pourra conduire à la réfutation erronée de propriétés en réalité correctes (*false negative*). Ces réfutations erronées pourront toutefois être contrôlées *a posteriori* à peu de frais (cf. 3.3.4, p. 60).

3.3.1 Remplacement des variables d'états par des entrées libres

Une abstraction simple et très utilisée pour réduire le nombre de variables à manipuler consiste à remplacer certaines variables d'états (registres) du circuit par des signaux d'entrée libres de prendre n'importe quelle valeur à n'importe quel instant.

Cette technique permet ainsi de réduire :

- le nombre de fonctions de registres à construire ;
- le nombre de fonctions de registres à manipuler durant les calculs d'images ;
- le nombre de variables à substituer.

Notons en outre que le remplacement de variables d'états par des entrées libres entraîne la disparition "transitive" des variables d'états et d'entrées qui n'intervenaient que dans le cône d'influence des variables d'états remplacées, du fait du *sweeping*.

Cette technique ne réduit pas le nombre de variables à quantifier existentiellement : les registres transformés en entrées libres sont tout autant à quantifier.

Le remplacement des variables d'états par des entrées libres relâche les contraintes entre ces variables d'états : elles peuvent désormais avoir n'importe quelle valeur à n'importe quel instant. On y perd notamment des exclusions entre variables d'états, ce qui conduit à admettre que le modèle abstrait puisse être au même moment dans plusieurs états du modèle concret.

Le résultat du calcul de l'espace d'états atteignables est donc une approximation *par excès* de l'espace d'états réellement atteignables. Cette approximation par excès peut déclencher un effet "boule de neige" : des états non atteignables du modèle concret sont trouvés à tort atteignables dans le modèle abstrait, ces états conduisent à leur tour à des états qui peuvent ne pas être réellement atteignables dans le modèle concret, etc. De ce fait, les variables à remplacer par des entrées libres doivent être choisies avec soin.

Lorsqu'une variable d'état est remplacée par une entrée libre du modèle, la corrélation entre les diverses occurrences de cette variable dans les équations du circuit est conservée. Par exemple, la Figure 3.3 présente un fragment de circuit généré par une expression telle que **`present I then`** *p* **`else`** *q*. Le fil *go*, qui détermine si l'expression est active, est combiné avec l'entrée `I`. Supposons que le fil *go* soit alimenté par une variable d'état : quand bien même cette variable d'état serait remplacée par une entrée libre, nous pourrions toujours déterminer que les branches *then* et *else* sont mutuellement exclusives.

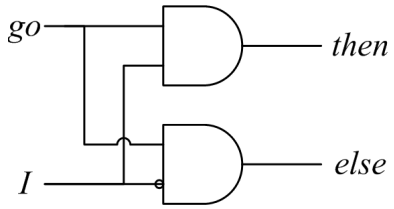


Figure 3.3: Fragment de circuit généré par l'expression **`present I then ... else ...`**

Notons que la technique de remplacement des variables d'états par des entrées libres est souvent implicite dans de nombreuses études, notamment celles mentionnées en 3.3.5 (p. 60).

Exemple

Appliquons cette technique au circuit de la Figure 2.9 (p. 43), en supposant que nous voulions prouver que $r_1 \wedge r_2 = 0$.

Nous pouvons remplacer r_4 par une entrée libre et appliquer l'algorithme de calcul de l'espace d'états atteignables du modèle : à partir de l'état initial $(1, 0, 0)$, la première itération découvre le nouvel état $(0, 1, 0)$, la seconde itération découvre le nouvel état $(0, 0, 1)$ et la troisième itération atteint le point fixe. Nous pouvons toujours prouver que $r_1 \wedge r_2 = 0$, mais les calculs ont été effectués avec moins de variables d'état, donc moins de fonctions à construire, de variables à substituer, mais tout autant de variables à quantifier existentiellement.

A l'inverse, si nous choisissons de remplacer r_3 par une entrée libre alors, à partir de l'état initial $(r_1, r_2, r_4) = (1, 0, 0)$, la première itération découvre les nouveaux états $(0, 1, 0)$ et $(1, 1, 0)$, la seconde itération découvre le nouvel état $(0, 0, 0)$ et la troisième itération atteint le point fixe. Du fait d'une sur-approximation excessive due à un mauvais choix des variables d'état à remplacer par des entrées libres, nous échouons à prouver que $r_1 \wedge r_2 = 0$ dès la première itération.

3.3.2 Abstraction des variables à l'aide d'une logique trivaluée

Le remplacement de variables d'états par des entrées libre est une technique très intéressante pour réduire le coût des calculs d'espaces d'états atteignables. Néanmoins, cette technique ne réduit pas le nombre de quantifications existentielles à effectuer. Pour ce faire, nous proposons d'utiliser une logique trivaluée, qui va nous permettre de faire complètement disparaître certaines variables des calculs d'images, par *pré-quantification*.

Logique trivaluée

La logique trivaluée de Scott est basée sur la logique booléenne habituelle à laquelle on ajoute une troisième valeur, notée $\perp$, signifiant qu'une variable est *indéfinie*, et en étendant les opérateurs booléens usuels.

A la suite des travaux de Malik [Mal94], Shiple, Berry et Touati [SBT96] ont utilisé une logique de Scott trivaluée pour analyser les circuit cycliques. L'idée était d'utiliser la valeur $\perp$ pour représenter une valeur électriquement instable. En injectant dans le circuit une valeur $\perp$ dans les portes alimentant un cycle, il faut alors contrôler qu'il n'existe pas de valuation d'entrées valide permettant aux valeurs $\perp$ de se propager jusqu'aux sorties ou registres du circuit, ce qui signifierait que le circuit n'est pas constructif.

De manière assez similaire, nous proposons d'introduire une troisième valeur signifiant qu'une variable est *définie mais indifférenciée*, c'est-à-dire qu'elle a une valeur exacte entre *vrai* et *faux*, notée d. L'algèbre de la logique trivaluée $\{0, 1, d\}$ est exactement la même que celle de l'algèbre $\{0, 1, \perp\}$ et nous ne faisons qu'utiliser la logique usuelle de Scott. Néanmoins, l'intuition étant différente, nous préférons utiliser le symbole d.

Les 3 valeurs $\{0, 1, d\}$ sont respectivement encodées en utilisant les paires de valeurs booléennes $\{1, 0\}$, $\{0, 1\}$ et $\{0, 0\}$. Dans les expressions, nous encodons les variables à conserver par une paire de la forme $(x, \overline{x})$ et les variables que nous voulons abstraire par la paire constante $d = (0, 0)$.

Les fonctions trivaluées (*Ternary Valued Functions*, TVFs) sont encodées en utilisant une paire de fonctions booléennes (f^0, f^1), telles que f^0 (resp. f^1) est la fonction caractéristique de l'ensemble pour lequel f s'évalue à 0 (resp. 1). L'ensemble f^d des valeurs pour lesquelles f est indifférenciée est $f^d = \overline{f^0 + f^1}$ et, par construction, $f^0 \cdot f^1$ est toujours faux. Dès lors, f ne caractérise plus une partition de deux ensembles $\left(f, \overline{f}\right)$ comme en logique booléenne, mais une partition de trois ensembles $\left(f^0, f^1, f^d\right)$, comme le montre la Figure 3.4.

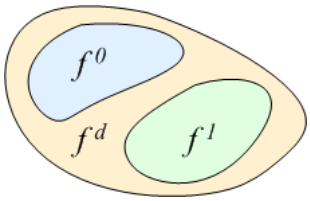


Figure 3.4: Ensembles f^0, f^1 et $f^d = \overline{f^0 + f^1}$

Les opérateurs booléens standards sont étendus aux fonctions trivaluées selon les formules suivantes :

$$\begin{aligned} \neg\left(f^0, f^1\right) &= \left(f^1, f^0\right) \\ \left(f^0, f^1\right) + \left(g^0, g^1\right) &= \left(f^0 \cdot g^0, f^1 + g^1\right) \\ \left(f^0, f^1\right) \cdot \left(g^0, g^1\right) &= \left(f^0 + g^0, f^1 \cdot g^1\right) \end{aligned}$$

Par exemple, $f+g$ est faux lorsqu'à la fois f et g sont fausses, mais vrai dès lors que f ou g est vraie.

Les logiques trivaluées sont connues pour être monotones.

Application aux calculs d'espaces d'états atteignables

L'abstraction de variables à l'aide d'une logique trivaluée peut nous permettre de réduire encore plus le coût de calcul de l'espace d'états atteignables que ne le permettait la technique précédente de remplacement de variables d'états par des entrées libres. Alors que les variables d'états remplacées par des entrées libres continuaient à intervenir dans les calculs intermédiaires et devaient toujours être quantifiées existentiellement, les variables abstraites sont directement remplacées par des constantes. Elles disparaissent donc complètement des BDDs manipulés.

Reprenons la méthode de calcul du RSS selon des méthodes implicites, introduite en 2.9 (p. 38). A l'aide de techniques symboliques, construire l'espace d'états atteignables d'un modèle revient à calculer la limite de la séquence convergente d'ensembles finis d'états définie par les équations suivantes [CBM89, CM91] :

$$\begin{aligned} \mathrm{RSS}_0 &= \mathcal{I} \\ \mathrm{RSS}_{k+1} &= \mathrm{RSS}_k \cup \delta(\mathcal{J}, \mathrm{RSS}_k) \end{aligned} \tag{3.1}$$

En utilisant des BDDs pour manipuler les fonctions caractéristiques des ensembles, (3.1) devient :

$$\mathrm{RSS}_{k+1} = \mathrm{RSS}_k \cup \{\, r' \in \mathbf{B}^n \mid \exists r \in \mathrm{RSS}_k\,,\, \exists i \in \mathbf{B}^m \,.\, \mathcal{J}(i) \wedge r' = \delta(i, r) \,\} \tag{3.2}$$

Dans [CM91], Coudert et Madre ont introduit l'opérateur *image* $\mathrm{Img}(f, \chi)$, qui calcule

l'image par une fonction vectorielle f de l'espace d'états de fonction caractéristique χ[1] :

$$\mathrm{Img}(f,\chi) = \lambda r'.\left(\exists r,\, i\,.\,\chi(r) \wedge \mathcal{J}(i) \wedge \left(\bigwedge_{k=1}^{n} r'_k = f_k(i,r)\right)\right) \tag{3.3}$$

Une égalité de la forme $a{=}b$ pouvant être exprimée comme $a{\cdot}b + \overline{a}{\cdot}\overline{b}$, la formule (3.3) est expansée en interne par :

$$\mathrm{Img}(f,\chi) = \lambda r'.\left(\exists r,\, i\,.\,\chi(r) \wedge \mathcal{J}(i) \wedge \left(\bigwedge_{k=1}^{n} r'_k{\cdot}f_k(i,r) + \overline{r'_k}{\cdot}\overline{f_k(i,r)}\right)\right) \tag{3.4}$$

Si l'on utilise une logique trivaluée, nous ne pouvons plus simplement remplacer f_k par f_k^1 et $\overline{f_k}$ par f_k^0 : nous n'avons pas $f_k^1 \vee f_k^0$, à l'inverse de $f_k \vee \overline{f_k}$, comme le représente la Figure 3.4. A l'opposé d'une partition $\left(f, \overline{f}\right)$, nous avons désormais trois ensembles f^0, f^1, et $f^d = \overline{f^0 + f^1}$, ce dernier correspondant aux valuations pour lesquelles nous savons uniquement que f est définie, sans en connaître la valeur exacte. Dès lors, nous devons *élargir* la fonction positive f par $\overline{f^0}$, et la fonction négative $\overline{f}$ par $\overline{f^1}$. Intuitivement, cela revient à prendre aussi en compte l'ensemble f^d là où l'on ne prenait en compte que f^0 ou f^1.

Nous introduisons l'opérateur OImg (*Over-approximated Image*) comme l'*élargissement* de l'opérateur standard de calcul d'image Img :

$$\mathrm{OImg}(f,\chi) = \lambda r'.\left(\exists r,\, i\,.\,\chi(r) \wedge \mathcal{J}(i) \wedge \left(\bigwedge_{k=1}^{n} r'_k{\cdot}\overline{f_k^0(i,r)} + \overline{r'_k}{\cdot}\overline{f_k^1(i,r)}\right)\right) \tag{3.5}$$

De manière informelle, nous avons remplacé la fonction caractéristique de l'ensemble "sur lequel f est vraie" (le *onset* de f), par un sur-ensemble "sur lequel f n'est certainement pas fausse", et *vice versa* (Figure 3.5). Notons que lorsque l'on applique cet opérateur élargi sur une variable non abstraite de la forme $(x, \overline{x})$, les *onsets* de f^0 et f^1 forment une partition du domaine sur lequel f est définie, et le résultat de l'opérateur OImg demeure exact.

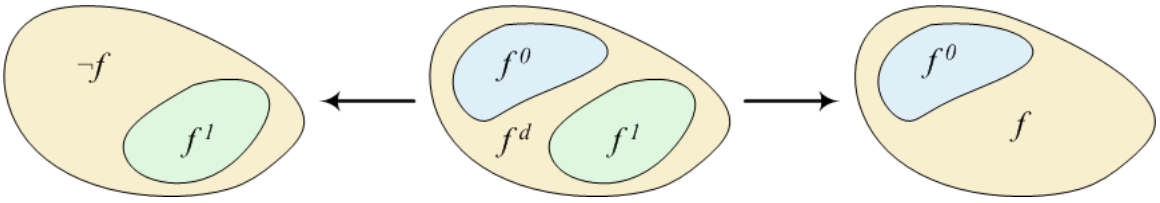


Figure 3.5: Elargissement de la fonction trivaluée f

L'opérateur OImg est croissant sur le treillis complet de l'espace d'états. L'algorithme termine donc avec un plus petit point fixe unique.

[1] $\lambda r'.E$ est une notation standard du λ-calcul pour la fonction anonyme de corps E et d'argument r'.

Exemple

Appliquons la technique d'abstraction de variables à l'aide d'une logique trivaluée au circuit de la Figure 2.9 (p. 43). Toujours en cherchant à prouver que $r_1 \wedge r_2 = 0$, nous pouvons abstraire r_4 : à partir de l'état initial $(r_1, r_2, r_3) = (1, 0, 0)$, la première itération découvre l'état $(0, 1, 0)$, la deuxième itération découvre l'état $(0, 0, 1)$ et la troisième itération atteint le point fixe. En ayant abstrait r_4, nous pouvons toujours prouver que $r_1 \wedge r_2 = 0$ mais avec moins de fonctions à construire, moins de variables à substituer, mais surtout moins de variables à quantifier existentiellement.

Inversement, si nous choisissons d'abstraire r_3, à partir de l'état initial $(r_1, r_2, r_4) = (1, 0, 0)$, la première itération découvre les états $(0, 1, 0)$ et $(1, 1, 0)$, la deuxième itération découvre l'état $(0, 0, 0)$ et la troisième itération atteint le point fixe. Du fait d'une surapproximation excessive due à un mauvais choix des variables à abstraire, nous échouons à prouver que $r_1 \wedge r_2 = 0$.

Discussion

Comme pour la technique de remplacement de variables d'état par des entrées libres, l'abstraction de variables réduit :

- le nombre de fonctions de registres à construire ;
- le nombre de fonctions de registres à manipuler durant les calculs d'images ;
- le nombre de variables à substituer.

Surtout, l'abstraction de variables réduit le nombre de variables à quantifier existentiellement *a posteriori*. L'utilisation d'une troisième valeur d nous permet la quantification existentielle de variables *a priori*. Ainsi, la formule $\exists\, x,\, y\,.\, f(x, y)$ devient $f(d, 0) + f(d, 1)$ lorsque la variable x est abstraite, à la place de $f(0, 0) + f(0, 1) + f(1, 0) + f(1, 1)$. Certes, si les variables ainsi abstraites n'intervenaient réellement pas dans les calculs, elles auraient disparues des BDDs, mais *durant* leur construction ou leur combinaison. Notre technique les fait disparaître *avant tout calcul*.

Précisons que, si cette technique nécessite la construction de deux fonctions de transition f^0 et f^1 pour chaque registre, cette étape du calcul d'espace d'états atteignables est loin d'être critique en pratique. De plus, les paires de fonctions (f^0, f^1) ne comportant pas de variable abstraite sont opposées : $f^0 = \overline{f^1}$. De telles fonctions correspondent donc à un seul et même BDD. Dès lors, les résultats de tous les calculs sur les BDDs étant conservés dans des structures de mémoire cache, les "doubles" calculs faisant intervenir f^0 et $f^1 = \overline{f^0}$ ne sont en fait effectués qu'une seule fois.

Comme la technique précédente, l'abstraction de variables relâche les contraintes entre les variables abstraites, ce qui conduit à calculer une approximation par excès de l'espace d'états atteignables. De plus, l'abstraction de variables ajoute deux nouvelles sources d'approximation par excès :

- Nous procédons à un *élargissement* de la fonction de transition lorsque nous remplaçons la fonction positive f par $\overline{f^0}$, et la fonction négative $\overline{f}$ par $\overline{f^1}$.

– La corrélation entre les différentes occurrences des variables abstraites dans les équations du circuit est perdue. En revenant à l'exemple du test de la Figure 3.3 (p. 52), abstraire le signal d'entrée I fait perdre l'information que les branches *then* et *else* sont exclusives. En effet, les expressions comme $x \cdot \overline{x}$ sont abstraites en $d \cdot \overline{d} = d$, et non 0.

Du fait du renforcement de l'approximation, l'effet "boule de neige" peut être accru (des états non atteignables sont trouvés à tort atteignables, ces états conduisent à leur tour à des états qui peuvent ne pas être réellement atteignables, etc.). Les variables à abstraire doivent donc être choisies avec encore plus de soin. Nous verrons en 3.4 (p. 61) une technique permettant de réduire cette approximation, en utilisant les informations fournies par le compilateur `Esterel` sur la structure interne des modèles.

3.3.3 Implémentation et expérimentations

Les deux techniques que nous venons de voir —le remplacement de variables d'états par des entrées libres et l'abstraction de variables à l'aide d'une logique trivaluée— sont implémentées de manière stable dans la librairie `TiGeREnh` et rendues disponibles par l'outil `evcl`.

Néanmoins, l'implémentation que nous avons réalisée des calculs d'espaces d'états atteignables à l'aide d'une logique trivaluée n'est pas totalement optimisée. En résumé, nous implémentons l'Algorithme 2.1 (p. 40) pratiquement tel quel. Nous ne partitionnons notamment pas les calculs d'images. Le faire permettrait de pouvoir réellement comparer les deux techniques sur un pied d'égalité. Néanmoins, nos expériences —détaillées en 6.4 (p. 118) et en 6.5 (p. 125)— exhibent déjà des cas où l'abstraction de variables à l'aide d'une logique trivaluée améliore les performances de la technique usuelle de remplacement de variables d'états par des entrées libres.

En pratique, les conseils d'abstraction donnés par les développeurs ont permis de diviser les temps de calculs par des facteurs allant jusqu'à 300 en réduisant la mémoire consommée par des facteurs allant jusqu'à 50. Les variables à abstraire étant correctement choisies, l'éventuelle approximation —difficile à quantifier— demeurait largement tolérable puisque les propriétés ont pu être vérifiées en grande majorité.

Enfin, nous avons constaté que l'effet "boule de neige" a pour conséquence positive que les propriétés sont très rapidement trouvées violables à tort lorsque les variables à abstraire sont mal choisies ou que le modèle ne se prête pas à l'abstraction de variables. Ainsi, en cas d'erreur de jugement, les calculs terminent très rapidement et en n'ayant consommé que très peu de mémoire.

Cela nous porte à conclure que l'abstraction de variables, si elle ne se prête qu'à certains types de modèles et nécessite une bonne connaissance des modèles à abstraire, ne présente guère d'inconvénients à être tentée, sachant qu'elle peut permettre d'accélérer les calculs et de réduire la consommation mémoire de plusieurs ordres de grandeur.

3.3.4 Automatisation de l'abstraction

Dans notre outil, la sélection de variables d'état à remplacer par des entrées libres ou à abstraire doit être effectuée par le développeur du modèle. Ces techniques nécessitent en effet une bonne connaissance du modèle. En pratique, la sélection se fait souvent très rapidement, en énumérant les différents modules composant le modèle et en estimant si certains de ces modules sont influents ou non.

Pour chacune de nos expériences, les développeurs ont su, en quelques minutes seulement, discriminer les modules estimés influents. Si un module est estimé non influent, toutes les variables d'état qui le composent sont alors abstraites. Notre outil propose aussi un niveau de granularité beaucoup plus fin, puisqu'il est possible d'abstraire des variables d'états individuelles, voire directement des portes quelconques. En pratique, l'abstraction de modules complets s'est avérée être un niveau de granularité suffisant.

Dans le cadre d'un environnement de développement intégré comme `Esterel Studio`, qui propose notamment une vue arborescente des modules constituant un modèle, l'abstraction de variables pourrait ainsi être rendue facilement accessible *via* une interface graphique. Néanmoins, proposer des stratégies d'automatisation de l'abstraction serait une amélioration intéressante pour l'utilisateur.

Une approche raisonnable visant à automatiser l'abstraction du modèle à analyser consiste à partir d'un modèle où un grand nombre de variables ont été abstraites. Sur un tel modèle abstrait, les calculs d'espaces d'états devraient être extrêmement rapides, mais devraient aussi probablement conduire à des réfutations erronées des propriétés dues à une approximation excessive. L'abstraction initiale doit alors être raffinée en réintégrant progressivement les variables abstraites, jusqu'à permettre la validation (ou la réfutation correcte) des propriétés.

Abstraction basée sur la profondeur des variables d'états

Une première idée pour automatiser la sélection des variables à abstraire est d'utiliser la profondeur des variables d'états. Cette profondeur correspond au nombre de variables d'état sur le plus court chemin entre la variable d'état candidate à l'abstraction et les sorties des propriétés à vérifier. Cette profondeur se calcule pour l'ensemble des variables d'états du circuit par un simple parcours du graphe formant le circuit, soit avec un coût linéaire en fonction du nombre de portes. La profondeur d'une variable d'état correspond donc au "délai" par rapport à l'instant initial avec lequel la variable d'état peut influencer les propriétés à vérifier.

L'idée est alors de tenter de vérifier les propriétés en abstrayant toutes les variables d'état puis, tant que les propriétés n'ont pas été validées (ou correctement invalidées avec génération de contre-exemples corrects), de réintégrer progressivement dans les calculs les variables d'état abstraites, en fonction de leur profondeur.

Malheureusement, nous avons constaté que sur la plupart des circuits générés par le compilateur `Esterel` v5, la majorité des variables d'état sont à très faible profondeur. Cela provient essentiellement de la façon dont les continuations des parallélisations de blocs sont

calculées.

La Figure 3.6 présente le sous-circuit généré par la mise en parallèle de trois modules. La porte `sel` [Ber99] indique qu'au moins une des branches du parallèle était sélectionnée. Les portes `dead` indiquent que la branche correspondante avait déjà terminé, c'est-à-dire qu'elle n'était plus sélectionnée alors qu'au moins une autre branche du parallèle l'était. Les portes `min k0` indiquent que la branche correspondante avait déjà terminée ou vient de terminer. Les portes `min k1` indiquent que la branche correspondante avait déjà terminé, vient de terminer ou est en pause. Les portes ∪`k0` indiquent qu'au moins une des branche vient de terminer. Les portes ∪`k1` indiquent qu'au moins une des branches est en pause. Un parallèle termine donc (porte `k0`) lorsqu'au moins une des branches vient de terminer et que toutes les autres avaient déjà terminé ou terminent également.

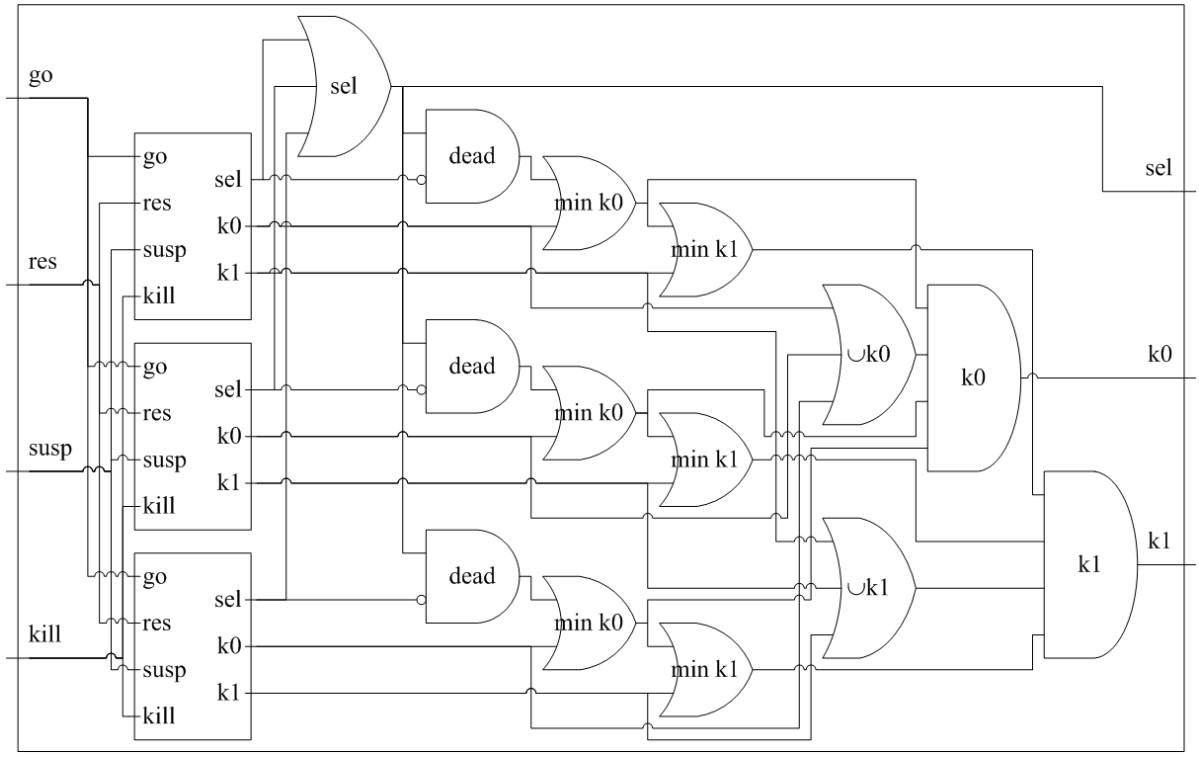


Figure 3.6: Circuit généré pour les constructions parallèles (3 composants)

Le calcul de la porte `k0` référence donc tout l'arbre de sélection des branches du parallèle, donc tous les registres de pause qu'il contient. Notre analyse de profondeur ne permet donc guère de distinguer les variables d'états entre elles, *a fortiori* lorsque le modèle est une mise en parallèle de modules au plus haut niveau, ce qui est assez fréquent.

Pour `Esterel` v5_9x, un simplifieur de l'arbre de sélection, `scsimplify` (cf. 2.5, p. 32), a été développé par Dumitru Potop-Butucaru [PB01]. Cet outil s'efforce de simplifier l'arbre lorsque certaines branches ne terminent jamais (hors préemption) ou lorsque des branches de l'arbre sont en exclusion mutuelle (séquencement). Ce simplifieur a surtout pour effet de réduire le nombre de registres du circuit –ce qui présente un intérêt non négligeable–

mais il n'augmente guère la profondeur du circuit.

Les simplifications apportées par `scsimplify` sont ou seront intégrées au compilateur v7. L'automatisation de l'abstraction à partir des niveaux de profondeur des registres sera donc à reconsidérer lorsque le compilateur v7 aura atteint sa maturité...

Analyse des réfutations erronées

Les techniques de remplacement de variables d'états par des entrées libres ou d'abstraction de variables à l'aide d'une logique trivaluée calculent des sur-approximations de l'espace d'états car elles opèrent sur un modèle simplifié. La confrontation des propriétés énoncées par l'utilisateur avec des espaces d'états sur-approchés peut conduire à réfuter ces propriétés sur le modèle abstrait à une certaine profondeur alors qu'elles sont tout à fait valides sur le modèle concret à cette profondeur.

Les réfutations erronées (*false negatives*) se détectent très simplement lors de la génération de contre-exemples menant à la violation des propriétés. Ces contre-exemples sont générés sur le modèle concret par calculs d'images inverses des fonctions de transitions. Il s'agit alors de trouver des séquences d'entrées qui, à partir de l'ensemble d'états atteignables découverts à la profondeur précédente, mènent à des états violant les propriétés. Si de telles séquences, envisageables sur le modèle abstrait, ne peuvent être trouvées sur le modèle concret, alors les calculs d'images inversent échoueront à un moment ou à un autre.

A partir de là, plusieurs attitudes sont envisageables. L'attitude minimale consiste à poursuivre l'exploration des espaces d'états atteignables dans le but de déterminer si les propriétés peuvent réellement être violées à une profondeur plus importante. Il est également envisageable de mettre à jour les espaces d'états atteignables en supprimant les états qui n'ont pas été confirmés comme réellement accessibles sur le modèle concret lors des calculs d'images inverses. Cette option vise à limiter l'effet "boule de neige" en supprimant des sur-approximations construites les états qui ne sont pas réellement accessibles. Cette option peut néanmoins impliquer un coût prohibitif de mise à jour en cascade des ensembles d'états atteignables. Nous n'avons pas évalué cette option.

Dans le cadre d'une abstraction automatisée du modèle, la détection de réfutations erronées dues à une abstraction excessive peut conduire à un raffinement de l'abstraction visant à éliminer les réfutations erronées. Il existe dans la littérature de nombreux travaux proposant diverses stratégies de raffinement des abstractions par analyse des contre-exemples erronés [CGJ+00, GD00, WHL+01]. Ces différentes méthodes de raffinement de l'abstraction nécessitant au préalable d'automatiser celle-ci, nous ne les avons pas étudiées plus profondément.

3.3.5 Travaux connexes

Dans le cadre des calculs d'approximations par excès de l'espace d'états atteignables d'un modèle, une des approches connexes à la nôtre est basée sur la décomposition du modèle. Dans [CHM+93], Cho *et al.* proposent un algorithme qui décompose l'ensemble des

variables d'états en sous-ensembles disjoints. Chaque sous-ensemble est utilisé pour calculer une partie de l'espace d'états atteignables, le résultat final étant le produit cartésien de ces parties d'espaces d'états atteignables. Ces travaux ont été étendus aux sous-ensembles de variables d'états non disjoints par Govindaraju *et al.* dans [GDHH98], puis raffiné dans [GDB00] par l'ajout de variables d'état auxiliaires permettant d'accroître la corrélation entre les sous-ensembles de variables d'états. L'extensions aux calculs d'image inverses est proposé dans [GD98]. Le raffinement des choix de partitionnement basé sur l'analyse des contre-exemples erronés est proposée dans [GD00].

Ces travaux remplacent par des entrées libres les variables d'états ne faisant pas partie des sous-ensembles de variables d'états considérés : il serait possible d'appliquer à ces travaux notre technique à base de logique trivaluée.

3.4 Utilisation des informations structurelles sur les modèles

La plupart des langages de haut-niveau offrent la possibilité de décomposer les programmes en modules, alors utilisés de manière hiérarchique. Avec le langage `Esterel`, la hiérarchisation apportée par les modules est encore renforcée par la hiérarchisation des constructions du langage, comme le montre par exemple le Programme 2.2 (p. 22).

La structure hiérarchique des modules et des constructions du langages se retrouve dans l'arbre de sélection (cf. 2.3.5, p. 28), qui indique la hiérarchie des registres de pause, ou variables d'états. Ces registres de pause sont générés explicitement par l'instruction **`pause`** et implicitement par les instructions dérivées l'utilisant (**`halt`**, **`await`**, etc.). L'arbre de sélection indique des compatibilités (exécution en parallèle) ou des exclusions (exécution en séquence) entre registres de pause ou grappes de registres de pause.

Nous proposons ici quelques méthodes tirant partie de ces informations structurelles de haut-niveau, avec toujours comme objectif la réduction des coûts de calculs d'espace d'états atteignables.

3.4.1 Approximation syntaxique de l'espace d'états accessibles

A partir de l'arbre de sélection du programme, nous pouvons construire un BDD qui exprime une approximation par excès de l'espace d'états atteignables du circuit. Les états déclarés non atteignables par ce BDD le sont effectivement *par construction*. La Figure 3.7 présente un tel BDD, correspondant au Programme 2.6.

La construction d'un tel BDD se fait en remontée d'un simple parcours en profondeur de l'arbre de sélection. En utilisant un ordre de variable cohérent avec la structure des registres dans le programme initial, le temps de construction du BDD et sa taille finale sont négligeables.

Les propriétés ne dépendant que du fait que certaines parties de programmes ne puissent être actives en même temps *par construction* pourraient ainsi être vérifiée à moindre frais, en évitant un calcul d'espace d'états coûteux. Notre outil de vérification formelle propose

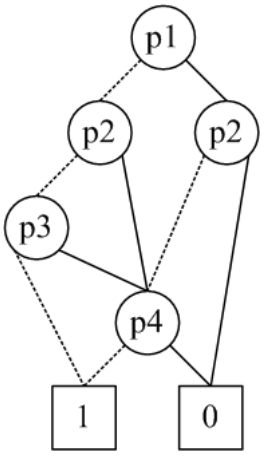


Figure 3.7: BDD de l'arbre de sélection du Programme 2.6

une telle fonctionnalité. Néanmoins, nous devons reconnaître n'avoir jamais rencontré de telles propriétés sur les modèles que nous avons analysés.

3.4.2 Renforcement des relations entre variables remplacées par des entrées libres

Les relations exprimées par l'arbre de sélection peuvent être utilisées pour renforcer les contraintes entre les variables d'état qui ont été remplacées par des entrées libres. Ces relations enrichissent alors l'*input care set* ($\mathcal{J}$) spécifié par le développeur, qui décrit des contraintes entre les entrées du modèle (cf. 2.7, p. 35).

Cette technique vise deux objectifs :

1. limiter la sur-approximation due au relâchement des contraintes entre les variables d'états remplacées par des entrées libres en réintégrant les contraintes les plus simples (les exclusions provenant des séquencements) ; limiter la sur-approximation revient à réduire le cardinal de l'espace d'états atteignables et peut rendre vérifiable une propriété qui ne l'était sur un espace d'états excessivement sur-approché ;
2. réduire le nombre de nœuds des BDDs en restreignant ceux-ci à un domaine plus strict (*via* l'opérateur `Restrict` [CM90]).

L'*input care set* pouvant référencer à la fois des variables d'états et des entrées, nous pouvons donc y intégrer toutes les relations spécifiées par l'arbre de sélection qui référencent au moins une variable d'état remplacée par une entrée libre.

En pratique, les bénéfices apportés par cette technique ne sont pas généralisables. La réduction de la sur-approximation est notable dans de nombreux cas, mais elle s'effectue souvent au prix d'un ralentissement des calculs et d'une augmentation de la consommation mémoire (cf. Expériences 6.4, p. 118 et 6.5, p. 125).

3.4.3 Borne supérieure d'approximation

Les relations exprimées par l'arbre de sélection peuvent aussi être utilisées comme *borne supérieure* de l'espace d'états atteignables. A la fin de chaque étape du calcul d'espace d'états atteignables, nous ne conservons alors que l'intersection entre les nouveaux états découverts et le BDD d'approximation dérivé de l'arbre de sélection. Une partie des états découverts atteignables à tort sont donc automatiquement supprimés, ce qui réduit l'effet "boule de neige", donc la sur-approximation.

Les variables abstraites à l'aide d'une logique trivaluée disparaissent complètement en tant que variables. Elles ne peuvent donc pas apparaître dans un BDD quel qu'il soit. De même, les variables d'états remplacées par des entrées libres ne peuvent apparaître dans le BDD bornant l'espaces d'états atteignables, qui ne peut référencer que des variables d'états. Nous ne construisons donc le BDD de borne supérieure de l'espace d'états atteignables qu'à partir des relations de l'arbre de sélection restreintes aux variables d'états conservées en tant que telles.

En pratique, les bénéfices apportés par cette technique sont déjà plus notables. La réduction des temps de calculs et de consommation mémoire sont souvent significatifs. Le cardinal des espaces d'états atteignables est souvent réduit d'un ou plusieurs ordres de grandeur, ce qui permet parfois de rendre certaines propriétés vérifiables (cf. Expériences 6.4, p. 118 et 6.5, p. 125).

3.4.4 Stratégies d'ordonnancement des variables

Construire des BDDs dérivés de l'arbre de sélection pose le problème du choix d'un ordonnancement de variables. Plusieurs approches sont possibles :

1. utiliser le même ordonnancement de variables —calculé à partir d'heuristiques basées sur la topologie du circuit— à la fois pour la construction des BDDs dérivés de l'arbre de sélection et pour le calcul d'espace d'états atteignables ; cette stratégie semble être pénalisante pour la construction des BDDs dérivés de l'arbre de sélection : les temps de construction ainsi que le nombre de nœuds de BDDs construits semblent souvent ne pas être optimaux ;
2. appliquer un ordonnancement des variables cohérent avec la structure des registres dans le programme initial puis utiliser ce même ordonnancement pour le calcul d'espace d'états atteignables ; cette stratégie permet de construire des BDDs dérivés de l'arbre de sélection de taille très compacte et en temps négligeable, mais cet ordonnancement ne semble pas favoriser l'efficacité des calculs d'espaces d'états ;
3. utiliser un premier ordonnancement de variables cohérent avec la structure des registres, le temps de construire les BDDs dérivés de l'arbre de sélection, puis mettre à jour ces BDDs afin qu'ils respectent l'ordonnancement des variables calculé à partir d'heuristiques basées sur la topologie du circuit ; cette stratégie présente un risque d'explosion en temps et/ou en mémoire précisemment lors du réordonnancement des BDDs dérivés de l'arbre de sélection.

Après diverses expériences, l'approche que nous avons retenue est cette dernière qui, en pratique, semble offrir le meilleur compromis quant aux temps de calculs et à la consommation mémoire. Les cas d'explosion en temps et/ou en mémoire lors du réordonnancement des BDDs semblent rares en pratique. Dans tous les cas, un comportement suspect lors de cette phase, qui pourrait empêcher l'obtention de résultats, est aisément détectable par l'utilisateur.

3.5 Représentation interne des circuits

Le package de BDDs que nous utilisons, `TiGeR` [CMT93], calcule l'espace d'états accessibles de circuits logiques représentés par des structures `C` qui lui sont propres et forment un `Tgr_Network`. Selon le type de circuit donné en entrée, notre outil propose trois différents flots de construction de `Tgr_Network`s (Figure 3.8).

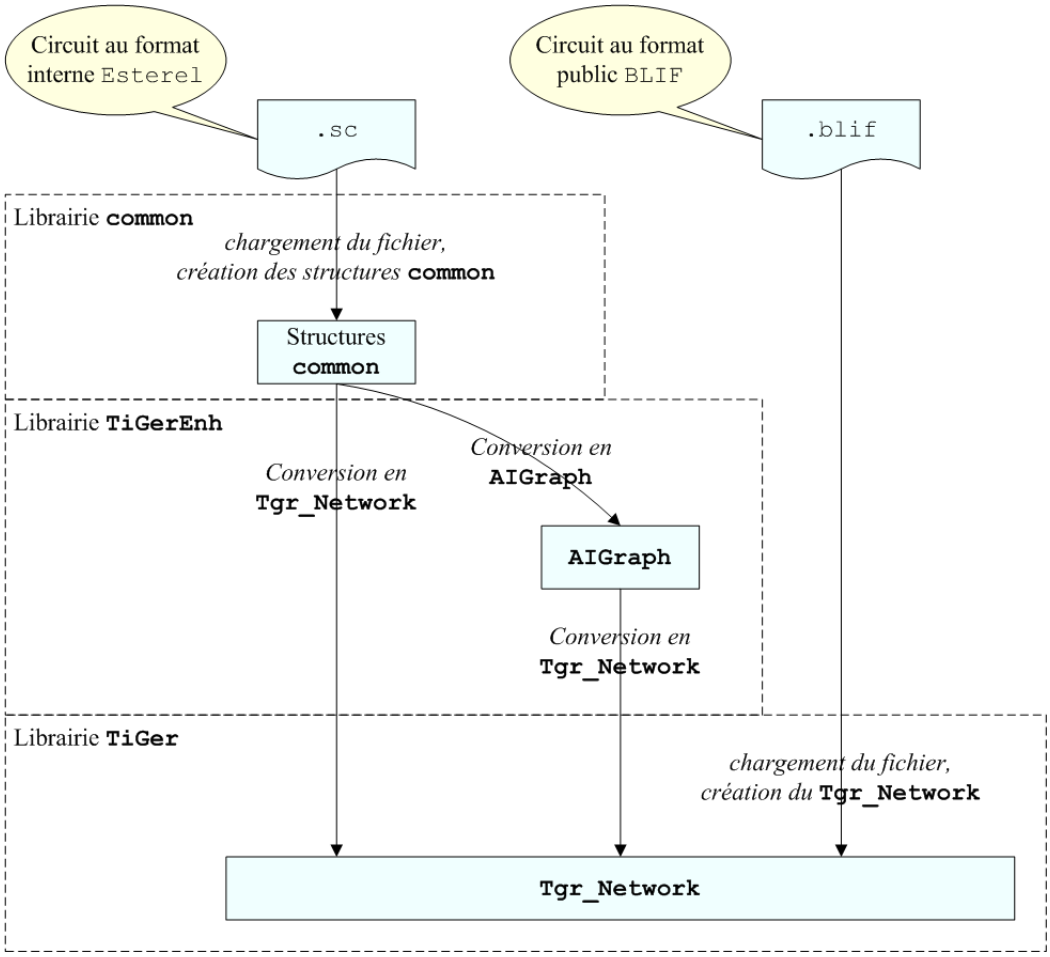


Figure 3.8: Flots de création de `Tgr_Network`s

3.5.1 Construction directe de `Tgr_Network`s à partir de circuits au format `BLIF`

La première méthode, utilisée par `Xeve`, consiste à se reposer sur une fonction fournie par le package `TiGeR` et permettant le chargement direct d'un circuit au format `BLIF` ainsi que sa conversion en `Tgr_Network`. Les circuits au format `BLIF` sont dérivés des circuits `Esterel` (cf. Figure 2.5, p. 30), mais ils n'en conservent que la logique et perdent toute information structurelle sur les modèles.

3.5.2 Construction de `Tgr_Network`s à partir de circuits au format interne `sc`

La deuxième méthode consiste à travailler directement sur les circuits `Esterel`, afin justement de permettre l'utilisation de ces informations structurelles (cf. 3.4, p. 61). Les circuits `Esterel`, au format interne `sc`, sont d'abord chargés par la librairie `common`, librairie commune à différents processeurs et dédiée au chargement des fichiers intermédiaires du compilateur. Ces circuits sont alors convertis en `Tgr_Network` par notre librairie `TiGeREnh`.

Les `Tgr_Network`s sont des graphes acycliques dont les nœuds sont des *PLAs* (*Programmable Logic Array*), plus précisément des fonctions combinatoires en forme normale (somme de produits). A chaque nœud d'un `Tgr_Network` sont associés plusieurs variables. Le nombre de variables disponibles est limité : sur des architectures 32 bits, les variables de BDDs sont stockées dans les 16 premiers bits du premier mot de la structure correspondant à un nœud de BDD (Figure 3.9). Cela limite donc le nombre de variables disponibles à 65 536 et à encore moins le nombre maximal de nœuds dans un `Tgr_Network`.

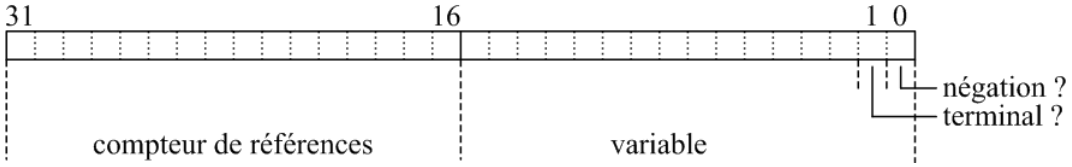


Figure 3.9: Têtes de nœuds de BDDs

Les autres bits sont utilisés pour stocker le nombre de références faites à ce nœud (pour le *Garbage Collector*) et par des drapeaux indiquant si le nœud est terminal ou s'il est inversé. Pour augmenter le nombre de variables disponibles, il est donc possible de réduire l'espace utilisé pour stocker les compteurs de références, ce qui en limite alors la valeur maximale. Lorsqu'un compteur de références atteint la valeur maximale, le nœud correspondant devient *immortel* : la mémoire qu'il occupe et, par transitivité, tous les nœuds de BDDs référencés à partir de celui-ci ne peuvent alors plus être récupérés par le *Garbage Collector*. Enfin, la taille d'un grand nombre de structures (essentiellement des tables de correspondances) dépend du nombre maximal de variables. Augmenter le nombre de variables disponibles se fait donc au prix d'un risque notable d'augmentation de la consommation mémoire.

L'approche de `Xeve`, qui consiste à construire directement le `Tgr_Network` à partir du circuit `BLIF` puis à effectuer le *sweeping* (à partir de la version v5_94), consomme donc inutilement des variables : les variables des nœuds supprimés ne sont pas récupérées. La construction de `Tgr_Network` à partir de circuit `sc` nous permet de réduire la consommation de variables car nous effectuons le *sweeping* à la volée, en construisant le `Tgr_Network` à partir des sorties des propriétés à vérifier.

Sur certains gros modèles, il était nécessaire d'utiliser une version spéciale de `Xeve`, compilée pour permettre plus de variables au détriment donc du seuil d'immortalité des nœuds de BDDs, alors que notre outil traitait le circuit en version standard.

3.5.3 Représentation intermédiaire du circuit par un AIGraph

Les diverses analyses de circuits que nous avons mises en œuvre nécessitent d'associer, parfois temporairement, des structures de données spécifiques aux différents nœuds du circuit analysé.

L'implémentation de ces analyses par extension des structures formant un `Tgr_Network` n'était guère pratique :

- Dans `TiGeR`, l'association de données temporaires à des nœuds de `Tgr_Network`s se fait *via* des tables d'adressages dispersées qui maintiennent les associations. Cette approche, si elle résout de manière assez correcte le besoin d'associer *temporairement* des données aux nœuds du graphe est intrinsèquement lourde.
- `TiGeR` est écrit en C. Même si cela a été fait avec une approche objet très forte et que le résultat est d'une rare qualité, les contortions nécessaires pour parvenir à encapsuler les données, la perte de typage statique fort (auquel nous tenons dans nos développements) et les transtypages (*casts*) incessants que cela requiert sont lourds lorsqu'on a la possibilité de faire mieux, en l'occurrence en `C++`.
- Enfin, si les PLAs des nœuds de `Tgr_Network`s sont très versatiles, puisqu'il permettent d'exprimer toutes sortes de fonctions combinatoires, leur manipulation n'en est pas moins complexe.

De manière similaire, il n'était pas envisageable de développer nos analyses en utilisant directement les structures de données que la librairie `common` instancie à partir des circuits au format `sc`. En effet, cette librairie ne facilite pas plus, voire rend difficile, l'association de données aux nœuds du graphe représentant le circuit[2].

Nous avons donc choisi de développer nos analyses de circuits sur une nouvelle structure de graphe. Bien qu'un quelconque graphe de portes logiques soit satisfaisant, dès lors que l'on en a la maîtrise de l'implémentation, nous nous sommes orientés vers l'utilisation d'un And/Inverter Graph, ou AIGraph, proposé par Andreas Kuehlmann *et al.* [KK97, GK00, PK00, KGP01]. Cette structure de données est très simple à manipuler, permet aisément d'implémenter des analyses de graphes et peut permettre une compression de circuits à la volée.

[2]La librairie `common` a depuis été réécrite en v7, avec notamment la volonté de permettre l'extension des classes instanciées à partir des fichiers intermédiaires du compilateur.

Les nœuds d'un AIGraph sont restreints à des portes `et` binaires. Les références à ces nœuds peuvent être inversées par marquage du bit de poids faible du pointeur, à l'instar de ce qui est fait dans la plupart des librairies de BDDs. La conversion de nœuds arbitraires de circuits logiques en de telles portes `et` binaires se fait trivialement, à la volée lors de la construction du circuit, par application des lois de De Morgan. Cette conversion de portes n-aires en portes binaires n'est certes pas linéaire (elle est d'ordre $n \cdot \log(n)$), mais la majeure partie des portes générées par le compilateur `Esterel` sont binaires. Nous n'attendons donc pas d'augmentation notable du nombre de portes suite à la mise en œuvre de cette technique.

Similairement à ce qui est fait dans les librairies de BDDs, les nœuds sont stockés dans une table d'adressage dispersé. Le code de hachage est calculé à partir des identificateurs des nœuds fils et de leur attribut de négation. La table d'adressage dispersé est utilisée pour détecter —à coût pratiquement nul— les nœuds identiques, donc les sous-circuits isomorphes. Ces sous-circuits isomorphes étant fonctionnellement équivalents, ils sont immédiatement fusionnés. La canonicité structurelle des sous-circuits est renforcée par leur restructuration jusqu'au second niveau (nœuds petit-fils), ce qui augmente encore la détection de sous-circuits isomorphes [GK00] sans surcoût notable.

Cette structure de données avait initialement été utilisée par Andreas Kuehlmann *et al.* pour la vérification formelle d'équivalence de machines (*equivalence checking*). Dans ce contexte, on vise justement à détecter des sous-circuits fonctionnellement équivalents, ce qui simplifie le *miter* analysé (cf. 2.10.3, p. 45). Par exemple, dans [PK00, KGP01], l'apport de l'AIGraph comme structure de base est renforcé par l'alternance de passes de *BDD sweeping* —dans lesquelles on se sert de la représentation canonique des fonctions qu'offrent les BDDs pour identifier des sous-circuits fonctionnellement équivalents— et de passes de *SAT solver* —pour vérifier l'équivalence de certaines portes.

En ce qui nous concerne, la vérification formelle d'équivalence de machines ne représente par un domaine significatif d'application de nos outils. Nous ne nous attendons donc pas à travailler sur des circuits redondants par construction. Néanmoins, les circuits générés de manière automatique ont souvent une certaine quantité de nœuds redondants [GK00]. Les circuits générés par le compilateur `Esterel`, *a fortiori* s'ils proviennent de programmes `SyncCharts` ou autres, n'échappent probablement pas à la règle. De plus, la section suivante présente une fonctionnalité qui peut conduire à des registres redondants. Ces registres redondants seront automatiquement fusionnés —à coût pratiquement nul— par l'AIGraph.

La compression à la volée des circuits analysés n'était donc pas l'objectif recherché, d'autant plus que l'AIGraph, dans les travaux d'Andreas Kuehlmann *et al.*, est censé être utilisé en conjonction avec d'autres techniques permettant de compresser les circuits. Nos expériences semblent cependant indiquer une légère simplification du circuit, ce qui a un impact sur la consommation de variables de BDDs.

Nous avions donc choisi l'AIGraph essentiellement pour sa simplicité : c'est une structure de données très versatile qui facilite l'implémentation de la plupart des analyses de circuits. Nous envisageons de généraliser son utilisation à nos outils d'analyse explicite ou hybride implicite/explicite de circuits (Chapitres 4 et 5), ce qui permettrait de simplifier

les algorithmes de stabilisation des circuits qui y sont mis en œuvre.

3.6 Vérification formelle en boîte blanche ou noire

La vérification formelle par observateurs [HLR93] (cf. 2.10, p. 44) consiste à exécuter, en parallèle du modèle à vérifier, un ou des modules dont les entrées sont celles du modèle observé ainsi que ses sorties, dont la correction est contrôlée.

Initialement, l'instanciation des observateurs était à la charge de l'utilisateur. Les observateurs pouvaient tout autant être exécutés en parallèle au plus haut niveau du modèle observé ou plus profondément dans l'arbre d'instanciation des modules. Les signaux émis par l'observateur devaient toutefois faire partie de l'interface du modèle, de façon à être visibles par le vérifieur. Cette méthodologie de vérification est dite *en boîte blanche* : l'observateur peut avoir accès aux entrées et aux sorties primaires du modèle observé, mais aussi à toute donnée (signaux locaux, variables, etc.) visibles dans sa portée d'instanciation. L'approche en boîte blanche amène par contre l'inconvénient que le modèle doit être compilé différemment selon qu'il est destiné à la production ou à la vérification formelle, autrement dit s'il doit intégrer ou non les observateurs.

Comme nous l'avons vu en 3.3.4 (p. 58), le câblage des modules exécutés en parallèle a tendance à lier les supports des modules parallélisés. Or, la vérification formelle par observateur revient généralement à des parallélisations au plus haut niveau. Dans le cas où l'observateur ne vérifie en fait qu'une petite partie du circuit, il se peut que le *sweeping* soit sans effet et que l'on doive calculer l'espace d'états atteignables du circuit avec tous ses registres.

Pour s'affranchir de ce problème, nous proposons de compiler les observateurs séparément, l'édition de liens étant alors effectuée par le vérifieur formel et non plus par le compilateur `Esterel`. Cette approche est dite *en boîte noire* parce que l'observateur n'a accès qu'aux entrées et aux sorties primaires du modèle observé. Elle permet d'accroître le champs d'action du *sweeping* et de ne pas nécessiter différentes compilations selon que le modèle doit être vérifié ou produit.

Enfin, si les observateurs ont des structures d'implémentation similaires (des boucles d'attentes de signaux par exemple), ils auront probablement des registres de pauses équivalents entre eux, mais aussi équivalents à certains registres de pause du modèle observé. La structure d'AIGraph vue à la section précédente permet de fusionner ces registres redondants à coût pratiquement nul.

3.7 *Bounded Model Checking*

Le *Bounded Model Checking* consiste à ne calculer l'espace d'états atteignables du modèle que jusqu'à une certaine profondeur et, en conséquence, à ne vérifier le modèle que jusqu'à cette profondeur. Cette technique est utile pour permettre, notamment au début des développements, de limiter la profondeur d'analyse pour en diminuer le coût : on ne

cherche alors qu'à vérifier la surface du modèle, ce qui permet de détecter les erreurs les plus facilement atteignables. Cette technique peut être obligatoire pour les *SAT solvers* qui ne savent pas découvrir par eux-même le diamètre du modèle, ou qui ne savent le faire qu'avec un surcoût encore inadmissible.

Notre outil propose une telle fonctionnalité. Dès lors, il semblerait intéressant d'essayer de tirer partie de la limitation de profondeur d'analyse pour réduire le coût de celle-ci. A la manière du *sweeping* (cf. 3.1.1, p. 49), qui spécialise le circuit analysé en fonction des propriétés à vérifier, nous aimerions spécialiser le circuit en fonction de la profondeur considérée. Cela revient à faire disparaître les registres qui n'interviennent pas dans les niveaux de profondeur considérés et ne deviennent actifs que dans les instants ultérieurs, et à les remplacer par une constante.

Malheureusement, comme nous l'avons vu en 3.3.4 (p. 58), la majorité des registres sont à très faible profondeur dans les circuits générés par le compilateur `Esterel` v5. Cette optimisation sera donc à reconsidérer en fonction des améliorations apportées au câblage des circuits par la version v7 du compilateur `Esterel`...

3.8 Ingénierie d'`evcl` et de la librarie `TiGeREnh`

3.8.1 Fournitures d'information supplémentaires à des écouteurs

Le pilotage de la librairie `TiGeREnh` se fait de manière standard par appels de méthodes, les résultats des calculs étant retournés par ces méthodes.

Toutes les informations supplémentaires que la librairie est susceptible de fournir, comme par exemple des détails concernant la progression des calculs, se font selon le principe des *écouteurs* (*listeners*) : tout module intéressé par de telles informations s'enregistre comme écouteur auprès de la librairie en fournissant une instance de la classe d'écouteur adéquate. Les différentes méthodes de cette classe seront invoquées au moment opportun, avec les informations disponibles. Cette technique permet d'accroître de manière notable la versatilité de la librairie, puisqu'elle détache celle-ci de toute nécessité de connaître les modules auquels elle doit communiquer des informations.

Par exemple, `evcl` pilote la librarie par appels de méthodes et transmet à l'utilisateur les résultats demandés par affichage sur la sortie standard ou dans des fichiers (comme pour les contre-exemples). Si l'utilisateur a activé le mode verbeux, différents écouteurs dont la seule tâche consiste à afficher sur la sortie d'erreur standard les informations qui leurs sont fournies sont enregistrés auprès de la librairie. De manière similaire, il est possible d'activer la production de fichiers `XML` permettant de conserver ces mêmes informations. Un script `PHP` trivial permet alors l'extraction de ces données et leur conversion en tableaux au format standard `CSV` (*comma separated values*), directement utilisables par `Excel`™®©☠ ou tout autre tableur/grapheur.

Ce choix d'architecture avait aussi été fait dans le but de permettre le développement rapide d'une interface graphique facilitant l'utilisation de l'outil...

3.8.2 Vérification (informelle) du vérifieur

Les outils de vérification formelle ayant pour objectif de valider ou d'invalider de façon définitive les modèles qui leur sont soumis, il importe de mettre en œuvre tous les moyens raisonnables pour garantir leur correction : il n'est guère concevable qu'il subsiste la moindre part de doute dans les algorithmes utilisés ou leur implémentation.

Notre outil `evcl` et la librairie qu'il pilote, `TiGeREnh`, sont écrit en `C++`. La librairie de BDDs embarquée, `TiGeR`, est écrite en `C`. La vérification formelle de programmes de telle taille, *a fortiori* écrits dans de tels langages génériques, n'est toujours pas envisageable dans un avenir proche. De plus, les outils à base de BDDs, friands d'arithmétique de pointeurs et de manipulations de données au niveau des bits, ne peuvent se contenter d'un sous-ensemble vérifiable du langage. Il existe certes différentes tentatives d'utilisation de prouveurs de théorèmes comme `Coq` [Tea02] pour la vérification d'algorithmes mis en œuvre par exemple dans les outils de vérification formelle [AM00]. Néanmoins, le manque actuel d'automatisation des prouveurs de théorèmes et l'importance de l'investissement requis pour permettre aux prouveurs de théorèmes d'interpréter correctement des structures de données souvent complexes [BMZ02] impliquent que de telles tâches doivent être effectuées dans une thèse séparée...

De ce fait, notre outil de vérification formelle n'est actuellement validé que de manière informelle, selon deux approches complémentaires : par auto-contrôle de l'outil et des librairies et par contrôle externe des résultats produits.

L'auto-contrôle du programme est réalisé selon des paradigmes de programmation défensive usuels, plus précisemment ceux de la programmation par contrat (*Design by Contract*) [Mey88, Mey91], bien que le langage `C++` ne facilite guère la mise en œuvre de tels mécanismes. Nous nous attachons donc, au sein même des implémentations, à vérifier en permanence la validité des données manipulées. En sortie des fonctions, nous nous efforçons autant que faire se peut de contrôler la validité des résultats retournés en vérifiant les propriétés notables que ces données sont censées avoir. Il est alors commun de constater des différences de performances de plusieurs ordres de grandeur entre les versions de nos outils avec ou sans contrôles internes. Ainsi, une exécution de nos outils qui ne lève pas d'assertion est déjà un bon gage de crédibilité des résultats produits.

Cette confiance est renforcée par un banc de test que nous avons réalisé en `PHP`. Ce banc de test se charge d'initier la vérification formelle de différents programmes et de contrôler la validité des résultats produits, dont notamment :

- la terminaison correcte de l'outil ;
- la validité des analyses des propriétés (réfutables ou non) ;
- le réalisme des éventuels contre-exemples ;
- la cohérence des représentations internes des circuits, quelle qu'ait été la méthode de construction (cf. 3.5, p. 64) ;
- la correction du cardinal de l'espace d'états atteignables ;
- la correction du cardinal des ensembles d'états atteignables aux différents niveaux de profondeurs.

Ce banc de test pourrait encore être amélioré en vérifiant les BDDs des espaces d'états

atteignables totaux ou à chaque niveau de profondeur. De même, il serait possible de contrôler que les espaces d'états atteignables générés en remplaçant certaines variables d'états par des entrées libres ou en abstrayant certaines variables à l'aide d'une logique trivaluée soient bien des sur-ensembles de l'espace d'états atteignables du modèle original.

3.9 Génération de séquences de tests exhaustives

La génération de séquences de tests exhaustives permet de contrôler la conformité de deux implémentations à différents niveaux d'abstraction. Par exemple, il est possible de modéliser une application en langage de haut-niveau (comme les `SyncCharts`), pour la vitesse de développement et la lisibilité du résultat, puis d'implémenter cette application en langage de plus bas niveau (comme `VHDL`, `Verilog` ou un langage propriétaire), pour la production. On génère alors à partir de la description de haut-niveau des séquences de tests exhaustives qui couvrent l'ensemble des comportements du modèle. La réaction des deux implémentations à ces séquences de tests doit concorder.

Nous n'avons pas étudié personnellement le problème de la génération de séquences de tests par des techniques purement implicites. Cela a été fait par Amar Bouali, au sein d'Esterel Technologies, et nous présentons l'approche qu'il a choisie. Comme nous le verrons en 5.4 (p. 110), il nous semble préférable d'aborder ce problème à l'aide de techniques hybrides implicites/explicites.

La solution proposée vise la couverture *comportementale* du modèle, selon différents objectifs :

- Couverture d'états : cette couverture garantit que tous les états sont visités au moins une fois par les tests.
- Couverture de transitions entre paires d'états : cette couverture garantit que, s'il existe une transition possible entre deux états, elle sera empruntée au moins une fois par les tests. S'il existe différentes transitions possibles entre deux états (cf. Figure 4.10(a), p. 95), une seule est empruntée.
- Couverture d'outputs : cette couverture garantit que tous les chemins directs — autrement dit sans cycles— conduisant à l'émission d'un certain output sont générés.

Afin de s'assurer que les séquences de tests générées automatiquement demeurent compréhensibles par le développeur, deux types de séquences sont proposées :

- Séquences courtes : la longueur de chaque séquence ne peut être supérieure au diamètre du modèle.
- Séquences longues : les séquences peuvent être plus longues que le diamètre du modèle, mais leur longueur peut être limitée à la demande.

La génération de tests se fait en deux temps :

1. Dans un premier temps, l'espace d'états atteignables du modèle est calculé. Pour le cas spécifique de la couverture de transitions, il est nécessaire de construire le graphe des paires d'états entre lesquels une transition est possible.

2. Dans un second temps, ces graphes sont parcourus en arrière à partir des profondeurs les plus basses afin de générer les séquences d'entrées les couvrant. Les parcours en arrière sont réalisés par des calculs d'images inverses, dont les résultats sont mémorisés dans des structures de mémoire cache.

Dans cette approche de la génération de séquences de tests, les techniques mises en œuvre sont purement symboliques. L'espace d'états atteignables est calculé de manière usuelle comme nous l'avons vu en 2.9 (p. 38). Les états ou les paires de transitions visitées sont stockés dans des BDDs.

En pratique, les coûts en temps et en mémoire de calcul de l'espace d'états atteignables du modèle et la taille des séquences générées limitent la génération de séquences de tests exhaustives aux modèles de taille moyenne, voire à des sous-modules du modèle. Cela est d'autant plus vrai pour la couverture de transitions, qui nécessite de doubler le nombre de variables d'états impliquées dans le calcul d'espace d'états atteignables.

Pour de nombreuses raisons, nous ne pensons pas qu'une approche purement implicite soit appropriée à la génération de séquences de tests exhaustives.

Certes, les calculs d'espaces d'états atteignables sont généralement beaucoup moins coûteux avec les techniques implicites qu'avec les techniques explicites. Cette différence de coût n'est probablement plus la même dans le cadre de la couverture de transitions, où les techniques implicites imposent de doubler le nombre de variables de BDDs impliquées. De plus, doubler ce nombre de variables ne permet de conserver qu'une information encore grossière sur la structure des transitions entre états : cela permet d'indiquer uniquement si une transition entre deux états est possible. Connaître plus finement la structure des transitions entre états nécessiterait en plus de conserver dans les BDDs les variables d'entrées, ce qui ferait encore empirer les coûts de construction des graphes de transitions. Avec les techniques explicites, les différences de coût pour l'obtention d'informations de différente granularité concernant la structure des transitions entre états est beaucoup moins marquée.

De plus, en terme d'implémentation, la représentation en intention de l'espace d'états atteignables ne facilite pas l'exploration de celui-ci. Il est nécessaire de stocker de nombreuses informations comme les états ou les transitions couvertes dans des BDDs séparés, sans cesse mis à jour par des opérations au coût non constant. De même, la génération de valuations d'entrées permettant la transition d'un état à un autre se fait par des calculs d'image inverses coûteux. La mémorisation dans des structures de mémoire cache des résultats de ces calculs d'images inverses évite certes de répéter sans cesse les mêmes calculs mais n'évite pas le coût des premiers calculs.

Enfin et surtout, la génération de séquences de tests implique de manière intrinsèque une exploration énumérative de la structure de l'espace d'états atteignables : l'inconvénient principal des techniques explicites n'en est donc plus un dès lors que l'énumération devient incontournable.

Un outil de génération de séquences de tests par une approche purement explicite a été développé par une des équipes de consultants d'Esterel Technologies (cf. 4.4, p. 101). Leur approche est basée sur l'analyse des automates explicites produits par notre générateur

d'automates présenté en 4.1.6 (p. 91). Les expériences menées ont été concluantes bien que cette approche soit loin d'être optimale. En effet, il n'est pas nécessaire de générer un automate explicite représentant complètement le système pour générer des séquences de tests exhaustives. Nous proposons en 5.4 (p. 110) des techniques moins coûteuses et plus en adéquation avec les objectifs de couverture des séquences de tests, basées sur une approche hybride implicite/explicite de l'exploration d'espaces d'états atteignables.

3.10 Conclusion

Nous avons présenté dans ce chapitre un nouvel outil de vérification formelle basé sur des techniques implicites. Cet outil propose plusieurs techniques permettant de réduire le nombre de variables impliquées dans les calculs d'espaces d'états atteignables, notamment :

- le remplacement de certaines variables d'états par des entrées libres : cette technique usuelle permet de supprimer des variables des BDDs d'espaces d'états atteignables et de réduire le nombre de substitutions à effectuer mais ne réduit pas le nombre de quantifications existentielles ;
- l'abstraction de certaines variables à l'aide d'une logique trivaluée : cette technique étend la précédente en faisant complètement disparaître les variables abstraites de tous les BDDs, même intermédiaires, et réduit à la fois le nombre de substitutions et de quantifications existentielles à effectuer ;
- la prise en charge, à l'intérieur du vérifieur, de l'édition de lien des observateurs et du modèle observé : cette technique permet de s'affranchir de liaisons excessives et pénalisantes générées par le compilateur `Esterel` entre les différentes parties des circuits et d'augmenter le rayon d'action du *sweeping*.

Comme le remplacement de variables d'états par des entrées libres ou l'abstraction de variables à l'aide d'une logique trivaluée calculent des sur-approximations de l'espace d'état, nous proposons d'utiliser des informations structurelles concernant les modèles —fournies par le compilateur `Esterel`— afin de réduire cette sur-approximation.

Bien qu'à l'heure actuelle notre implémentation du calcul d'espace d'états à l'aide d'une logique trivaluée ne soit guère optimisée —essentiellement, nous ne partitionnons pas les calculs d'images—, l'abstraction de variables s'avère déjà être notablement plus efficace dans certains cas que la technique usuelle de remplacement de variables d'états par des entrées libres. Adapter dans nos calculs d'images trivalués les différentes techniques adoptées ces dix dernières années dans le domaine des calculs d'images de fonctions purement booléennes permettrait d'améliorer encore les performances de cette technique.

Aussi, notre outil dépend de l'utilisateur pour la sélection des variables à abstraire. En pratique, cela semble aisé à déterminer pour quelqu'un maîtrisant le modèle, *a fortiori* s'il dispose d'une vue arborescente de la structure du modèle. Néanmoins, l'automatisation de la sélection des variables à abstraire demeure à approfondir : les techniques que nous avons envisagées ne sont par encore applicables aux circuits générés par le compilateur

`Esterel` en version 5. Il conviendrait de les étudier de nouveaux sur la prochaine version du compilateur mais aussi de chercher de nouvelles heuristiques.

Enfin, rappelons que cette technique d'abstraction de variables est complètement orthogonale à de nombreuses autres. Par exemple, le partitionnement des modèles visant à ne calculer que l'espace d'états atteignables de partitions du modèle et de combiner ces espaces d'états atteignables par la suite : cette stratégie remplace les registres ne faisant pas partie de la partition en cours d'analyse par des entrées libres. Il pourrait être intéressant d'appliquer à cette technique les résultats de nos travaux sur l'abstraction de variables à l'aide d'une logique trivaluée.

Chapitre 4

Approche explicite

Les approches explicites de l'analyse de circuits sont basées sur une énumération des états atteignables du système, ces états étant stockés et manipulés *en extension*.

Par rapport aux techniques implicites à base de BDDs, les approches explicites souffrent de certains défauts qui en limitent le domaine d'application. La représentation explicite des états atteignables occupe souvent un espace mémoire important, mais qui a l'avantage d'être prévisible. De même, l'analyse individuelle des états implique souvent des débits d'analyses —le nombre d'états analysés par unité de temps— plus faibles.

Néanmoins, à l'inverse des approches implicites à base de BDDs, où les coûts des analyses sont fort peu prévisibles, la progression des calculs dans les approches explicites est souvent très régulière. De plus, les approches explicites permettent une analyse plus fine des transitions du système, ce qui permet la génération d'automates. Aussi, les techniques explicites ne sont sensibles au nombre de variables d'états que selon un facteur linéaire.

Nos travaux sur les techniques explicites ont débuté par la génération d'automates à partir des circuits générés par le compilateur `Esterel`. L'ancien générateur d'automates explicites, datant de la version 4 du compilateur, considérait les circuits comme des listes d'équations à résoudre linéairement (cf. 2.5, p. 31). Cela nécessitait au minimum un tri topologique préalable, voire une décyclisation du circuit, processus potentiellement très coûteux à la fois en temps et en mémoire.

Or, travailler sur un circuit acyclique n'est pas nécessaire : selon la sémantique constructive [Ber99] du langage `Esterel`, un circuit est *constructif* si, pour peu que l'on maintienne ses entrées électriquement stables durant un temps suffisant, toutes les portes qui le constituent se stabilisent à leur tour en un temps fini. Cette définition est indifférente aux cycles.

Nous avons donc développé un nouveau moteur d'évaluation explicite de circuits, complètement conforme avec la sémantique constructive du langage, présenté dans la section suivante. La section 4.2 présente l'utilisation de ce moteur à des fins de génération d'automates explicites. La section 4.3 présente son utilisation à des fins de vérification formelle. Enfin, la section 4.5 présente une conclusion sur les approches explicites.

4.1 Le nouveau moteur d'évaluation explicite de circuits

En respect avec la sémantique constructive du langage `Esterel`, le nouveau moteur d'évaluation explicite de circuits ne considère plus les circuits comme des listes triées de portes logiques (*nets*) mais comme des graphes orientés arbitraires, éventuellement cycliques. L'algorithme de base simule la propagation du courant électrique au travers des portes, jusqu'à la stabilisation électrique du circuit.

4.1.1 Portes logiques

Nous considérons les portes logiques comme des objets en attente de messages indiquant la valeur de leurs prédécesseurs, qui calculent leur propre valeur à partir de ces informations, puis informent à leur tour leurs successeurs de leur valeur. A partir de valuations spécifiées des registres et des entrées du circuit, la propagation s'effectue jusqu'à ce que la totalité du circuit soit évaluée. S'il demeure des portes non stabilisées alors qu'il n'y a plus d'information à propager, le circuit est rejeté comme non-constructif.

Cet algorithme est linéaire en fonction du nombre de littéraux dans les équations. Il ne peut boucler indéfiniment dans la mesure où la valeur d'une porte ne peut plus changer dès lors qu'elle est résolue.

Les règles d'évaluation des portes logiques sont habituelles :

- une porte `ou` transmet la valeur *vrai* dès qu'une de ses entrées est évaluée à *vrai*, la valeur *faux* lorsque toutes ses entrées sont valuées à *faux* ;
- une porte `et` transmet la valeur *faux* dès qu'une de ses entrées est évaluée à *faux*, la valeur *vrai* lorsque toutes ses entrées sont valuées à *vrai* ;
- une porte `non` transmet immédiatement la négation de la valeur de son prédécesseur.

Les portes plus évoluées apparues dans la version 7 du compilateur `Esterel` (*égalité*, *implication*, *équivalence*, *exclusion*, *xor* et *pla*) se dérivent aisément des portes précédentes[1].

Notons que l'utilisation d'un graphe de portes limitées à des portes `et` binaires et d'inverseurs (cf. 3.5.3, p. 66) permettrait d'unifier les différentes portes et de simplifier l'algorithme de stabilisation du circuit décrit plus loin. Cela n'a pas encore été fait mais est envisagé.

4.1.2 Algorithme de base

Pour que l'évaluation du circuit soit exhaustive, nous devons considérer toutes les valuations possibles des entrées du circuit.

De manière récursive, on construit l'arbre de décision en choisissant une première valeur (*faux*, par exemple) pour une entrée qui n'a pas encore été fixée, puis on propage

[1] D'autant plus que, en vertu des lois de De Morgan, un seul des couples de fonctions (*et*,*non*) et (*ou*,*non*) était déjà suffisant.

cette valeur ; une fois la récursion terminée, on recommence (*backtrack*) avec l'autre valeur possible (en l'occurrence, *vrai*).

Une fois le circuit stabilisé, les valeurs des registres constituent la signature d'un état accessible du circuit. Cette signature, le vecteur de valuation des registres, peut alors servir de clé de hachage pour la table stockant les états connus. Les états qui n'ont pas encore été traités sont ajoutés dans une file de traitement, sur laquelle on itère jusqu'à ce que tous les états accessibles aient été complètement évalués (Algorithme 4.1).

```
1   Propager les constantes
2   Tant qu'il y a des états non analysés dans la file
3     Retirer un état non analysé de la file et le propager
4     Tant que le circuit n'est pas stabilisé
5       Si il existe au moins une entrée non traitée
6         Choisir une entrée (ou un test) non traitée
7         Propager la valeur faux pour cette entrée et récursion en 4
8         Propager la valeur vrai pour cette entrée et récursion en 4
9       Sinon le circuit n'est pas constructif
10    Si l'état atteint est nouveau, l'ajouter dans la file
```

Algorithme 4.1: Algorithme de base du moteur d'évaluation explicite de circuits

4.1.3 Complexité de l'algorithme

Les métriques majeures dans l'analyse de la complexité de cet algorithme, ainsi que les métriques que nous utiliserons par la suite sont :

- *inputs*, le nombre de signaux d'entrées ou de tests du circuit. Notons que, par la suite, nous confondrons signaux d'entrées et tests.
- *états*, le nombre d'états atteignables du circuits.
- *littéraux*, le nombre de littéraux du circuit.
- *registres*, le nombre de registres du circuit.
- *outputs*, le nombre d'outputs du circuit. Notons que, selon l'objectif de l'analyse, nous pouvons être amenés à considérer comme outputs plus ou moins de portes qu'il n'y a réellement d'outputs du circuit. Par exemple, dans le cadre d'une génération d'automate (cf. 4.2, p. 92), les portes activant des actions de manipulation de données sont aussi considérés comme outputs du circuit. Inversement, dans le cadre de la vérification formelle d'un circuit (cf. 4.3, p. 100), seuls les outputs des observateurs sont pris en considération.

Dans le pire des cas, la phase de propagation des constantes n'a que très peu d'effets. On considèrera donc qu'elle n'a qu'un coût constant. La ligne 2 indique que le nombre d'états atteignables de la machine intervient de manière linéaire dans la complexité de l'algorithme. Les lignes 7 et 8 indiquent une double récursion, dont la profondeur maximale correspond au nombre de signaux d'entrées du circuit. Les diverses propagations (lignes 1, 3, 7 et 8)

ont, dans le pire des cas, un coût linéaire en fonction du nombre de littéraux (une fois que la valeur d'un littéral est calculée, elle ne peut plus évoluer).

Ainsi, on peut exprimer la complexité de l'algorithme 4.1 comme étant d'ordre

$$O\left(états * 2^{inputs} * littéraux\right)$$

dans le pire des cas.

On retrouve clairement l'explosion combinatoire selon le nombre d'entrées entrevue en 2.2 (p. 23), que nous traiterons en premier lieu. Nous verrons en 4.1.5 (p. 88) comment réduire les facteurs linéaires de complexité.

4.1.4 Réduction de l'explosion combinatoire

Utilisation des relations entre les entrées du circuit

Le langage `Esterel` donne à l'utilisateur la possibilité de spécifier des relations entre les signaux d'entrées du programme. Ces relations expriment des restrictions sur l'environnement dans lequel le programme sera executé, en l'occurence des exclusions ou des implications entre signaux. De telles relations sont très fréquemment utilisées dans les programmes traités par le compilateur `Esterel`, notamment pour indiquer au compilateur l'encodage implémenté par certains faisceaux d'entrées (encodage *one-hot* par exemple).

Ne pas prendre en compte ces relations conduirait à exécuter le programme dans des configurations pour lesquelles il n'a pas été prévu. D'autre part, les relations peuvent réduire le nombre de signaux d'entrées sur lesquels s'applique le facteur exponentiel, dès lors que le choix d'une valeur pour un signal intervenant dans une relation va potentiellement déterminer la valeur d'autres signaux. Par exemple, avec les relations $I3 \Rightarrow I2$ et $I2 \Rightarrow I1$, les signaux $I2$ et $I1$ sont déduits à *vrai* dans la branche où $I3$ est *vrai*, et les signaux $I3$ et $I2$ sont déduits à *faux* dans la branche où $I1$ est *faux* (contraposées).

Dans la version 5 du langage `Esterel`, les opérateurs admis dans les expressions de relations sont l'implication (opérateur binaire) et l'exclusion (opérateur n-aire). Dans notre moteur d'évaluation explicite de circuits, les relations sont représentées sous la forme d'un graphe orienté arbitraire, dont les nœuds sont des signaux d'entrées, et les arcs des implications de valeurs. Les arcs de ce graphe sont étiquetés à leur source par les valeurs des signaux d'entrées auxquelles ces implications s'appliquent et, à leur cible, par la valeur que doit nécessairement prendre le signal d'entrée cible. La Figure 4.1 présente un tel graphe pour les implications $I3 \Rightarrow I2$, $I2 \Rightarrow I1$ et l'exclusions entre les signaux $I1$, $I4$ et $I5$. Une telle représentation permet la détermination en temps constant des conséquences d'un choix sur la valeur d'un signal d'entrée, et dans n'importe quel ordre. La construction d'un tel graphe se fait en temps et en espace constants pour les implications : un arc pour l'implication elle-même et un autre arc pour la contraposée. Par contre, les exclusions nécessitent un temps de traitement et un espace quadratiques, puisque chaque signal est mis en correspondance avec tous les autres. Néanmoins, la phase de construction du graphe de relations demeure toujours très rapide en pratique.

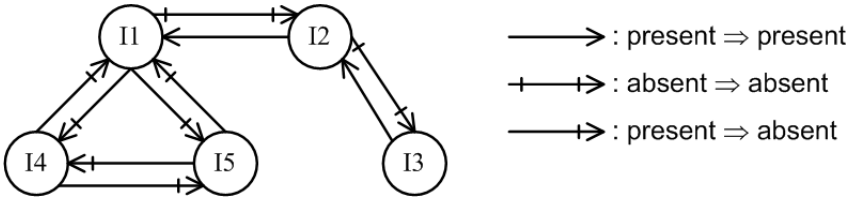


Figure 4.1: Graphe de relations entre entrées

Expansion des compteurs

En `Esterel`, les constructions de *répétitions* (**`await n expr`**, **`repeat n times`**, ...) génèrent par défaut des *compteurs*, décrémentés à la fin de chaque itération jusqu'à ce qu'ils atteignent 0.

La Figure 4.2 présente le circuit généré à partir du Programme 4.1. Avant que la boucle ne démarre, la variable stockant le compteur est initialisée à la valeur spécifiée (en l'occurrence : 3). Si celle-ci n'est pas nulle, l'intérieur de la boucle est exécuté, puis le compteur est décrémenté *via* l'action `dsz` (*Decrement and Skip on Zero*), qui indique alors si la variable a atteint 0.

```
1  module Counters:
2
3    output  O;
4
5    repeat 3 times
6      emit O ;
7      pause
8    end
9
10 end module
```

Programme `Esterel` 4.1: `Counters`

Lorsque le moteur d'analyse explicite de circuits rencontre un test de nullité, il considère à tout instant que ce test peut tout autant retourner *vrai* que *faux*. Cela a pour effet de désynchroniser fortement des parties de programmes que l'utilisateur voulait synchronisées. Le relâchement des contraintes sur les compteurs augmente donc le degré de liberté du modèle, ce qui peut conduire à une explosion combinatoire de l'analyse du modèle. De manière similaire à ce que nous avons vu en 3.3.1 (p. 52), ce relâchement de contraintes peut déclencher un effet "boule de neige" : des états non atteignables sont trouvés à tort atteignables, ces états conduisent à leur tour à des états qui peuvent ne pas être réellement atteignables, etc.

A partir de la version 5_98 du compilateur `Esterel`, il est possible d'expanser les compteurs : un compteur démarrant à n est alors remplacé par une combinaison de $\lceil \log_2(n) \rceil$

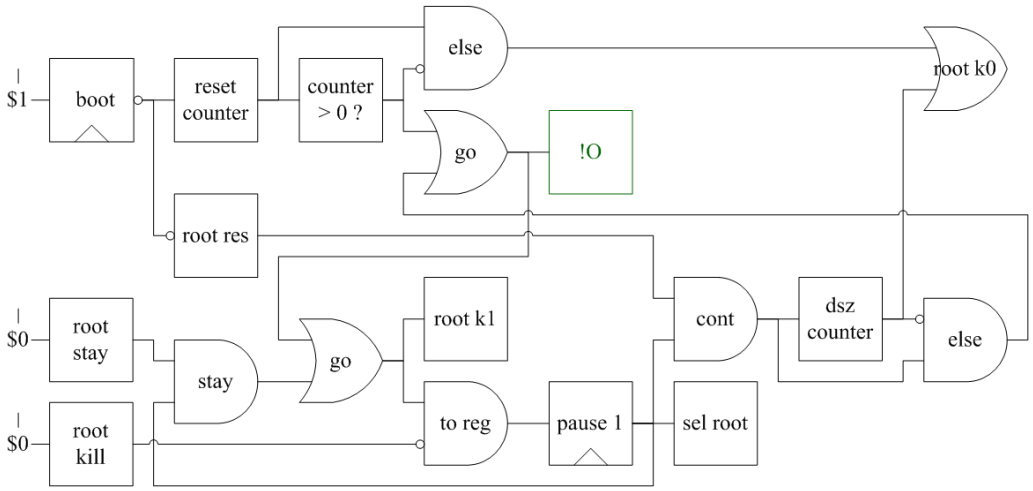


Figure 4.2: Le circuit du Programme 4.1 sans expansion des compteurs

registres[2]. Le remplacement des compteurs permet donc de respecter les synchronisations désirées par l'utilisateur, de rétablir le degré de liberté du modèle, mais au prix de l'augmentation du nombre de registres. S'il n'est pas clair que le remplacement des compteurs par des registres soit systématiquement intéressant pour le calcul d'espace d'états avec des BDDs, cela semble l'être pour l'approche explicite. Effectivement, le remplacement des compteurs par des registres ne fait qu'ajouter des bits aux vecteurs identifiant les états et augmenter le coût de certaines analyses en fonction du logarithme de la valeur initiale du compteur. Cela semble largement préférable à l'explosion combinatoire, voire indispensable comme le montrent les expérimentations sur le programme `Testbench` (cf. 6.7, p. 147).

Partitionnement des entrées du circuit

En fonction de l'état en cours d'analyse, certaines entrées du circuit n'ont pas d'influence sur son comportement et ne devraient donc pas intervenir dans cette analyse.

En terme de portes logiques, cela signifie que la condition d'activation du test de présence ou d'absence d'une entrée est déterminée par un registre (ou une combinaison de registres) qui s'évalue à *faux* (cf. Figure 3.3, p. 52). Par exemple, dans le cas trivial d'une séquence d'instructions non instantanées, les différents registres alimentant ces instructions seront tous exclusifs.

La détection des entrées non influentes se fait, après propagation des valeurs des registres déterminant l'état en cours d'analyse, par un parcours arrière en profondeur du graphe représentant le circuit, à partir de chacune des portes dont la valeur finale est importante (outputs, registres, actions, ...). Lorsqu'une porte déjà référencée comme atteignant une porte importante est de nouveau atteinte, la récursion en cours peut-être interrompue. Cette analyse n'a donc qu'un coût linéaire en fonction du nombre de portes du circuit. Les

[2]Cela était déjà possible depuis longtemps pour la sortie `BLIF`, soit plus en aval dans la chaîne de compilation.

entrées non visitées lors de ce parcours ne participent pas aux calculs de valeurs de ces portes, et elles ne doivent pas être sélectionnées à la ligne 6 de l'algorithme 4.1.

Cette analyse, encore améliorée dans la section suivante, apporte déjà une énorme réduction du temps de calcul.

Pondération des entrées du circuit

L'analyse des entrées influentes peut être menée plus en avant en calculant une pondération de cette influence. Le critère que nous avons retenu est le nombre de portes importantes (outputs, registres, actions, ...) pouvant être influencées par chaque entrée. L'idée est d'effectuer en premier lieu des branchements sur les entrées les plus décisives de l'état en cours d'analyse. On espère ainsi que ces branchements anticipés permettront d'éviter d'autres branchements ultérieurs sur des entrées ayant moins d'influence.

Similairement au partitionnement précédent, la pondération des entrées se fait par parcours arrière en profondeur du graphe représentant le circuit, toujours à partir des portes dont la valeur finale est importante. Cette fois-ci, le nombre de portes importantes atteignables doit être accumulé, donc chacun des parcours successifs peut-être amené à revisiter les mêmes portes. Cette analyse n'est donc plus linéaire mais quadratique dans le pire des cas. Dans la pratique, le pire des cas est toujours loin d'être atteint, et cette analyse se révèle être assez peu coûteuse.

L'exemple de l'arbitre de bus de Robert de Simone [BdS92, BMdST96] (une version simplifiée de celui de Ken McMillan [McM92]) met bien en valeur la validité de cette approche. Cet arbitre de bus implémente un protocole de type Token Ring, à priorité tournante. Il est constitué de cellules pouvant demander l'accès (exclusif) au bus à n'importe quel moment, l'équité et l'absence de famine étant garanties par une priorité tournante à chaque top d'horloge. Au premier instant, la première cellule peut obtenir l'accès dans l'instant, si elle le désire, en priorité sur toutes les autres cellules ; la deuxième cellule ne peut obtenir l'accès que si la première ne le demande pas, et la dernière cellule ne peut obtenir l'accès que si aucune autre ne le demande. Au second instant, la première cellule devient la moins prioritaire, la seconde cellule devient la plus prioritaire, etc.

Dans l'implémentation en `Esterel`, la cellule i désirant accéder au bus émet un signal $\texttt{Req}_i$, et la cellule obtenant l'accès reçoit un signal $\texttt{Ack}_i$. L'automate résultant a autant d'états qu'il y a de cellules, chaque état déterminant la cellule la plus prioritaire. Chacune des cellules pouvant demander l'accès au bus à n'importe quel moment, on peut s'attendre à ce que chaque transition soit de taille 2^n, n étant le nombre de cellules. Néanmoins, dès lors qu'une cellule de haute priorité demande l'accès au bus, les demandes d'accès provenant des cellules de plus basse priorité sont ignorées. Les requêtes des cellules de plus haute priorité doivent donc être testées en premier, puisqu'elles déterminent directement le nouvel état de la machine et qu'elles rendent inutiles les tests des cellules de plus basse priorité.

Appliquée à cet exemple, la pondération des entrées du circuit permet la détection de ce système de priorité, et les signaux de requêtes d'accès sont testés, pour chaque

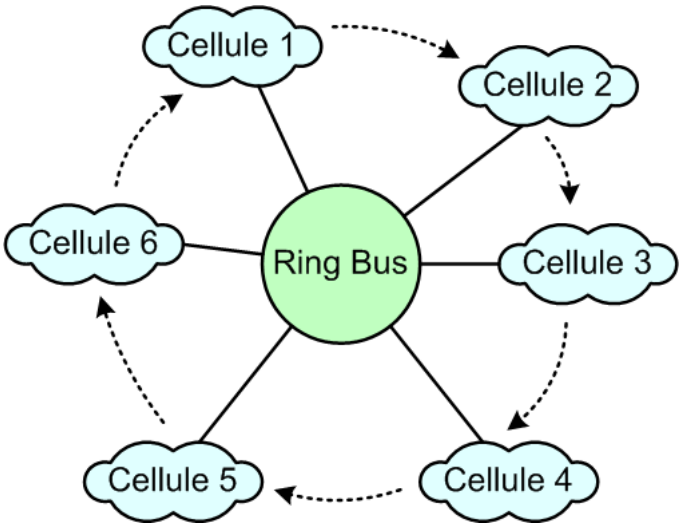


Figure 4.3: Arbitre de bus (à 6 cellules)

état, par ordre décroissant de priorité. L'automate résultant de l'analyse du circuit est ainsi directement minimal, chaque transition ayant une forme de peigne, de taille linéaire (Figure 4.4).

Dans la version actuelle du moteur d'évaluation explicite de circuits, la pondération des entrées est effectuée par défaut après la propagation de l'état en cours d'analyse. Cette pondération détermine ainsi l'ordre dans lequel les entrées seront testées pour la totalité de l'instant. Afin d'essayer de minimiser encore plus le nombre de branchements nécessaires à l'analyse d'un état, la pondération des entrées pourrait être mise à jour avant tout branchement ou, solution intermédiaire, régulièrement tous les n branchements. Cette approche gloutonne, qui propose donc d'effectuer plusieurs analyses de coûts quadratiques pour essayer de réduire le facteur exponentiel, n'a pas été testée.

Le partionnement des entrées apportait déjà un gain très notable par rapport à une sélection aléatoire des entrées à propager (il ne présente aucun intérêt de chercher à mesurer cet apport). Le Tableau 4.1 présente quelques mesures des gains apportés par la pondération des entrées par rapport au simple partionnement. On constate que le temps passé à calculer les poids des entrées pour chaque état est toujours rentabilisé, souvent très largement. La réduction de la consommation mémoire nécessaire aux données de sauvegarde des portes est souvent notable.

Constructivité faible

En respect de la sémantique constructive [Ber99] du langage `Esterel`, un circuit est *constructif* si, pour peu que l'on maintienne ses entrées électriquement stables durant un temps suffisant, toutes les portes qui le constituent se stabilisent à leur tour en un temps fini. L'algorithme de base du moteur d'évaluation explicite de circuit cherche donc à stabiliser toutes les portes, par branchements successifs sur les entrées. S'il demeure

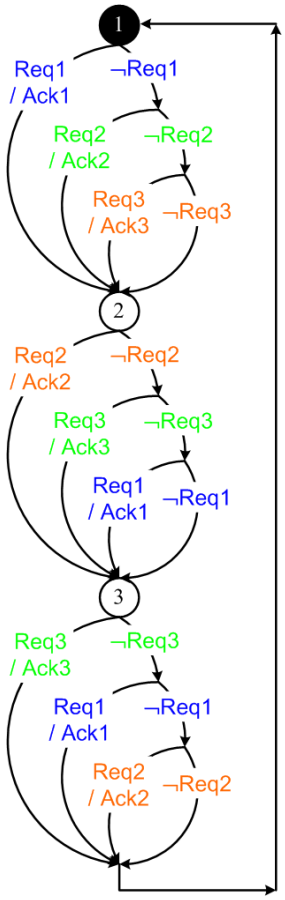


Figure 4.4: Automate de l'arbitre de bus (à 3 cellules)

modèle	∅	états	partitionnement			pondération					
			nœuds	temps	mém.	nœuds		temps		mém.	
ATDS-100-C2	15	81	353	0.04s	172Ko	267	-24%	0.04s	≈	172Ko	≈
Wristwatch	9	41	1 408	0.04s	140Ko	1 350	-4%	0.04s	≈	128Ko	-9%
Arbiter12	12	13	794	0.36s	172Ko	288	÷3	0.01s	÷36	172Ko	≈
Renault	11	161	11 819	0.90s	716Ko	9 451	÷4	0.48s	÷2	484Ko	-32%
TCINT	9	310	3 256	1.16s	248Ko	3 151	-3%	0.36s	÷3	248Ko	≈
Testbench	242	243	81	1.99s	340Ko	81	=	1.97s	≈	340Ko	≈
TI	181	652 948	1 251 945	3h 25	164Mo	1 007 969	-19%	3h 25	≈	144Mo	-12%

Tableau 4.1: Apports de la pondération des entrées

des portes non évaluées alors que toutes les entrées ont été valuées, le circuit est rejeté comme non constructif. Exiger que toutes les portes se stabilisent électriquement peut donc nécessiter des branchements sur toutes les entrées, avec un coût strictement exponentiel en $O\left(2^{inputs}\right)$.

Afin d'éviter ce coût strictement exponentiel, nous proposons un relâchement de la règle de constructivité des circuits : la *constructivité faible*. Le moteur ne cherche alors qu'à stabiliser les portes strictement nécessaires à la poursuite de son analyse. Si l'on cherche à faire de la vérification formelle (cf. 4.3, p. 100), les portes nécessaires sont uniquement les registres —qui déterminent l'état suivant— et les signaux des observateurs à vérifier. Si l'on cherche à produire un automate explicite (cf. 4.2, p. 92), il faut y ajouter tous les signaux de sorties ainsi que les portes d'actions. Ce sont ces mêmes portes nécessaires, "objectifs" de l'analyse, qui sont utilisées pour la pondération des entrées (cf. 4.1.4, p. 81).

La constructivité faible permet donc d'éviter d'avoir à stabiliser des sous-ensembles du circuit dont les sorties n'alimentent que des portes déjà stabilisées. Cela permet donc d'éviter certains branchements et de réduire le facteur sur lequel s'applique l'exponentielle. Le Tableau 4.2 présente quelques mesures des gains apportés par le relâchement de la règle de constructivité sur différents modèles. La constructivité faible peut apporter dans certains cas des augmentations de performances considérables.

modèle	∅	états	nœuds	constr. forte		constr. faible			
				temps	mém.	temps		mém.	
`Wristwatch`	9	41	1 350	0.04s	128Ko	0.04s	≈	128Ko	≈
`ATDS-100-C2`	15	81	267	0.15s	172Ko	0.04s	÷4	172Ko	≈
`Arbiter12`	12	13	288	0.27s	172Ko	0.01s	÷27	172Ko	≈
`Renault`	11	161	9 451	0.48s	484Ko	0.48s	≈	484Ko	≈
`Testbench`	242	243	81	2.27s	340Ko	1.97s	-13%	340Ko	≈
`TCINT`	9	310	3 151	894.14s	1 196Ko	0.36s	÷2500	248Ko	÷5
`TI`	181	652 948	1 007 969	3h 26	147Mo	3h 25	≈	144Mo	-2%

Tableau 4.2: Apports de la constructivité faible

Traitement spécifique des actions *reset*

Lorsque l'option `-simul` est spécifiée au compilateur `Esterel`, un certain nombre d'informations supplémentaires sont générées par les différents processeurs. Ces informations optionnelles servent par exemple à permettre la simulation, graphique ou non, d'un programme, à faciliter son débogage, etc.

Avec l'option `-simul`, les variables stockant la valeur d'un signal valué sont marquées comme non initialisées au premier instant d'un programme afin de détecter les accès en lecture à des variables non initialisées. Jusqu'à la version 5_90, ces initialisations étaient insérées très tardivement dans la chaîne de compilation, au niveau des générateurs de code. Du fait qu'un sous-module peut être réinitialisé à tout moment, ces actions d'initialisations

de variables ont par la suite été générées au niveau du circuit, afin de permettre la compilation compositionnelle des modules.

Le Programme `Esterel` 4.2 définit trois signaux d'entrée valués : `I1`, `I2` et `I3`. La Figure 4.5 représente le sous-circuit correspondant à l'initialisation des variables. Les conditions d'activation des réinitialisations correspondent donc à la conjonction du registre de *boot* et de l'absence du signal correspondant (sa présence initialiserait la variable). Le fil "`go after intf act`" correspond à la continuation de ces actions, et expose une dépendance de données représentée par les pointillés orange.

```
1 module ResetActions:
2
3   input   I1: integer;
4   input   I2: integer;
5   input   I3: integer;
6
7   ...
8
9 end module
```

Programme `Esterel` 4.2: `ResetActions`

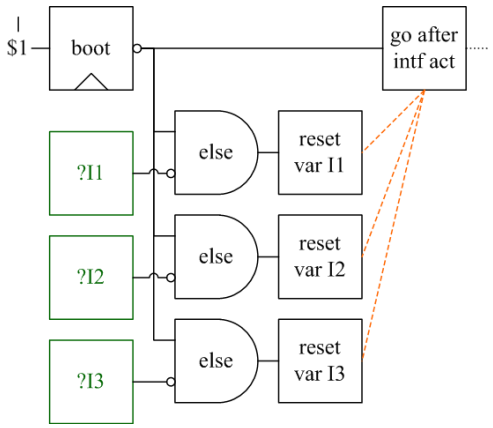


Figure 4.5: Circuit du programme `Esterel` 4.2

Du fait de cette dépendance de données, le moteur d'évaluation explicite de circuits va commencer, pour l'analyse du premier instant, par effectuer des branchements successifs sur les signaux d'entrées `I1`, `I2` et `I3`, avec un coût exponentiel. Lorsque l'on produit un automate, on obtient (avec la détection *a posteriori* des branches reconvergentes présentée en 4.2.3), le début de graphe de transition de la Figure 4.6, qui est purement linéaire !

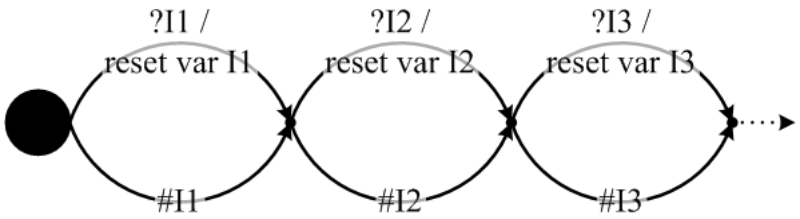


Figure 4.6: Début du graphe de transition d'initialisation de variables

Afin d'éviter tout risque d'explosion combinatoire lors de l'analyse des sous-circuits responsables des initialisation de variables, nous leur appliquons un traitement spécifique. Ces sous-circuits sont automatiquement débranchés et, lorsque l'on produit un automate, les débuts des graphes de transitions des premiers instants sont déduits des sous-circuits par un traitement séparé. Ce processus est purement linéaire.

Analyse *a priori* des branches reconvergentes

La complexité exponentielle de l'algorithme d'évaluation explicite de circuits provient du fait que toutes les entrées influentes sont traitées comme si elles avaient un impact global sur circuit, que ce soit réellement le cas ou que leur effet soit au contraire très localisé. Dès lors, l'analyse du circuit se doit de diverger systématiquement pour chaque entrée influente, d'où la complexité en 2^{inputs}.

Par exemple, dans le Programme `Esterel` 4.3, les réactions à la présence ou l'absence des signaux `I1`, `I2` et `I3` sont indépendantes. Néanmoins, le moteur va analyser les deux branches correspondant au signal `I1` puis, dans chacune de ces branches, celles correspondant au signal `I2`, etc.

```
1  module ReconvergentBranches:
2
3    input   I1, I2, I3;
4    output  O1, O2, O3;
5
6    loop
7         [
8           present I1 then emit O1 end ;
9           present I2 then emit O2 end
10        ]
11     ||
12        present I3 then emit O3 end
13    each tick
14
15 end module
```

Programme `Esterel` 4.3: `ReconvergentBranches`

Notons que ce problème d'explosion combinatoire n'est pas le même que celui évoqué en 2.2 (p. 23). Dans les programmes dérivés de `AB_O` (Programme 2.3, p. 23), il est réellement nécessaire de tester de manière arborescente tous les signaux n'ayant pas été reçus, puisque chaque combinaison détermine un nouvel état. Dans le Programme 4.3, aucun comportement ne dépend de plus d'un signal d'entrée à la fois.

La Figure 4.7 représente le circuit correspondant au Programme 4.3. Les trois blocs encadrés en pointillés correspondent aux lignes **`present Ix then emit Ox end present`**. Il serait bon que le moteur d'analyse explicite détecte qu'il n'y a aucune dépendence de données entre ces trois blocs. Certes, les comportements divergent localement selon que ce soit les portes *then* ou *else* qui s'évaluent à *vrai*, mais il y a reconvergence au niveau des portes de continuations (*go*).

Tout le problème réside donc dans la reconnaissance de sous-graphes présentant de tels cas de divergence puis de reconvergence du flot de contrôle. Notons que, paradoxalement, cette information structurelle est très aisée à obtenir dans les parties amont du

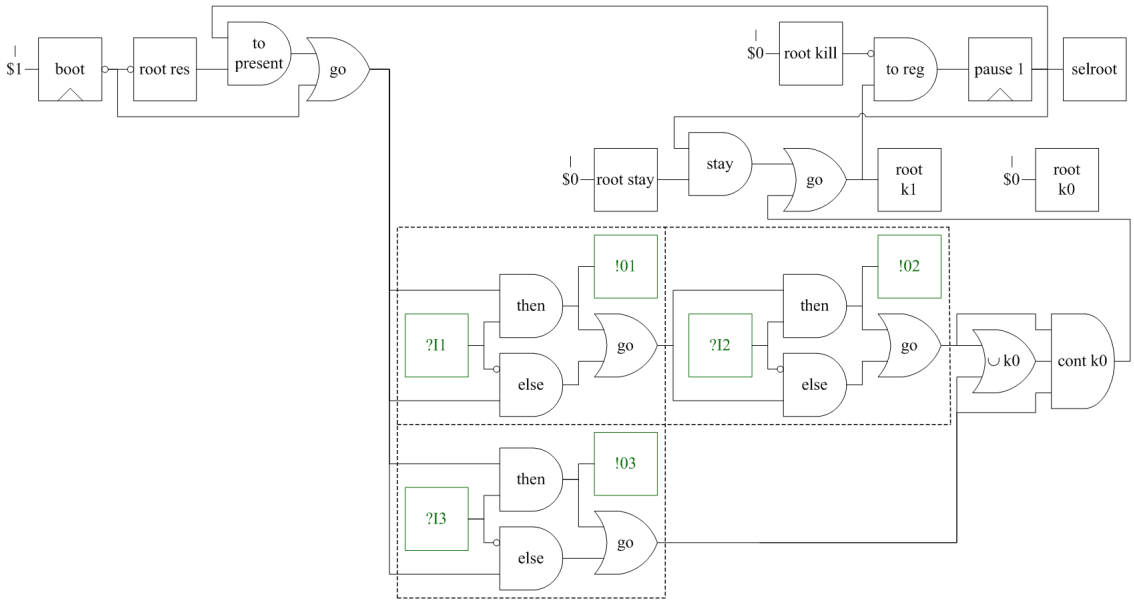


Figure 4.7: Circuit du programme `Esterel` 4.3 (`ReconvergentBranches`)

compilateur. Malheureusement, il n'est guère envisageable de compter sur la présence de telles informations au niveau des circuits : en cas d'optimisation du circuit préalable à son analyse explicite, de telles informations seraient probablement perdues.

Cette analyse correspond en fait à une détection d'*équipotentielles*, autrement dit des portes égales. En l'occurrence, il faudrait détecter que les quatre portes *go* sont égales. Il existe de nombreuses approches à ces analyses, qui sont à la base de la vérification formelle d'équivalence de machines (cf. 2.10.3, p. 45). Dans notre cadre, nous pourrions envisager :

- La compression de circuit basée sur l'utilisation d'un AIGraph (cf. 3.5.3, p. 66), avec notamment des passes de *BDD sweeping*, dans lesquelles la représentation canonique que permettent les BDDs d'une fonction est utilisée pour identifier des portes identiques. Cette méthode présente le désavantage de chercher à compresser le circuit sans prendre en considération les états atteignables : seul l'*input care set* est utilisable pour contraindre les BDDs manipulés et donc réduire leur taille. Le *BDD sweeping* ne serait donc probablement pas applicable à la totalité des portes d'un gros circuit.
- L'utilisation de techniques de tableaux, à la base des *SAT Solvers*. Cette méthode présente le désavantage d'avoir à identifier des paires de portes susceptibles d'être égales avant de pouvoir vérifier de telles hypothèses.

L'approche que nous avons choisie pour limiter *en pratique* les explosions combinatoires sur les branches reconvergentes est basée sur la propagation de BDDs plutôt que sur la propagation de constantes booléennes, et donc l'abandon de notre algorithme à branchements récursifs au profit d'un algorithme de stabilisation symbolique. Cette approche fait l'objet du Chapitre 5.

4.1.5 Optimisation des parties linéaires

Cette section expose différentes optimisations de notre moteur d'évaluation explicite de circuits. Ces optimisations relèvent plus du domaine des choix et stratégies d'implémentation. Elles permettent des améliorations non négligeables des temps de calculs ou de la consommation mémoire mais, à l'inverse des optimisations précédentes, n'influent pas réellement sur le domaine d'application des techniques explicites. Néanmoins, le cumul des apports de ces optimisations n'est pas pour autant à négliger.

Mise en queue des tests à effectuer (branchement retardé)

Les sous-circuits générés par les instructions de test (**`if`**, **`present`**) sont similaires à celui de la Figure 4.8 : la porte *go* active l'action de test *test*, dont le résultat correspond à l'entrée *test return*.

Lorsque le moteur d'évaluation explicite de circuit trouve une porte de test dont le fil *go* est activé, le branchement selon le résultat de ce test n'intervient pas tout de suite. Au lieu de cela, la porte de test est mise dans une queue, le branchement effectif n'intervenant que lorsqu'il n'y a plus d'information à propager : cela évite d'effectuer les mêmes calculs à plusieurs reprises dans chaque branche.

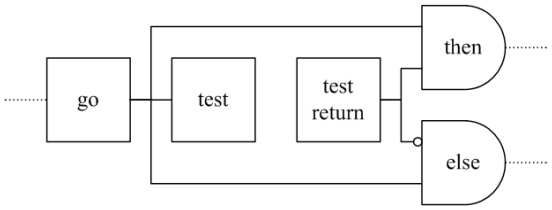


Figure 4.8: Sous-circuit correspondant à une instruction de test

Préservation de l'état du circuit entre les branchements

L'algorithme de base du moteur d'évaluation explicite de circuit alterne des phases d'évaluation des portes composant le circuit avec des branchements récursifs sur les points de décisions. Il est donc nécessaire de pouvoir préserver l'état du circuit avant branchement, afin de le restaurer à l'identique une fois la récursion terminée.

Une première approche consiste à sauvegarder la totalité du circuit avant chaque branchement. Pour qu'une implémentation efficace de cette technique soit possible, il faudrait que toutes les portes du circuit soit adjacentes en mémoire. La sauvegarde et la restauration se feraient alors avec la primitive `C memcpy`. Avec des objets C++, il faudrait redéfinir l'opérateur d'allocation mémoire (**`operator new`**) des classes de portes, ce qui ne serait guère élégant vu la finesse de la méthode.

Une approche dérivée consiste, avant tout branchement, à sauvegarder les données de toutes les portes du circuit dans des structures, appelées `UndoBlock`s, allouées de manière

adjacente. Si toutes les portes du circuit sont accessibles par un index, sauvegarde et restauration du circuits deviennent de simples itérations.

Le coût des sauvegardes et restaurations de la totalité du circuit entre chaque branchement demeure disproportionné. En considérant que la profondeur de récursion est bornée par le nombre d'entrées du circuit, la complexité en temps des processus de sauvegardes et de restaurations correspond au produit du nombre d'entrées par le nombre total de portes. La complexité en espace nécessite encore de multiplier par la taille de chaque structure de sauvegarde. De plus, les parties de circuit évoluant entre deux branchements successifs étant généralement assez circonscrites, la sauvegarde systématique de la totalité du circuit semble difficile à justifier.

Afin de minimiser le coût des sauvegardes et restaurations, nous ne sauvegardons les portes que lorsque cela est nécessaire. Avant tout changement d'état d'une porte, ses données sont sauvegardées dans un `UndoBlock`, qui mémorise aussi une référence vers cette porte. L'`UndoBlock` ainsi initialisé est stocké au sommet d'une pile. Avant tout branchement, le pointeur courant de cette pile est préservé. La restauration du circuit se fait par restaurations successives des portes dans l'ordre inverse de leur sauvegarde, en dépilant les `UndoBlock`s jusqu'à ce que toutes les portes modifiées dans la branche courante soient restaurées.

Cette approche n'est pas encore minimale puisqu'une porte peut être sauvegardée à plusieurs reprises entre chaque branchement. Néanmoins, éviter cela nécessiterait de mémoriser le dernier point de branchement (autrement dit la profondeur de récursion) auquel la porte a été sauvegardée pour la dernière fois. Mais cette information devrait aussi faire partie des données sauvegardées de la porte ! Ce compromis ne semble typiquement pas être intéressant.

Le nombre de fois qu'une porte peut-être sauvegardée est donc désormais borné par le nombre de littéraux qu'elle référence. La complexité en temps des processus de sauvegarde et de restauration est bornée par le nombre de littéraux du circuit. La complexité en espace est bornée par le produit du nombre de littéraux du circuit et par la taille de chaque structure de sauvegarde.

Ce mécanisme de sauvegarde/restauration a été étendu aux autres données devant être restaurées après chaque branchement, comme le nombre de portes demeurant non évaluées, la liste des tests en attente, etc.

Au fur et à mesure de l'évaluation du circuit, un grand nombre d'`UndoBlock`s vont être créés, puis détruits, puis recréés, etc. Afin de ne pas subir un surcoût à la fois en espace et en temps dû au gestionnaire de tas par défaut, les `UndoBlock`s sont alloués par gros blocs d'une centaine de kilo-octets (des *chunks*, voir aussi 4.2.5, p. 97), chaînés entre eux de manière à former une pile (Figure 4.9). Cette technique garantie une complexité en espace et en temps à peu près minimale pour les processus de sauvegardes et restaurations. Typiquement, cette optimisation permet de diviser par un facteur 2 le temps d'analyse des circuits. Bien que cela soit difficilement mesurable, nous nous plaisons à croire que l'augmentation de la

localité en mémoire des données de restauration, en augmentant l'efficacité du cache du processeur, contribue à cette amélioration.

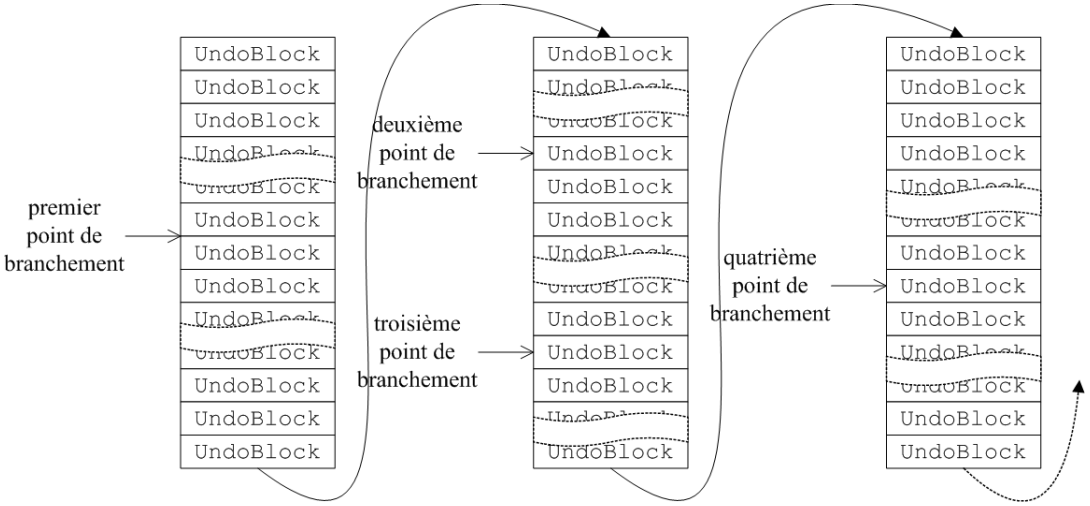


Figure 4.9: Chaînage des piles d'`UndoBlock`s

Notons enfin que, si l'implémentation de l'Algorithme 4.1 avait été strictement récursive, les portes auraient pu être sauvegardées directement dans la pile d'exécution du programme, ce qui aurait été plus élégant (quoique plus coûteux en mémoire, de pile en l'occurrence). Mais, l'implémentation réelle alternant récursivité et itération (sur les successeurs des portes, sur les tests en attentes, ...), cela n'est pas possible.

Mécanisme d'évaluation rapide des portes logiques

Les circuits produits par le compilateur `Esterel` v5 sont formés de portes n-aires (`et`, `ou`) ou unaires (`non`, fils). Le principe de base étant de simuler la propagation du courant électrique, en respect avec la sémantique constructive du langage [Ber99], on s'interdira toute rétention inutile d'information. Par exemple, dès qu'un des littéraux d'une porte `et` est évalué à *faux*, on peut immédiatement propager récursivement la valeur *faux* pour cette porte.

Notons qu'en pratique la propagation des valeurs n'est pas systématiquement effectuée dès qu'une porte devient évaluée : les circuits `Esterel` contiennent aussi des informations de dépendances de données, qui spécifient que la valeur de certaines portes ne doit pas être propagée tant que celles dont elles dépendent *causalement* (et non *fonctionnellement*) ne l'ont pas été. Ce mécanisme permet de garantir le respect de la *causalité* des actions spécifiées par l'utilisateur[3]. Ainsi, la propagation d'une valeur intervient donc en fait dès qu'elle est déterminée *et* propageable. De plus, nous voulons absolument, notamment pour

[3]Par exemple, si l'utilisateur écrit `x:= y; x:= x+1`, il est nécessaire de respecter l'ordre d'exécution de ces actions, alors qu'il est envisageable que l'évaluation de leur condition d'activation intervienne en ordre inverse, *a fortiori* avec un circuit préalablement optimisé.

les opérateurs n-aires, éviter de reparcourir la liste des littéraux de la porte à chaque fois qu'une nouvelle information est reçue.

Ces deux objectifs sont atteints en utilisant un compteur de prédécesseurs non résolus, initialisé au nombre de littéraux que référence la porte. La réception d'une valeur non contrôlante (*vrai* pour une porte `et`, *faux* pour une porte `ou`) induit la décrémentation de ce compteur. Lorsqu'il atteint 0, la valeur par défaut de la porte (*vrai* pour une porte `et`, *faux* pour une porte `ou`) est propageable. Par contre, la réception d'une valeur contrôlante (*faux* pour une porte `et`, *vrai* pour une porte `ou`) court-circuite ce processus et permet l'évaluation immédiate de la porte. Une porte `non` s'évalue immédiatement en la négation de la valeur de son prédécesseur.

Enfin, le respect de règle de dépendances de données s'effectue en maintenant un compteur de prédécesseurs bloquants, décrémenté à chaque fois qu'un prédécesseur de ce type se trouve évalué. Lorsque ce compteur atteint 0, alors la valeur de la porte est propageable dès lors qu'elle est déterminée.

4.1.6 Analyse des registres redondants

Il est courant qu'un modèle —*a fortiori* s'il a été développé à partir d'un langage de haut niveau— comporte un certain nombre de registres que l'on considère équivalents s'ils sont toujours égaux ou opposés. Les informations concernant les classes d'équivalences de registres sont utiles à plusieurs fins :

- elles sont à la base du réencodage de ces registres, justement pour en diminuer le nombre ;
- elles permettent d'indiquer ou de confirmer à l'utilisateur des points de code "synchronisés" ou en exclusion mutuelle ;
- enfin, un trop grand nombre de registres redondants peut révéler à l'utilisateur des choix d'architecturation de son application potentiellement coûteux.

Lorsque l'espace d'états atteignables du système est calculé avec des BDDs, les classes d'équivalences de registres sont directement disponibles puisqu'elles servent à réduire la complexité des calculs (cf. 2.9.4, p. 42). Inversement, lorsque l'espace d'états atteignables est énuméré par des techniques explicites, les classes d'équivalences de registres ne sont pas disponibles et doivent être calculées.

Nous avons implémenté une telle analyse dans notre moteur d'analyse explicite de circuits. Elle est basée sur un parcours linéaire de la table des états connus. Au départ, tous les registres sont considérés à la fois équivalents, opposés et s'impliquant deux à deux. Les états sont ensuite analysés et les relations sont mises à jour en conséquence. Ainsi, nous minimisons les accès aux bits identifiant un état dès lors que les registres correspondants sont connus comme n'étant pas en relation.

Durant nos expérimentations, cette analyse s'est avérée être fort peu coûteuse *a fortiori* au regard du coût de construction de l'espace d'états atteignables : dans le pire des cas, cet algorithme est quadratique.

4.2 Génération d'automates explicites

Le moteur d'évaluation explicite de circuit que nous venons de présenter peut être dérivé afin de générer des automates d'états finis. Parce qu'un automate rend complètement explicites les états atteignables d'un programme et les transitions entre ces états, il facilite énormément l'exécution (simulation), la vérification formelle, la génération de tests, etc., du programme ainsi explicité.

En ce qui nous concerne, un automate est un graphe dont les nœuds sont des états, et les arcs des arbres de décisions. Ces arbres de décisions comportent des nœuds binaires (branchements sur les signaux d'entrées du programme) ou unaires (signaux de sorties émis ou actions définies par l'utilisateur), comme l'automate de l'arbitre de bus présenté en 4.1.4 (Figure 4.4, p. 83).

Les arbres de transitions s'obtiennent en accumulant les actions à effectuer, déterminées par les portes correspondantes, au fur et à mesure de l'évaluation du circuit. Les nœuds correspondants à des points de décision sont construits en retour de récursivité, après que les deux sous-arbres, correspondants aux deux valeurs possibles du signal d'entrée, ont été construits.

Dans les versions 5_x du compilateur `Esterel`, les automates produits sont transcrits au format public `oc5` [oc598], commun aux compilateurs `Lustre` et `Esterel`. Dans les versions suivantes du compilateur `Esterel`, les automates produits sont transcrits dans un format dérivé, interne au compilateur.

Notre générateur d'automate fait partie de la chaîne de compilation `Esterel` depuis la version v5_91 (cf. Figure 2.6, p. 33) et est intégré à l'environnement de développement intégré `Esterel Studio`.

4.2.1 Complexité en espace des automates

La complexité en temps du générateur d'automate dépend essentiellement de celle du moteur d'évaluation de circuit sous-jacent, dont nous venons de traiter. Les problématiques issues précisemment de la génération d'automate proviennent donc pour la plupart de la taille des automates produits.

La complexité en espace d'un automate est tout autant exponentielle que la complexité en temps du processus de génération : pour chaque états, l'arbre de décision peut avoir 2^{inputs} nœuds. La complexité en espace est donc d'ordre

$$O\left(états * 2^{inputs} * outputs\right)$$

dans le pire des cas.

Nous avons vu en 4.1.4 (p. 81) que la pondération des entrées permettait de réduire, en pratique, de manière très notable le nombre de tests à effectuer. Outre le gain en temps, cela se répercute sur la taille des automates produits. De même, le branchement retardé vu en 4.1.5 (p. 88) permet de factoriser les nœuds d'outputs/actions en amont des tests, plutôt que de les dupliquer dans les sous-branches.

4.2.2 Partage en mémoire des sous-arbres isomorphes

Afin de réduire l'espace utilisé pour stocker l'automate, les arbres de décisions des transitions entre états sont en fait manipulés comme des graphes orientés acycliques. Les nœuds identiques des arbres de décisions sont partagés et il n'en existe dans le système que des instances uniques. Les sous-arbres isomorphes sont donc identifiés en tant que tels et partagés.

Les nœuds créés au fur et à mesure de l'exploration du programme sont stockés dans des tables d'adressage dispersé. Ces tables permettent la recherche de nœuds identiques en temps constant attendu. Afin de réduire le nombre de collisions (soit le nombre de nœuds ayant le même code de hachage), à chaque type de nœud correspond sa propre table d'adressage dispersé. De plus, la classe des nœuds correspondant à un nouvel état et la classe des états connus est la même. Nous évitons ainsi de consommer de la mémoire supplémentaire pour les nœuds correspondant à un nouvel état. Par la même occasion, cela permet de factoriser la table d'adressage dispersé des nœuds correspondant à un nouvel état et la table des états connus.

Le partage des sous-arbres de transitions isomorphes permet souvent une réduction notable de la taille des automates produits. Cette réduction de taille se fait néanmoins au détriment de la vitesse d'exécution de l'automate produit. En effet, lorsque le moteur d'interprétation de l'automate produit atteint un nœud partagé, la "linéarité" du processus d'interprétation est brisée et un branchement (*jump*) vers le nœud partagé doit être effectué. Il est à prévoir que la localisation en mémoire des nœuds partagés ne soit pas proche des nœuds les référençant, ce qui peut provoquer des défauts de cache ralentissant encore l'interprétation de l'automate.

Le Tableau 4.3 présente quelques mesures des gains apportés par le partage des nœuds isomorphes. La réduction du nombre de nœuds est parfois très importante. Cette réduction du nombre de nœuds semble souvent aller de pair avec une réduction de la consommation mémoire nécessaire à la constuction de l'automate. Dans la plupart des cas, le surcoût des tables d'adressage dispersé maintenant les nœuds uniques est (parfois largement) compensé par la réduction du nombre de nœuds total. Enfin, nous ne constatons pas de variation notable du temps nécessaire à la construction des automates.

modèle	∅	états	sans partage			avec partage					
			nœuds	temps	mém.	nœuds uniques		temps		mém.	
`Arbiter12`	12	13	312	0.01s	172Ko	288	-8%	0.01s	≈	172Ko	≈
`Wristwatch`	9	41	1 557	0.04s	128Ko	1 350	-13%	0.04s	≈	128Ko	≈
`ATDS-100-C2`	15	81	420	0.04s	172Ko	267	-36%	0.04s	≈	172Ko	≈
`TCINT`	9	310	7 314	0.37s	612Ko	3 151	÷2	0.36s	-3%	248Ko	÷2.5
`Renault`	11	161	21 878	0.47s	700Ko	9 451	÷2	0.48s	+2%	484Ko	-31%
`Testbench`	242	243	81	2.03s	336Ko	81	0%	1.97s	-3%	340Ko	+1%
`TI`	181	652 948	1 273 750	3h 25	147Mo	1 007 969	-21%	3h 25	≈	144Mo	-2%

Tableau 4.3: Apports du partage des nœuds isomorphes

Au sein du générateur d'automate, le partage des nœuds est immédiat et total. Néanmoins, les nœuds partagés ne sont générés en tant que tels dans l'automate produit qu'à partir d'un certain seuil de rentabilité. Ce seuil de rentabilité, défini par l'utilisateur, permet de gérer le compromis entre la taille de l'automate et sa vitesse d'interprétation.

A l'heure actuelle, notre générateur d'automate construit totalement l'automate en mémoire. Ce choix est fait dans le but de maximiser le partage des nœuds, bien que cela soit au détriment de la taille des automates générables, en pratique limitée par la quantité de mémoire physique disponible. En effet, nous estimons que le *working set* du générateur d'automates —l'ensemble des pages mémoire auquelles le processus accède régulièrement— est proche de l'ensemble des pages mémoire allouées. Cela est dû au fait que la répartition en mémoire des nœuds ou des états connus est pratiquement aléatoire (au *pooling* de blocs mémoire près, cf. 4.2.5, p. 97). Une fois la mémoire physique du système totalement consommée, il est à craindre que le processus passe alors la majorité de son temps à échanger des pages entre la mémoire vive et le fichier d'échange (*swap*). Cela ne serait pas le cas si le processus n'accédait régulièrement qu'à un nombre limité de pages mémoire.

Il serait tout à fait possible, une fois un seuil de mémoire libre atteint, de décharger des nœuds vers le fichier de sortie, avec une politique d'élection des nœuds à définir. Les nœuds ainsi déchargés ne pourraient alors plus être partagés par les états demeurant à construire. N'ayant pas à ce jour constaté un tel besoin, nous n'avons pas implémenté cette fonctionnalité.

4.2.3 Suppression *a posteriori* des branchements inutiles et branches reconvergentes

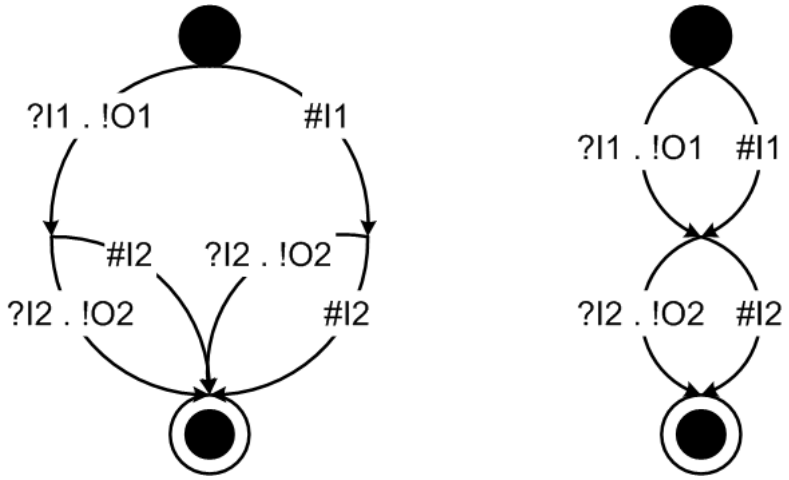
Si l'ordre d'évaluation des entrées ne change pas trop lors de la construction des graphes de transitions de chacune des branches d'un test, alors la technique de partage en mémoire des nœuds identiques peut permettre la détection *a posteriori* des branchements inutiles, par simple comparaison de pointeurs.

Outre le gain en espace qu'elle permet, cette technique (*hash-caching*), combinée avec la pondération des entrées (cf. 4.1.4, p. 81), permet de mieux faire apparaître la décomposition fonctionnelle du programme. Effectivement, les automates sont connus pour intriquer des parties de programmes non sémantiquement liées. Cela provient notamment du fait que les automates séquentialisent des processus initialement exécutés en parallèle. La détection de branches reconvergentes permet de mettre en exergue la séparation de processus non liés, comme le montre l'exemple du Programme 4.4, qui exécute en parallèle deux blocs non sémantiquement liés. La Figure 4.10(a) représente l'automate obtenu sans partage des nœuds des arbres de transitions. On constate que les deux branchements après celui sur le signal `I1` sont redondants, mais cela n'apparaît pas clairement. La Figure 4.10(b) présente le même automate, mais avec partage des nœuds des arbres de transitions. La séparation fonctionnelle des deux blocs initialement mis en parallèle y est alors directement visible.

Le Tableau 4.4 présente quelques mesures de la réduction du nombre de nœuds apportés par la suppression des branchements inutiles. La pondération des entrées (cf. 4.1.4, p. 81)

```
1  module UnrelatedParallelBlocks:
2
3    input I1, I2;
4    output O1, O2;
5
6       present I1 then
7         emit O1
8       end
9    ||
10      present I2 then
11        emit O2
12      end
13
14 end module
```

Programme Esterel 4.4: UnrelatedParallelBlocks



(a) Sans partage (b) Avec partage

Figure 4.10: Automates avec et sans partage du programme Esterel 4.4

vise déjà à limiter *a priori* les branchements. On ne constate donc généralement que peu de branchements inutiles *a posteriori*.

modèle	∅	états	sans suppression		avec suppression			
			nœuds	nœuds uniques	nœuds		nœuds uniques	
`Arbiter12`	12	13	312	288	312	=	288	=
`Wristwatch`	9	41	1 569	1362	1 557	-0.8%	1 350	-0.9%
`ATDS-100-C2`	15	81	444	271	420	-5%	267	-1.5%
`TCINT`	9	310	13 042	3 277	7 314	-44%	3 151	-4%
`Renault`	11	161	21 979	9 563	21 878	-0.5%	9 451	-1.2%
`Testbench`	242	243	81	81	81	=	81	=
`TI`	181	652 948	1 284 285	1 018 504	1 273 750	-0.8%	1 007 969	-1%

Tableau 4.4: Apports de la suppression des branchements inutiles

L'ordre de propagation des entrées étant le même lors de l'analyse de chaque état, les arbres de décisions équivalents devraient être isomorphes, donc détectés et supprimés. Pour aller plus loin et détecter les tests menant à des états différents mais équivalents, il est nécessaire de mettre en œuvre des techniques de bisimulation beaucoup plus onéreuses, comme proposé dans [CFG94, CFG95].

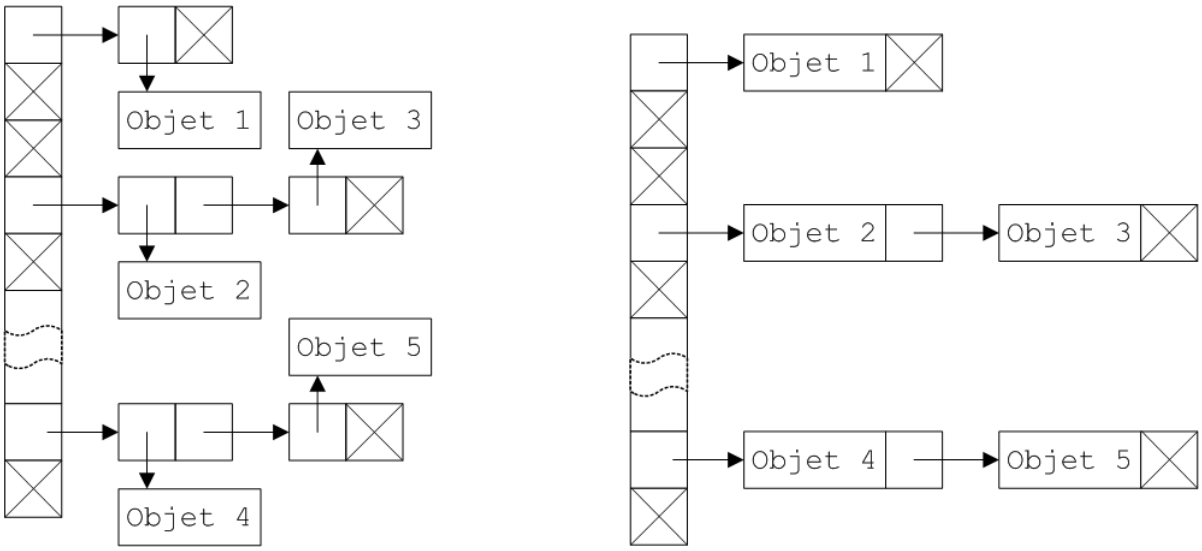
4.2.4 Implémentation des tables d'adressage dispersé

Outre les tables d'adressage dispersé permettant le partage à la volée des nœuds d'arbres de transition, notre moteur d'analyse explicite de circuits utilise de nombreuses autres tables d'adressage dispersé. La plus importante est celle des états connus, qui permet de retrouver rapidement un état à partir de son vecteur de registres. En mode vérification formelle, une autre table d'adressage dispersé permet de partager les vecteurs d'entrées identiques (cf. 4.3.1, p. 100). Les choix d'implémentation des tables d'adressage dispersé a donc un effet non négligeable sur les performances du moteur ainsi que sa consommation mémoire.

(Pour des raisons de simplicité, la discussion qui suit ne traite que des tables d'adressage dispersé simples, dans lesquelles les codes de hachage sont directement obtenus à partir des objets. L'extension aux tables d'adressage dispersé contenant des associations (clef,objet) est directe.)

Dans leur implémentation la plus usuelle, les tables d'adressage dispersé sont formées d'un tableau de listes chaînées, ou listes de collisions. Les objets sont stockés dans la liste chaînée d'index égal au quotient du code de hachage de l'objet par le nombre de listes. Pour minimiser la consommation mémoire, les listes chaînées peuvent être réduites à des pointeurs vers la première cellule de liste. Les cellules de listes sont constituées d'un pointeur vers l'objet et d'un pointeur vers la cellule suivante. La Figure 4.12(a) présente la structure de telles tables. Pour chacun des objets stockés dans cette table, il est donc nécessaire d'allouer une cellule de liste dont la taille est de deux mots (deux pointeurs).

Cette structure de table d'adressage dispersé est très versatile. Elle permet notamment à un objet d'être référencé par plusieurs tables différentes. Néanmoins, si la possibilité de référencer un même objet par plusieurs tables n'est pas utile, ce qui est le cas de la plupart des objets que nous manipulons, il est alors possible de réduire la consommation mémoire des tables en stockant le pointeur de chaînage de collisions directement dans l'objet. La Figure 4.12(b) présente une telle structure pour la table d'adressage dispersé vue en Figure 4.12(a).



(a) Structure usuelle

(b) Avec chaînage des collisions dans l'objet

Figure 4.11: Structures de tables d'adressage dispersé

Ces tables d'adressage dispersé avec chaînage dans l'objet permettent donc d'économiser un pointeur par objet. Cela est surtout notable lorsque les objets sont de petite taille, ce qui est le cas des nœuds des arbres de transitions (cf. 4.2.2, p. 93). De plus, dans tous les cas, cette structure renforce encore une fois la *localité* des données : lorsque l'on parcourt une liste de collision à la recherche d'un élément, l'accès à l'objet courant et au pointeur sur l'objet suivant se feront très probablement dans la même page mémoire.

D'autres structures de listes chaînées permettent encore de réduire la consommation mémoire, mais au prix d'une augmentation du temps d'accès aux données (tables *fermées*, par exemple). Pour cela, nous n'avons pas retenu de telles structures.

4.2.5 Gestionnaire de tas *ad hoc*

Notre moteur d'analyse explicite de circuits est implémenté en `C++`. Dans ce langage, l'allocation et la désallocation de blocs mémoire sont explicitement gérées *via* un gestionnaire de tas. Le gestionnaire de tas par défaut doit notamment respecter trois contraintes majeures :

– il ne doit pas faire d'hypothèse quant à la taille des blocs à allouer ;

- il ne doit pas faire d'hypothèse quant à la durée de vie des blocs ;
- il doit minimiser la fragmentation du tas : des blocs adjacents libérés devraient pouvoir être réutilisés pour fournir un bloc formé de tous ces blocs libres adjacents.

Le respect de ces contraintes a pour conséquence que l'allocation et la désallocation de blocs mémoire sont des processus coûteux en temps et, paradoxalement, en mémoire. On estime généralement qu'un gestionnaire de tas standard alloue deux mots supplémentaires à chacun des blocs demandés (8 octets sur des architectures 32 bits) pour ses propres besoins. Ces mots permettent notamment de chaîner les blocs libres entre eux : cette liste chaînée de blocs libres est parcourue linéairement lors de la demande de nouveaux blocs pour trouver le bloc libre le plus adéquat. La défragmentation du tas, autrement dit la fusion de blocs libres adjacents, est régulièrement activée par le gestionnaire de tas selon certains critères, ou manuellement par invocation de la fonction de défragmentation.

En ce qui nous concerne, l'usage que notre moteur d'analyse explicite de circuits fait de la mémoire présente certaines particularités :

- à l'exception notable des tableaux des tables d'adressage dispersé (cf. 4.2.4, p. 96), dont la taille croit avec le nombre d'éléments référencés, les différentes tailles de blocs manipulés sont en nombre limité ;
- à l'exception notable des vecteurs de bits identifiant les états, les blocs que nous allouons sont de petite taille (cellules de liste chaînées, nœuds de graphe de transition, etc.) : nous aimerions donc éviter de perdre 2 mots mémoire par bloc alloué ;
- la durée de vie d'une grande proportion des blocs alloués termine avec le programme : vecteurs d'états, nœuds de graphe de transition, etc. ;
- les blocs à durée de vie limitée sont généralement rapidement réutilisés : cellules de listes chaînées, nœuds de graphe de transition créés puis détruits parce qu'un exemplaire existait déjà en mémoire (cf. 4.2.2, p. 93), etc.

Le gestionnaire de tas standard impose donc un surcoût en temps et en mémoire pour respecter des contraintes qui ne s'appliquent guère à notre moteur d'analyse explicite de circuits. Dans le but de réduire tout surcoût lié à la gestion de la mémoire, nous avons donc implémenté notre propre gestionnaire de tas.

Notre gestionnaire de tas se place au-dessus du gestionnaire de tas standard, afin de s'abstraire des problèmes de portabilité. Notre gestionnaire de tas ne fait appel au gestionnaire de tas standard que pour lui demander de gros blocs d'une dizaine de kilo-octets : des *pools*. Ces *pools* sont ensuite découpés en petits blocs de la même taille, multiples de la granularité de l'architecture sur laquelle le programme s'exécute (sur une machine 32 bits : 4 octets, 8 octets, etc.). Ces petits blocs sont initialement utilisés comme des pointeurs pour les chaîner entre eux et ainsi former une liste de blocs libres. Enfin, les listes de blocs libres sont accessibles par un tableau en fonction de la taille des blocs. La Figure 4.12 représente la structure d'un tel gestionnaire de tas.

Lorsque notre moteur demande un bloc mémoire, celui-ci est directement disponible en tête de liste, en fonction de sa taille. Lorsque notre moteur libère un bloc mémoire, celui-ci est directement inséré en tête de liste, toujours en fonction de sa taille. Allocation et désallocation mémoire sont donc des processus extrêmement peu coûteux en temps et

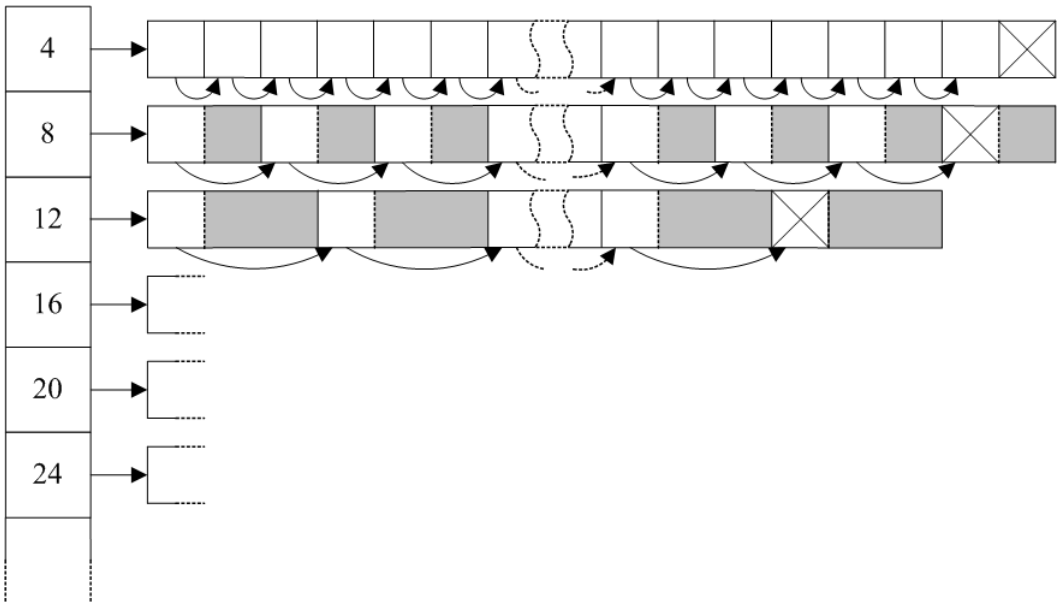


Figure 4.12: Structures des *pools* de notre gestionnaire de tas

avec une surconsommation mémoire minime, réduite au tableau des listes de blocs libres. Seule l'allocation d'un nouveau bloc alors que la liste de blocs libres correspondante est vide coûte en temps.

Notre gestionnaire de tas est *rétenseur* : une fois qu'un *pool* est créé, il ne peut fournir que des blocs d'une taille spécifique, et les *pools* ne peuvent être rendus au système que lorsque tous les blocs qu'ils fournissaient ont été libérés. Comme nous l'avons vu, ces contraintes sont tout à fait tolérables pour notre moteur.

Notre gestionnaire de tas évite donc les doubles chaînages habituellement mis en œuvre dans les gestionnaires de tas standards et réduit le coût en temps (amorti) des allocations et désallocations de blocs mémoire. Néanmoins, il alloue des *chunks* d'une dizaine de kilo-octets exclusivement dédiés à la fourniture de petits blocs de taille spécifique. Pour la génération d'automates ne comportant qu'un faible nombre de nœuds, notre gestionnaire de tas impose donc une surconsommation de mémoire qui n'est pas contrebalancée pas l'économie des doubles chaînages. Le gain en mémoire est néanmoins notable sur les modèles de plus grande taille. Enfin, la réduction en temps des allocations et désallocations de blocs mémoire est notable quelle que soit la taille du modèle.

Le Tableau 4.5 présente quelques mesures des apports de notre gestionnaire de tas sur différents modèles. La surconsommation de mémoire peut être importante sur les petits modèles, mais cela ne concerne que des quantités de mémoire insignifiantes. On y gagne en échange une importante réduction du temps de calcul, mais qui ne s'applique encore qu'à des durées insignifiantes. Pour les modèles de tailles plus conséquentes, le gain en mémoire peut être tout à fait notable.

modèle	∅	états	nœuds	alloc. standard		alloc. *ad hoc*			
				temps	mém.	temps		mém.	
`Wristwatch`	9	41	1 350	0.05s	56Ko	0.04s	-20%	128Ko	+129%
`ATDS-100-C2`	15	81	267	0.09s	20Ko	0.04s	-56%	172Ko	+760%
`TCINT`	9	310	3 151	0.43s	184Ko	0.36s	-16%	248Ko	+35%
`Renault`	11	161	9 451	0.64s	428Ko	0.48s	-25%	484Ko	+13%
`Testbench`	242	243	81	2.10s	300Ko	1.97s	-6%	340Ko	+13%
`TI`	181	652 948	1 007 969	3h 31	179Mo	3h 25	-3%	144Mo	-20%

Tableau 4.5: Apports du gestionnaire de tas *ad hoc*

4.3 Vérification formelle explicite

A partir de notre moteur d'évaluation explicite de circuit (cf. 4.1, p. 76), il est relativement aisé de greffer des fonctionalités de vérification formelle. Pour vérifier des propriétés de sûreté, soit l'émission ou la non-émission de signaux spécifiques, il suffit de se brancher sur les portes correspondant à ces signaux, et de réagir comme il se doit à leur valuation.

Afin de générer des séquences d'entrées menant aux bugs ainsi découverts, il convient de mémoriser, pour chaque état connu du système, l'état qui a permis sa découverte, ainsi que le vecteur de signaux d'entrées menant à cet état. La génération de séquences d'entrées se fait alors en temps linéaire par rapport à la profondeur de l'erreur détectée. L'analyse des états du système se faisant en largeur (*depth-first*), la longueur des séquences d'entrées conduisant à la violation d'une propriété est *de facto* minimale.

Nous proposons dans notre outil la vérification formelle de programmes selon le principe des observateurs (cf. 2.10, p. 44). Nous nous limitons donc aux propriétés de sûreté, pour les raisons mentionnés en 2.10.2 (p. 45).

4.3.1 Réduction de la consommation mémoire

En terme de mémoire, les fonctionnalités de vérification formelle nécessitent l'ajout de deux informations à chaque état connu : l'index de l'état *précurseur* (l'état ayant découvert le nouvel état en premier), et le vecteur d'entrées permettant la transition à partir de l'état précurseur.

Ces vecteurs portent généralement sur un faible nombre d'entrées. Aussi, le moteur d'analyse explicite de circuit cherche à minimiser le nombre d'entrées à valuer pour passer d'un état à un autre. Dès lors, il est fort probable que les mêmes vecteurs se retrouvent fréquemment entre deux états. Cette tendance est d'autant plus forte sur les systèmes très "linéaires" (comme les *testbenches*), pour lesquels le passage d'un état à un autre se fait en ne testant qu'un très petit nombre d'entrées, voire aucune.

Afin de réduire la consommation mémoire du système, les vecteurs d'entrées sont donc automatiquement partagés et n'existent qu'en instance unique dans le système, *via* une table d'adressage dispersé.

Ces vecteurs d'entrées n'étant utilisés que lors de la génération de séquences d'entrées, donc lors de la découverte d'une erreur, il est inutile de les conserver en mémoire durant tout le temps de l'analyse. Il serait donc envisageable de ne les stocker que sur disque, pour ne les consulter qu'en cas réel de besoin. La taille des données à stocker étant la même pour chaque état, et chaque état étant linéairement indexé, cela s'implémente très facilement *via* des fichiers structurés directement accessibles en mémoire (*memory mapped files*).

On pourrait aussi envisager de ne pas conserver les vecteurs d'entrées et de ne mémoriser, pour chaque état, qu'une référence vers l'état précurseur. La génération de séquences d'entrées s'effectuerait alors en analysant de nouveau les états, ou en déléguant totalement la tâche à un SAT Solver.

Néanmoins, le surcoût mémoire induit par le stockage des vecteurs d'entrées étant généralement assez faible par rapport à la taille de la table des états connus, ces approches n'ont pas été évaluées.

4.3.2 *Sweeping* structurel à la volée

A l'inverse de la génération d'automate, qui nécessite de traiter la totalité du circuit, la vérification formelle conduit généralement à se focaliser sur une propriété, ou un groupe de propriétés. Le support de ces propriétés (leur cône d'influence) est généralement un sous-ensemble restreint du circuit initial. Les portes non influentes doivent donc être ignorées.

L'analyse des portes non influentes est faite à la volée, directement lors de la construction de notre représentation interne du circuit à partir du fichier d'entrée. Le graphe des portes est construit à partir des signaux à surveiller, et les portes non influentes sont automatiquement non visitées. Cette optimisation revient à un *sweeping* structurel du circuit, mais effectué *a priori*.

Cette optimisation est très importante puisque, encore une fois, moins le moteur d'évaluation explicite a de portes à stabiliser, moins il a, en règle générale, de branchements à effectuer.

4.4 Génération de séquences de tests exhaustives

A partir de notre générateur d'automates (cf. 4.2, p. 92), un outil de génération de séquences de tests exhaustives a été développé par une des équipes de consultants d'Esterel Technologies. Cet outil, un script `Perl` assez conséquent, repose sur l'analyse des automates explicites générés.

Les expériences menées ont été concluantes bien que cette approche soit loin d'être optimale. En effet, générer un automate explicite nécessite entre autres d'ordonnancer, dans chaque transition, les actions à exécuter. Pour la génération de séquences de tests, les informations relatives aux actions exécutées par le programme sont inutiles. Le surcoût nécessaire à leur calcul devrait donc être évité.

Dès lors, il semble plus adéquat de baser la génération de séquences de tests sur une approche hybride implicite/explicite de l'exploration d'espaces d'états atteignables. Cette

approche est étudiée en 5.4 (p. 110).

4.5 Conclusion

Nous avons présenté dans ce chapitre un moteur d'exploration des états atteignables d'un circuit basé sur des techniques explicites. Les états sont analysés individuellement par simulation de la propagation de l'information dans le circuit, qui est stabilisé par branchements récursifs sur les entrées. Cet algorithme étant exponentiel dans le pire des cas, ce moteur intègre différentes techniques et heuristiques visant à réduire le risque d'explosion en temps. De plus, ce moteur a été développé selon plusieurs stratégies d'implémentations qui lui confèrent de très bonnes performances pour une consommation mémoire réduite.

Ce moteur a tout d'abord été utilisé à des fins de génération d'automates explicites. Le générateur d'automates utilisant ce moteur fait partie du compilateur `Esterel` v5, commercialisé au sein de l'environnement de développement intégré `Esterel Studio`. Ce moteur a par la suite été utilisé à des fins de vérification formelle, et il a rendu possible ou nettement accéléré la vérification formelle de modèles sur lesquels les techniques implicites ou les analyseurs de satisfiabilité échouaient ou peinaient.

Tel que, ce moteur d'analyse explicite d'espace d'états atteignables de circuits souffre de limitations intrinsèques concernant l'espace de stockage de la table des états connus et le temps d'analyse des états individuels.

Nous avons vu des stratégies d'implémentations visant à minimiser la taille de la table des états connus. Il existe dans la littérature de nombreux travaux permettant d'aller plus loin, notamment autour des outils destinés à l'analyse des processus exécutés en concurrence mais de manière asynchrone, comme `SPIN` ou `Mur`φ. Mentionnons par exemple les travaux basés sur le stockage de la table des états connus non plus en mémoire centrale mais sur disque dur [Ste97, SD98]. Afin de réduire encore plus l'espace occupé par cette table, l'approche probabiliste visant à n'utiliser qu'un ou deux bits par état (*bit-state hashing*) a fait l'objet de nombreuses publications [Hol88, Hol95]. Elle a également été fortement critiquée et sa validitée est très douteuse dans le cas général. On s'intéressera de préférence aux travaux moins agressifs qui proposent de compresser les vecteurs de bits identifiant les états par hachage [SD95b, SD96]. Notre outil gagnerait encore à intégrer les résultats de ces différentes études.

Enfin, pour ce qui est de réduire le coût d'analyse des états, nous proposons de traiter cela en abandonnant le principe de stabilisation du circuit par branchements récursifs sur les entrées, de ne plus propager de valeurs booléennes issues de ces branchements mais de propager directement des BDDs. Cette étude fait l'objet du chapitre suivant.

Chapitre 5

Approche hybride implicite/explicite

L'approche purement implicite du calcul d'espace d'états atteignables souffre de plusieurs défauts :

- la progression des calculs est très peu prévisible ;
- les calculs d'image sont aisément susceptibles d'exploser en temps et surtout en mémoire ;
- les performances sont très dépendantes de l'ordonnancement initial des variables ;
- les modèles au comportement très linéaire, donc avec beaucoup de registres redondants, sont difficilement analysables (cf. 2.9.4, p. 42) ;
- cette approche ne s'applique qu'aux circuits acycliques.

De son côté, l'approche purement explicite du calcul d'espace d'états atteignables ne présente pas ces inconvénients : la progression des calculs est beaucoup plus prédictible, il n'y a pas de risque imprévisible d'explosion en mémoire et cette approche est insensible aux modèles au comportement linéaire ainsi qu'aux circuits cycliques. Néanmoins, l'approche purement explicite souffre de trois défauts majeurs :

- les états sont analysés individuellement ;
- la complexité d'analyse d'un état est potentiellement exponentielle ;
- lorsqu'à la fois les approches implicites et explicites ont des comportements corrects, le débit d'analyse d'états de l'approche explicite est généralement beaucoup plus faible ;
- la taille de la table des états connus étant fonction du produit du nombre d'états par le nombre de variables d'états du modèle, plus le modèle comporte de variables d'états, moins on pourra conserver d'états.

Nous nous proposons d'adresser certains défauts de l'approche purement explicite en abandonnant le principe d'analyse des états par branchements récursifs sur les entrées et en utilisant des BDDs. Plutôt que de propager des valeurs booléennes issues de prises de décisions, nous propageons des BDDs représentant les différentes combinaisons d'entrées possibles. Nous conservons le principe d'analyse d'états individuels : les variables des registres sont ainsi des constantes et n'apparaissent pas dans les BDDs manipulés. Cette technique nous permet de réduire drastiquement le nombre de variables présentes dans les

BDDs, par rapport à une approche purement explicite, et ainsi de réduire le risque d'explosion en mémoire. La manipulation symbolique de différentes combinaisons d'entrées permet de repousser les limites du domaine d'application des techniques purement explicites, en réduisant le risque d'explosion en temps dus aux branchements récursifs.

5.1 Algorithme de base

L'analyse hybride implicite/explicite de circuits reprend deux principes de base de l'analyse purement explicite :

- les états sont analysés individuellement ;
- les états connus sont stockés de manière explicite, sous la forme de leur vecteur de variables d'états, dans une table d'adressage dispersé.

De même, nous reprenons le principe de stabilisation du circuit, comme le ferait le courant électrique, par propagation de valeurs de portes, en respect du principe de constructivité des circuits `Esterel`. La différence majeure est que nous ne propageons plus des valeurs booléennes, mais des BDDs (Algorithme 5.1). L'analyse d'un état se fait en deux temps :

1. Stabilisation du circuit :
   (a) Propagation de l'état, autrement dit propagation des valeurs des registres caractérisant l'état.
   (b) Propagation de BDDs atomiques associés aux entrées.

   Nous obtenons ainsi pour chaque registre un BDD référençant des variables d'entrées.
2. Analyse du vecteur de BDDs de registres pour extraire les nouvelles valuations possibles de registres, décrivant autant d'états atteignables. Les nouveaux états sont alors ajoutés à la fin de la queue d'états à analyser.

L'algorithme termine lorsqu'il n'y a plus d'états à analyser.

```
1  Propager les constantes
2  Tant qu'il y a des états non analysés dans la file
3    Retirer un état non analysé de la file et le propager
4    Tant que le circuit n'est pas stabilisé
5      Si il existe au moins un input non traité
6        Alors choisir un input et le propager
7        Sinon le circuit n'est pas constructif
8    Analyser le vecteur de BDDs des registres et
     ajouter les nouveaux états dans la file
```

Algorithme 5.1: Algorithme de base du moteur d'évaluation hybride de circuits

5.1.1 Analyse des vecteurs de BDDs des registres

L'analyse des vecteurs de BDDs des registres se fait en parcourant récursivement les BDDs du vecteur en parallèle, comme le montre la Figure 5.1 : la branche $\neg I1 \wedge \neg I2$ permet la découverte de l'état $(1, 1, 0, 0)$, la branche $\neg I1 \wedge I2$ celle de l'état $(0, 1, 0, 1)$, la branche $I1 \wedge \neg I2$ celle de l'état $(0, 0, 1, 0)$ et enfin la branche $I1 \wedge I2$ permet la découverte de l'état $(0, 1, 1, 0)$.

Tant que les BDDs du vecteur ne sont pas tous terminaux —des constantes—, nous recherchons la plus grande variable de BDDs parmi les variables aux sommets. Cette variable nous sert alors de point de branchement pour effectuer une récursion sur les deux vecteurs de BDDs formés des cofacteurs par rapport à cette variable. Une fois les vecteurs uniquement constitués de BDDs terminaux, le vecteur de bits identifiant le nouvel état est trivialement disponible.

Notons que, comme nous ne cofactorisons les BDDs qu'avec des variables au sommet, ces cofactorisations ne créent pas de nouveaux nœuds et se font en temps constant (ce sont en fait de simples décompositions de BDDs).

5.1.2 Complexité de l'algorithme

Nous reprenons les métriques utilisées pour l'analyse de complexité de l'algorithme purement explicite (cf. 4.1.3, p. 77). Encore une fois, nous négligerons la phase de propagation des constantes.

Les BDDs référençant des entrées ont, dans le pire des cas, un nombre de nœuds exponentiel d'ordre $O\left(2^{inputs}\right)$. Les conjonctions et disjonctions de BDDs ont un coût quadratique par rapport aux nombres de nœuds des BDDs impliqués. La phase de stabilisation du circuit (propagation des registres et entrées) a donc un coût dans le pire des cas d'ordre

$$O\left(littéraux * 2^{2*inputs}\right)$$

L'analyse des vecteurs de BDDs associés aux registres nécessite, dans le pire des cas, de parcourir en parallèle des BDDs de taille 2^{inputs}, d'où une complexité d'ordre

$$O\left(registres * 2^{inputs}\right)$$

La complexité totale de l'algorithme, dans le pire des cas, est donc d'ordre

$$O\left(états * \left(littéraux * 2^{2*inputs} + registres * 2^{inputs}\right)\right)$$

Si cette complexité semble assez rédhibitoire, rappelons que :
- la phase de propagation des registres n'implique que des BDDs constants : les manipulations de BDDs de cette phase se font donc toutes en temps constant ;
- la phase de propagation des registres permet généralement de valuer un grand nombre de littéraux ;
- les BDDs n'ont que très rarement un nombre de nœuds maximal ;
- chaque porte ne dépend que rarement d'un grand nombre d'entrées ;

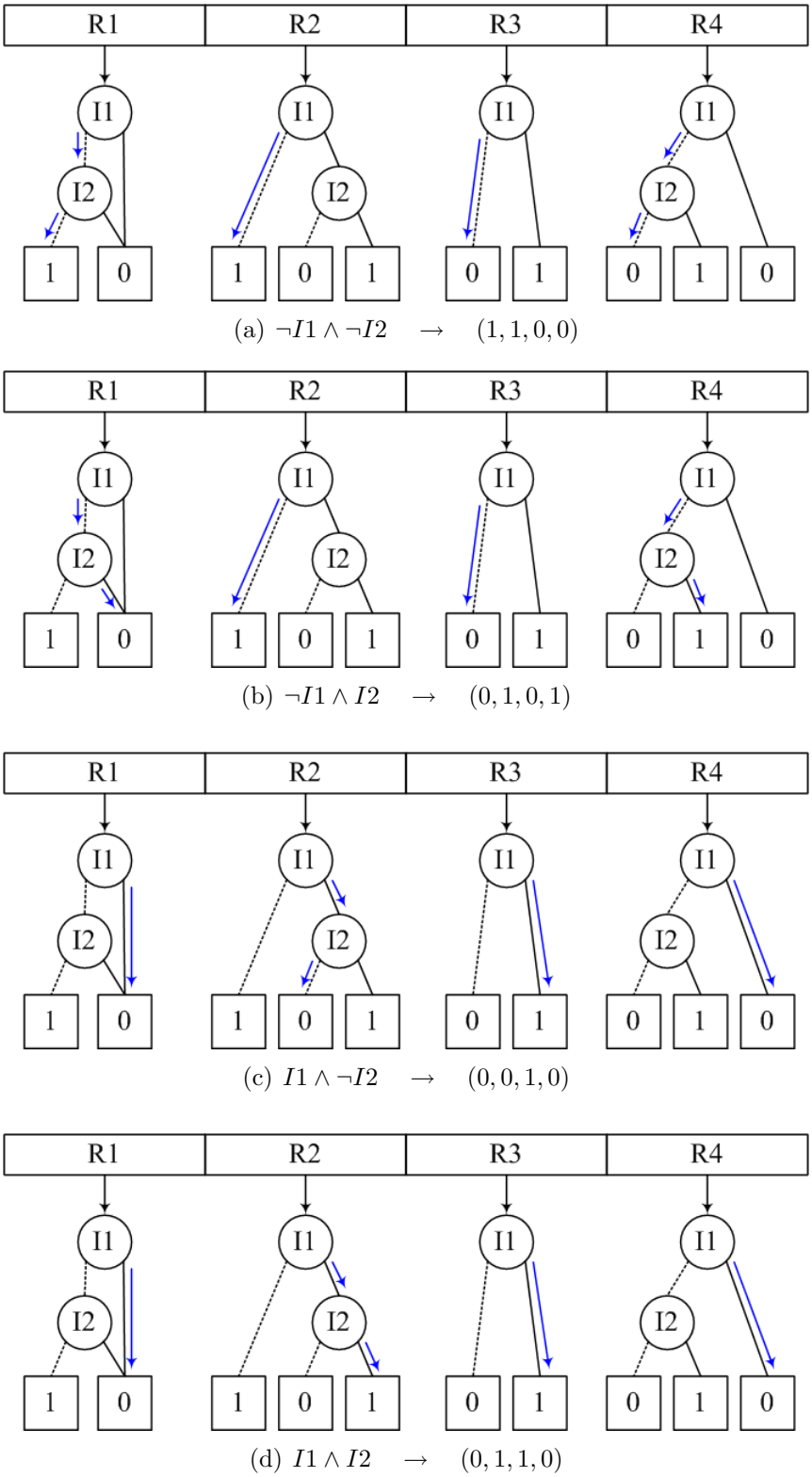


Figure 5.1: Analyse des vecteurs de BDDs des registres

– chaque état ne découvre généralement qu'un petit nombre d'états ;
– le nombre d'états atteignables est limité par l'*input care set* du modèle.

Enfin, rappelons que les résultats des manipulations de BDDs sont stockés en mémoire cache afin d'éviter les calculs redondants, ce qui réduit encore les coûts des calculs en pratique.

5.1.3 Optimisations

Nous reprenons dans le moteur hybride implicite/explicite la majorité des optimisations déjà adoptées pour le moteur purement explicite. La mise en œuvre des optimisations suivantes diffère légèrement.

Utilisation des relations entre les entrées du circuit

Dans le moteur purement explicite, les contraintes entre les signaux d'entrées spécifiées par le développeur sont converties en graphe de relations entre ces signaux (cf. 4.1.4, p. 78).

Dans le moteur hybride implicite/explicite, ces contraintes sont converties en un BDD d'*input care set*, comme pour le moteur purement implicite. Ce BDD est utilisé pour ignorer les nouveaux états accessibles par une combinaison d'entrées invalide (ligne 8 de l'Algorithme 5.1, p. 104). Ce BDD est aussi utilisé pour restreindre à la volée le nombre de nœuds des BDDs de valuation des portes (*via* l'opérateur `Restrict`).

Pondération des entrées du circuit

De même que dans le moteur purement explicite (cf. 4.1.4, p. 81), nous utilisons le nombre de portes importantes (outputs, registres, actions, ...) qu'une entrée est susceptible d'influencer comme heuristique de pondération.

Néanmoins, alors que cette pondération était recalculée à chaque état dans le moteur purement explicite, elle n'est calculée qu'une seule fois dans le moteur hybride, après propagation des constantes. L'ordonnancement des variables de BDDs est déduit de cette pondération, et le BDD d'*input care set* est construit avec cet ordre.

Recalculer une pondération des entrées spécifique à chaque état pourrait permettre de stabiliser le circuit plus tôt, en ayant moins d'entrées à propager. Néanmoins, modifier l'ordonnancement des variables nécessite de mettre à jour tous les BDDs "vivants" du système, à savoir le BDD d'*input care set* et tous les BDDs résultant de calculs passés et conservés en mémoire cache. Le changement de l'ordonnancement des variables est connu pour être généralement très coûteux[1]. Une autre option serait de détruire tous les BDDs vivants à la fin de chaque analyse d'état et de reconstruire le BDD d'*input care set*. Détruire les résultats de calculs précédents conservés en mémoire cache ferait perdre un des plus

[1]Cela revient à faire des substitutions de variables, ce qui a potentiellement une complexité exponentielle. Si l'ordonnancement des variables doit être modifié, il ne l'est généralement que par de légers décalages ou permutations locales (*sifting*).

importants apports des BDDs. Nous n'avons pas évalué expérimentalement l'utilité de telles approches.

Préservation de l'état du circuit entre les branchements

Les mécanismes mis en œuvre pour pouvoir restaurer l'état des portes est exactement le même (cf. 4.1.5, p. 88). Néanmoins, comme il n'y a plus de branchements à effectuer, les portes ne sont désormais susceptibles d'être sauvegardées qu'une seule fois par état.

Mécanisme d'évaluation rapide des portes logiques

Le support des valeurs *contrôlantes* n'est pas aussi direct dans le moteur hybride que dans le moteur purement explicite (cf. 4.1.5, p. 90), puisque ce ne sont plus des constantes booléennes qui sont manipulées mais des BDDs.

Nous gérons donc les valeurs contrôlantes sous la forme de *valeurs puits* (*faux* pour une porte `et`, *vrai* pour une porte `ou`) : dès que le BDD d'une porte atteint une valeur puits, cette valeur est propageable.

5.2 Génération d'automates

Pour la génération d'automates, nous maintenons une liste de couples (index d'action, BDD d'activation) : dès qu'une porte d'action devient définitivement valuée, nous ajoutons le couple correspondant à la fin de cette liste.

La construction de l'arbre de transition d'un état se fait en parcourant cette liste en plus du vecteur de BDDs des registres. Pour respecter la causalité des actions, il est nécessaire d'injecter dans l'arbre de transition les actions dans l'ordre dans lequel les portes ont été évaluées. Tant que cette liste n'est pas exclusivement constituée de BDDs constants, nous parcourons celle-ci récursivement, en générant des branches d'arbre de transition au fur et à mesure.

Notons que la nécessité de respecter la causalité des actions implique de cofactoriser tous les BDDs en fonction de la variable au sommet du premier BDD d'action non constant. Les cofactorisations nécessitent généralement la création de nouveaux nœuds de BDDs. S'il n'y avait pas d'actions à gérer, comme nous l'avons vu en 5.1.1 (p. 105), nous nous contenterions de décomposer les BDDs par rapport à la plus grande variable de BDDs parmi les variables aux sommets, les décompositions de BDDs ayant un coût constant et ne créant pas de nouveaux BDDs.

5.2.1 Résultats expérimentaux

Le Tableau 5.1 présente quelques mesures permettant de comparer les moteurs purement explicite et hybride implicite/explicite pour la génération d'automates. Globalement, l'approche hybride implicite/explicite ne semble guère intéressante pour cet usage. Dans la plupart des cas, le nombre de nœuds d'arbres de transition, le temps d'analyse et la

consommation mémoire augmentent. Nous estimons que cela est inhérent à un trop grand coût et à de trop fortes contraintes relatives au traitement des actions.

modèle	∅	états	moteur purement explicite			moteur hybride implicite/explicite					
			nœuds	temps	mém.	nœuds		temps		mém.	
Arbiter12	12	13	288	0.01s	172Kb	288	=	0.18s	×18	328Kb	×2
Wristwatch	9	41	1 350	0.04s	128Kb	1 358	≈	0.04s	≈	328Kb	×2.5
ATDS-100-C2	15	81	267	0.04s	172Kb	427	+60%	0.50s	×12	328Kb	×2
TCINT	9	310	3 151	0.36s	248Kb	1 847	-41%	0.41s	+14%	340Kb	+37%
Renault	11	161	9 451	0.48s	484Kb	11 819	+25%	0.87s	+25%	716Kb	+48%
Testbench	242	243	81	1.97s	340Kb	81	=	2.26s	+15%	256Kb	-25%
TI	181	652 948	1 007 969	3h 21	144Mb	1 076 393	+7%	4h 03	+18%	147Mb	+2%

Tableau 5.1: Génération d'automates avec les moteurs purement explicite et hybride

5.3 Vérification formelle

La vérification formelle n'est pas alourdie par les contraintes imposées par la génération d'automates. Non seulement il n'y a pas d'actions à ordonnancer, mais il est possible de simplifier le circuit en fonction des signaux de sortie à vérifier (*Transitive Network Sweeping*, cf. 3.1.1, p. 49).

Comme dans le moteur purement explicite, la vérification formelle dans le moteur hybride implicite/explicite se fait en scrutant certains signaux de sortie et en générant des séquences d'entrées menant à la violation de la propriété le cas échéant. Nous n'avons donc à mémoriser, par état, qu'une référence vers l'état précurseur et un vecteur d'entrées menant de l'état précurseur vers le nouvel état. Comme dans le vérifieur formel purement explicite, ces vecteurs d'entrées sont partagés et n'existent qu'en instance unique dans le système.

A l'heure actuelle, nous n'avons pu tester ce vérifieur formel hybride que sur des modèles au comportement très ou totalement linéaire. Sur ces modèles, la plupart des BDDs manipulés n'ont que très peu de nœuds, voire sont constants. Or, le coût des opérations logiques de base sur des BDDs quasi-constants étant nettement plus important que le coût de ces mêmes opérations directement effectuées par le processeur sur des variables booléennes. De ce fait, la version hybride du vérifieur formelle est plus lente que la version purement explicite sur ces modèles. Dans une première expérience (cf. 6.6, p. 139), le ralentissement est de l'ordre de 25% (2h 33 → 3h 09) pour une augmentation de la consommation mémoire de l'ordre de 5% (104 Mo → 110 Mo). Dans une seconde expérience (cf. 6.7, p. 147), le ralentissement est de l'ordre de 12% (1.8s → 1.6s) pour une consommation mémoire négligeable quelle que soit l'approche. Des comparaisons de performances sur des modèles au comportement moins linéaire sont donc encore à réaliser.

5.4 Génération de séquences de tests exhaustives

A partir de notre moteur hybride implicite/explicite d'exploration d'espace atteignables, nous avons également développé un prototype d'outil permettant la génération de séquences de tests exhaustives. De manière similaire à l'outil basé sur une approche purement implicite présenté en 3.9 (p. 71), nous proposons trois critères de couverture :

- Couverture d'états : tous les états atteignables du système sont visités au moins une fois par les séquences.
- Couverture de transitions : toutes les transitions possibles d'un état à un autre sont visitées au moins une fois par les séquences.
- Couverture d'outputs : tous les chemins menant à l'émission d'un ou plusieurs signaux de sortie spécifiques sont empruntés par les séquences.

Ainsi, pour le programmes `AB_O` (Programme 2.3, p. 23), des séquences de tests exhaustives pour les différents objectifs de couverture pourraient être les suivantes (le point-virgule sépare les instants) :

- Couverture d'états : `[;A;B;] , [;AB;] , [;B;]`
- Couverture de transitions : `[;;A;B;] , [;AB;] , [;B;A;]`
- Couverture de l'output `O` : `[;A;B;] , [;AB;] , [;B;A;]`

L'approche hybride implicite/explicite donne accès à des informations de granularité plus fine sur la structure des transitions entre états que l'approche purement implicite. En effet l'approche purement implicite ne permet que de savoir s'il existe une transition entre deux états et cela au prix d'un doublement du nombre de variables d'états impliquées dans les calculs d'espaces d'états atteignables. A l'inverse, l'approche hybride implicite/explicite permet de connaître toutes les transitions possibles entre deux états, soit un niveau de granularité *micro-step*. De ce fait, l'approche hybride implicite/explicite de la couverture de transitions ou d'outputs peut être encore plus complète.

Comme pour la vérification formelle, la génération de séquences de tests ne nécessite pas d'ordonnancer les actions effectuées par le programme. Dans le cadre de la couverture d'états ou d'outputs, les portes du circuit qui mentionnent des actions sont purement ignorées. Seules les portes mentionnant l'émission de signaux d'outputs à couvrir sont prises en compte pour la couverture d'outputs, mais il suffit de savoir qu'un output à couvrir peut être émis ou non. Comme pour la vérification formelle, ne pas avoir à ordonnancer les actions permet de n'utiliser que des décompositions de BDDs, au coût constant, et jamais de cofactorisations, au coût exponentiel dans le pire des cas.

5.4.1 Algorithmes

Quel que soit l'objectif de couverture choisi, les algorithmes mis en œuvre sont basés sur la construction d'un graphe de transitions inverses. Une transition inverse indique une valuation de signaux d'entrée ou de tests ayant permis une transition d'un état à un autre. Les séquences de tests sont générées en parcourant le graphe de transitions inverses à partir des états de plus grande profondeur, en remontant vers l'état initial.

Néanmoins, la sélection des transitions inverses pertinentes, leur mode de construction et enfin la manière dont le graphe de transitions inverses est parcouru afin de générer les séquences de tests diffère selon l'objectif de couverture choisi.

Couverture d'états

Pour la couverture d'états, nous construisons un graphe de transitions inverses très simple, en ne conservant qu'une seule transition inverse entre chaque paire d'états connectés : la première transition ayant découvert l'état. Cette première transition inverse est généralement très simple —elle ne mentionne que très peu de signaux d'entrées— puisque nous commençons toujours par évaluer les branches dans lesquelles les signaux sont absents.

Le graphe de transitions inverses est alors parcouru en partant des états les plus profonds et en remontant vers l'état initial. A chaque état, la sélection de la transition inverse à emprunter est effectuée selon un algorithme glouton visant à favoriser les transitions inverses menant à un état non déjà couvert.

Couverture de transitions

Pour la couverture de transitions, nous conservons toutes les transitions inverses possibles entre chaque paire d'états connectés.

Le graphe de transitions inverses est parcouru en partant des états ayant au moins une transition inverse vers un des états de plus grande profondeur. Similairement à la couverture d'états, la sélection des transitions inverses à emprunter est effectuée selon un algorithme glouton visant à favoriser les transitions inverses non déjà couvertes.

Couverture d'outputs

Pour la couverture d'outputs, nous ne conservons que les transitions inverses dans lesquelles sont émis les signaux de sortie à couvrir, ou menant à des états possédant de telles transitions.

Les graphe de transitions inverses est alors parcouru à partir des transitions inverses menant aux états de plus grande profondeur. Cette fois-ci, aucune heuristique gloutonne n'est mise en œuvre, puisque tous les chemins conduisant à l'émission des signaux de sortie spécifiés doivent être générés.

5.4.2 Résultats expérimentaux

Seule la couverture des états atteignables permet de comparer le générateur de séquences de tests basé sur une approche purement implicite (développé par Amar Bouali au sein d'Esterel Technologies, cf. 3.9, p. 71) et le nôtre, basé sur une approche hybride implicite/explicite. En effet, l'approche hybride implicite/explicite donnant accès à des informations de granularité plus importante concernant la structure des transitions entre états, le nombre de transitions différentes entre chaque paire d'états connectés peut être plus important. En conséquence, la couverture de transitions ou d'outputs devrait générer

un nombre de séquences de tests plus important. Cela n'est pas le cas pour la couverture d'états où le nombre de séquences de tests ne devrait guère différer quelle que soit l'approche choisie.

Le Tableau 5.2 présente quelques mesures permettant de comparer les performances de ces deux outils pour la génération de séquences de tests exhaustives couvrant l'ensemble des états atteignables d'un modèle. Rappelons que notre outil n'est qu'un prototype dont les performances sont certainement améliorables, alors que l'outil basé sur une approche purement implicite est plus mature.

Nous pouvons constater que, quel que soit le modèle, l'approche hybride implicite/explicite est systématiquement plus rapide, les temps de calculs étant divisés par un facteur allant de 3 à plus de 80. L'outil de génération de séquences de tests basé sur une approche purement implicite n'étant pas instrumenté pour rapporter la consommation totale de mémoire, nous ne pouvons pas fournir de comparaisons à ce propos. Nous avons toutefois constaté que l'approche hybride implicite/explicite permettait également de réduire la consommation mémoire. Enfin, le nombre des séquences produites est similaire dans la plupart des cas —ce qui était attendu— sauf 3. Une légère modification de l'algorithme de parcours du graphe de transitions inverse devrait permettre d'améliorer cela.

(Le modèle `Arbiter12` (cf. 4.1.4, p. 81) étant cyclique, il ne peut être directement traité par l'outil basé sur une approche purement implicite. Le modèle `Testbench` (cf. 6.7, p. 147) étant purement linéaire, il comporte de nombreux registres redondants à chaque étape du calcul de l'espace d'états atteignables, ce qui ralentit extrêmement ces calculs par une approche purement implicite.)

modèle	états	approche implicite			approche hybride				
		temps (s)	mém. (Mo)	# séqu.	temps (s)		mém. (Mo)	# séqu.	
`NDA#5`	10	0.10	*n/a*	4	0.03	÷3.5	8	4	=
`Arbiter12`	13	modèle cyclique			0.03		8	1	
`NDA#2`	21	0.09	*n/a*	8	0.03	÷3.5	8	8	=
`Wristwatch`	41	1.39	*n/a*	16	0.09	÷16	8	16	=
`NDA#4`	65	0.74	*n/a*	63	0.07	÷11	8	63	=
`ATDS-100-C2`	81	3.58	*n/a*	35	0.10	÷36	8	37	106%
`Renault`	161	13.57	*n/a*	99	0.35	÷39	9	105	106%
`Testbench`	243	tué après ≫ 1h			3.16		11	1	
`TCINT`	310	33.36	*n/a*	140	0.39	÷86	9	140	=
`NDA#1`	535	35.57	*n/a*	307	0.47	÷76	9	308	≈
`NDA#3`	875	16.57	*n/a*	462	0.43	÷39	9	489	106%

Tableau 5.2: Couverture d'états purement implicite ou hybride implicite/explicite

5.5 Conclusion

Nous avons présenté dans ce chapitre un moteur d'exploration de l'espace d'états atteignables d'un circuit basé sur des techniques hybrides implicites/explicites.

Comme pour le moteur purement explicite, les états sont analysés individuellement. Néanmoins, les états ne sont plus analysés par branchements récursifs sur les signaux d'entrée et par propagation de constantes booléennes issues de ces branchements mais directement par propagation de BDDs référençant les signaux d'entrée.

Cette stratégie hybride nous permet de conserver les avantages de l'approche énumérative, à savoir la prévisibilité du coût et de la progression des calculs, un comportement insensible aux registres redondants et le support transparent des circuits cycliques. De plus, cette stratégie hybride nous permet de réduire en pratique le coût d'analyse des états individuels : quand bien même la complexité dans le pire des cas de l'approche hybride semble plus rédhibitoire que celle de l'approche purement explicite, l'approche hybride bénéficie de nombreux résultats de travaux antérieurs sur les BDDs visant à réduire le coût effectif des manipulations de BDDs. Enfin, les BDDs manipulés ne référencent que les variables associées aux signaux d'entrée, à l'inverse des techniques purement implicites avec lesquelles les BDDs référencent également les variables d'états. La restriction du nombre de variables sur lesquels les BDDs opèrent nous permet de réduire les risques d'explosion en mémoire inhérents aux BDDs.

Ce moteur hybride implicite/explicite d'exploration d'espaces d'états atteignables à tout d'abord été appliqué à la génération d'automates explicites. Dans ce cadre, l'approche hybride permet de résoudre le problème de la détection *a posteriori* des tests inutiles, grâce à la représentation canonique qu'offrent les BDDs des fonctions représentées. Néanmoins, l'approche hybride ne semble guère appropriée à la génération d'automates explicites. Cela est principalement dû à la nécessité, afin de respecter l'ordonnancement et la causalité des actions de manipulations de données, de cofactoriser les BDDs là où de simples décompositions pourraient être utilisées.

Nous avons également appliqué ce moteur hybride implicite/explicite à la vérification formelle de propriétés de sûreté. Les exemples sur lesquels nous avons conduit nos expérimentations sont de taille trop restreinte ou au comportement trop linéaire pour que les gains apportés par l'utilisation de BDDs puissent compenser et dépasser le surcoût qu'ils imposent lorsque les fonctions manipulées sont très simples. Néanmoins, nous pensons que la vérification formelle par une approche hybride implicite/explicite devrait être plus efficace que par une approche purement explicite sur des modèles de taille plus conséquente ou au comportement moins linéaire.

Enfin, nous avons appliqué notre moteur hybride implicite/explicite à la génération de séquences de tests exhaustives. Bien que notre générateur de tests ne soit encore qu'à l'état de prototype, nos expérimentations ont déjà révélées des performances bien meilleures, s'approchant parfois de deux ordres de grandeurs, qu'un outil basé sur une approche implicite et notablement plus mature.

Chapitre 6

Expérimentations

Ce chapitre présente une partie des modèles utilisés pour nos expérimentations.

Nous détaillons les résultats d'expériences pour ceux sur lesquels nos outils exhibent des comportements représentatifs ou notables.

6.1 Description des modèles

`Arbiter12`

Ce modèle correspond à l'arbitre de bus de Robert de Simone, présenté en 4.1.4 (p. 81), avec 12 cellules.

Le système de priorité tournante des stations mis en œuvre dans ce modèle souligne notamment l'intérêt de la pondération des entrées, également présentée en 4.1.4 (p. 81).

`ATDS-100-C2`

Ce modèle correspond à un pilote de vidéo.

`Carburant`

Ce modèle, réalisé par Dassault Aviation, correspond au système de gestion du carburant de l'avion bi-réacteur Mirage 2000-9.

Ce modèle se prête tout à fait aux techniques d'abstractions vues au Chapitre 3.

Les expériences que nous avons réalisées sur ce modèle sont détaillées en 6.4 (p. 118).

`FWS`

Ce modèle correspond au moniteur d'alarme de l'A380, le futur avion de ligne d'Airbus.

Comme le modèle précédent, `Carburant`, ce modèle se prête tout à fait aux techniques d'abstractions vues au Chapitre 3.

Les expériences que nous avons réalisées sur ce modèle sont détaillées en 6.5 (p. 125).

TCINT

Ce modèle correspond au pilote de bus `Turbo Channel` d'un processeur Alpha, utilisé pour gérer l'interface entre une carte FPGA Perl et l'Alpha.

Renault

Cette application correspond à une antique modélisation du tableau de bord de la Renault R11 Electronic.

Testbench

Ce modèle est un stimulateur parfaitement linéaire d'une application industrielle.

Tous ses registres étant mutuellement exclusifs, ce modèle est déraisonnablement long à vérifier à l'aide de techniques implicites. Son extrême simplicité en fait néanmoins un candidat idéal pour les techniques explicites.

Les expériences que nous avons réalisées sur ce modèle sont détaillées en 6.7 (p. 147).

TI

Ce modèle correspond à une nouvelle architecture de bus de données arborescent de Texas Instruments.

Parce qu'il comporte un grand nombre de registres redondants, les techniques implicites échouent très tôt à vérifier ce modèle. Parce qu'il est très profond, les analyses de satisfiabilité échouent de même. Par contre, parce qu'il a un comportement assez linéaire et un nombre d'états relativement restreint, ce modèle se prête parfaitement aux analyses explicites.

Les expériences que nous avons réalisées sur ce modèle sont détaillées en 6.6 (p. 139).

Wristwatch

Cette application est la modélisation d'une montre-poignet digitale réalisée par Gérard Berry, un exemple standard livré avec le compilateur `Esterel` et `Esterel Studio`.

6.2 Machine de test

Toutes les mesures ont été effectuées sur une machine bi-Pentium III cadencés à 1 Ghz avec 2 Go de mémoire et le système d'exploitation Linux 2.2.20.

Cette machine était réinitialisée avant chaque séquence d'expériences et —à notre connaissance— personne n'était logé sur cette machine durant nos tests, et seuls nos tests étaient exécutés.

6.3 Légendes

- Les lignes en vert indiquent une terminaison correcte de l'outil ainsi qu'un résultat correct.
- Les lignes en orange indiquent une terminaison correcte de l'outil mais un résultat faux dû à une sur-approximation excessive.
- Les lignes en rouge indiquent une terminaison incorrecte de l'outil due à l'épuisement de la mémoire disponible.
- Méthodes de calcul de l'espace d'états atteignables :
  - *RSS=1* : toutes les combinaisons de variables d'état sont considérées atteignables ; seules sont prises en considération les relations d'implications ou d'exclusions entre les signaux d'entrées éventuellement spécifiées par l'utilisateur.
  - *RSS=regtree* : nous utilisons les relations d'exclusions entre registres spécifiées par l'arbre de sélection (cf. 2.3.5, p. 28) comme seule restriction de l'espace d'états atteignable.
  - *exact* : méthode usuelle de calcul de l'espace d'états atteignable (cf. 2.9, p. 38).
  - *inputization* : en fonction d'indications fournies par des personnes maîtrisant le modèle, certaines variables d'états sont remplacées par des entrées libres (cf. 3.3.1, p. 51).
  - *inp. + rt as ics* : les informations provenant de l'arbre de sélection sont utilisées pour renforcer les contraintes entre les variables d'états remplacées par des entrées libres (cf. 3.4.2, p. 62).
  - *inp. + rt as orss* : les informations provenant de l'arbre de sélection sont utilisées comme borne supérieure de l'espace d'états atteignable (cf. 3.4.3, p. 63).
  - *inp. + rt as ics+orss* : les informations provenant de l'arbre de sélection sont conjointement utilisées à la fois pour renforcer les contraintes entre les variables d'états remplacées par des entrées libres et comme borne supérieure de l'espace d'états atteignable.
  - *abstraction* : en fonction d'indications fournies par des personnes maîtrisant le modèle, certaines variables d'états sont abstraites et l'espace d'états atteignables du modèle est calculé à l'aide d'une logique tri-valuée (cf. 3.3.2, p. 53).
  - *abs. + rt as orss* : les informations provenant de l'arbre de sélection sont utilisées comme borne supérieure de l'espace d'états atteignable.

6.4 Carburant

6.4.1 Description

Cette application modélise la gestion de carburant de l'avion bi-réacteur Mirage 2000-9 de Dassault Aviation [BNLD00].

Le système est composé de 5 réservoirs internes, de 5 bidons larguables en vol et de deux nourrices, qui servent de tampons pour alimenter les réacteurs. La tâche de cette application est de gérer une alimentation en carburant correcte des nourrices, en prenant notamment en compte :

- d'éventuelles défaillances de composants systèmes (pompes, gauges, moteurs, pressurisation, etc.) ;
- du maintient d'une bonne répartition du carburant entre les deux parties de l'avion afin d'en garantir la stabilité ;
- de l'ejection d'urgence de carburant ;
- du ravitaillement en vol ;
- des interventions du pilote.

En outre, l'application évalue régulièrement les quantités de carburant restantes, afin de détecter d'éventuelles défaillances des gauges et prévenir le pilote lorsque des seuils d'alertes sont atteints.

La modélisation en `SyncCharts` et en `Esterel++` —une évolution objet d'`Esterel` réalisée par Dassault Aviation— est liée à une interface graphique développée en Java, présentée par la Figure 6.1. Cette interface permet notamment de déclencher les diverses pannes envisagées, afin de vérifier de manière informelle les réactions du système.

L'application `Esterel` générée comprend 28 modules et 69 instances de modules pour près de 5 000 lignes de code. Le modèle déclare 72 signaux d'entrée et 147 signaux de sortie. Le circuit `Esterel` généré comporte plus de 9 000 portes, plus de 17 000 littéraux et plus de 500 registres.

6.4.2 Calcul d'espace d'états complet

Le calcul de l'espace d'états atteignables du système intégral échoue très tôt sur une machine disposant de 2 Go de mémoire. Seule la première itération, qui découvre 8 749 états en ne consommant pratiquement pas de mémoire, est complétée en moins d'une seconde. La seconde itération échoue après 26 minutes par épuisement de la mémoire disponible.

6.4.3 Vérification formelle

Ce modèle comporte 4 propriétés de sûreté correctes, les deux premières étant en fait purement combinatoires.

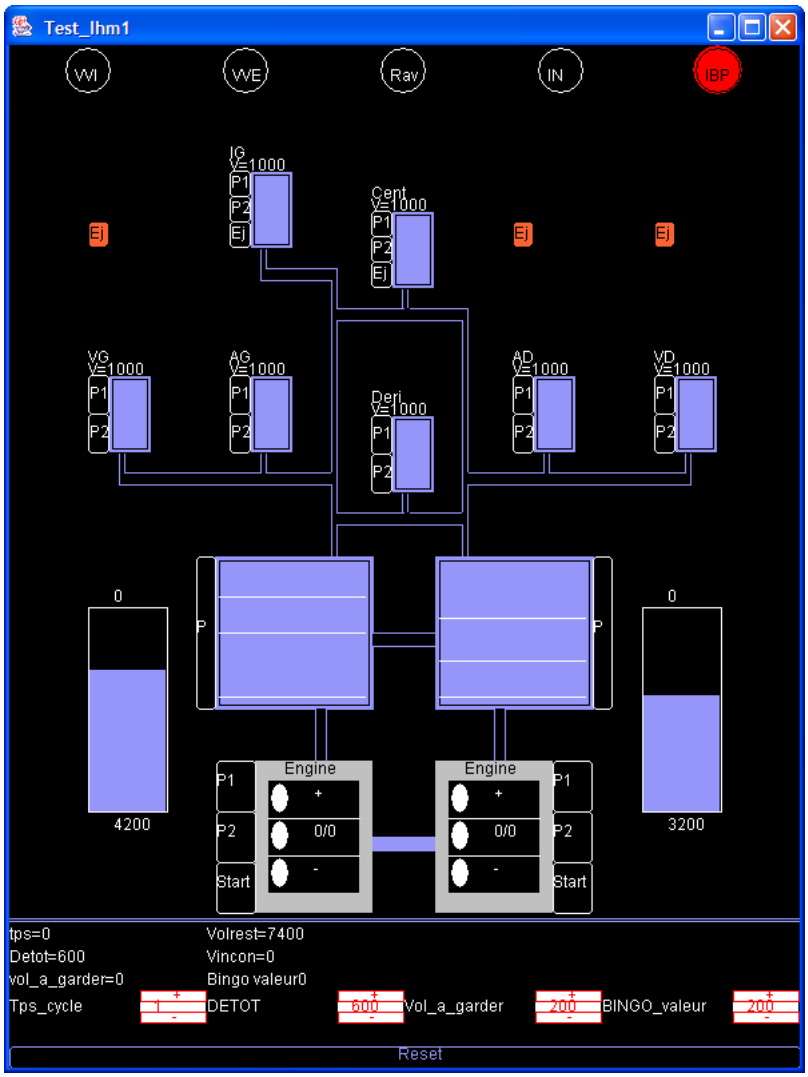


Figure 6.1: Interface graphique de l'application `Carburant`

La spécialisation du circuit à ces 4 propriétés par *sweeping* permet de supprimer 200 registres sur 500, mais cela ne rend pas pour autant le calcul d'espace d'états atteignables réalisable (Tableau 6.1).

Il faut donc soit décomposer la vérification formelle en 4 sessions —chacune focalisée sur une seule propriété— soit abstraire le modèle.

6.4.4 Abstraction du modèle

Nous avons contacté le responsable du projet chez Dassault Aviation, Emmanuel Ledinot, qui nous a immédiatement suggéré d'abstraire les bidons et les réservoirs. En effet, la plus grande partie de la logique de l'application est contenue dans les nourrices et les moteurs. Il était donc possible selon lui de relâcher complètement les comportements de ces modules sans que cela n'ait d'impact sur les propriétés.

Une telle abstraction permet de faire disparaître environ 300 registres. Par la suite, le *sweeping* permet de supprimer encore d'une soixantaine à la quasi-totalité des registres restants.

Propriétés 1, 2, 3 et 4

Nos expériences pour la vérification formelle des 4 propriétés à la fois sont synthétisées par le Tableau 6.1.

Le calcul d'espace d'états atteignables du modèle spécialisé par *sweeping* pour la vérification des 4 propriétés à la fois n'est toujours pas réalisable sur une machine disposant de 2 Go de mémoire.

Le remplacement des variables d'états de certains modules rend les 4 propriétés vérifiables en 35mn et en consommant une centaine de Mo.

L'utilisation des informations indiquées par l'arbre de sélection pour renforcer les relations entre les variables d'entrées triple le temps de calcul et double pratiquement la consommation de mémoire, sans toutefois réduire le cardinal de l'ensemble d'états atteignables. L'utilisation des informations indiquées par l'arbre de sélection comme borne supérieure de l'espace d'états atteignables n'a pas non plus d'impact sur le cardinal de l'espace, mais cela permet d'accélérer légèrement les calculs et de réduire la consommation mémoire d'un cinquième. L'utilisation conjointe des informations indiquées par l'arbre de sélection à la fois pour renforcer les relations entre les variables d'entrées et comme borne supérieure de l'espace d'états atteignables apporte les mêmes surcoûts que leur utilisation seule pour renforcer les relations entre les variables d'entrées.

L'abstraction des mêmes variables d'états à l'aide d'une logique trivaluée rend la vérification formelle des propriétés `P3` et `P4` possible. La vérification des propriétés `P1` et `P2` échoue du fait d'une sur-approximation excessive. Le calcul dure 1h 10 et nécessite plus de 450 Mo de mémoire. L'accroissement de la sur-approximation et l'effet "boule de neige" ont pour conséquence que deux itérations supplémentaires sont nécessaires pour atteindre le point fixe du calcul.

L'utilisation des informations indiquées par l'arbre de sélection comme borne supérieure de l'espace d'états atteignables permet de réduire le cardinal de l'espace d'états de près de 5 ordres de grandeur, sans toutefois rendre les deux premières propriétés vérifiables. Cela ramène néanmoins le temps d'exécution à une valeur proche des résultats de la technique précédente, la consommation mémoire étant 3.5 fois plus importante.

méthode	temps	mem.	nombre total d'états atteignables à la profondeur					
			1	2	3	4	5	6
RSS = 1	0.7s	1Mo						
RSS = regtree	56s	68Mo						
exact	≈1h	☠	8 749	2.45e13				
inputization	35mn	103Mo	37	70 574	1.46e6	1.56e6		
inp. + rt as ics	1h 43	192Mo						
inp. + rt as orss	32mn	83Mo						
inp. + rt as ics+orss	1h 39	190Mo						
abstraction	1h 10	456Mo		1.35e15	5.62e18	2.44e21	3.12e21	3.12e21
abs. + rt as orss	38mn	286Mo		1.18e11	2.29e14	4.76e16	7.95e16	8.22e16

Tableau 6.1: `Carburant` : Vérification des propriétés 1, 2, 3 et 4

Propriété 1

Nos expériences pour la vérification formelle de la propriété 1 sont synthétisées par le Tableau 6.2.

Cette propriété est purement combinatoire et il est possible d'en construire le BDD sans la moindre information de restriction, en une fraction de seconde et pratiquement sans consommer de mémoire.

Utiliser les informations indiquées par l'arbre de sélection comme espace d'états atteignables nécessite 40s de calculs et 63 Mo de mémoire.

Vérifier cette propriété sur un circuit spécialisé par *sweeping* nécessite plus de 5mn de calculs, 26 Mo de mémoire et 6 itérations.

Remplacer les variables d'état des bidons et des réservoirs par des entrées libres permet de diviser le nombre d'itérations nécessaires par 2 (6 → 3), le temps de calcul par 23 (5mn → 13s) et la consommation mémoire par plus de 6 (26 Mo → 4 Mo).

Utiliser les informations indiquées par l'arbre de sélection pour renforcer les relations entre les variables d'entrées et/ou comme borne supérieure de l'espace d'états atteignables n'a pas d'impact sur le cardinal de l'ensemble d'états et peut augmenter légèrement le temps de calcul ou la consommation mémoire.

Abstraire les variables d'état des bidons et des réservoirs à l'aide d'une logique trivaluée fait échouer la vérification de la propriété, du simple fait de la perte de corrélation entre différentes occurences d'une ou de plusieurs variables abstraites, puisque cela échoue dès la première itération alors que l'ensemble des d'états trouvés atteignables à cette profondeur n'est pas plus important que celui trouvé par la technique précédente. La vérification de

cette propriété échoue néanmoins en une fraction de seconde et en n'ayant pratiquement pas consommé de mémoire.

Utiliser les informations indiquées par l'arbre de sélection comme borne supérieure de l'espace d'états atteignables ne fait qu'augmenter légèrement le temps de calcul et la consommation de mémoire, sans pour autant permettre la vérification de la propriété.

méthode	temps	mem.	nombre total d'états atteignables à la profondeur					
			1	2	3	4	5	6
RSS = 1	0.1s	≈0						
RSS = regtree	40s	63Mo						
exact	>5mn	26Mo	8 749	5.57e11	1.5e16	3.93e16	4.60e16	5.14e16
inputization	13s	4Mo						
inp. + rt as ics	15s	5Mo	37	878	2 953			
inp. + rt as orss	13s	4Mo						
inp. + rt as ics+orss	15s	5Mo						
abstraction	0.1s	≈0						
abs. + rt as orss	0.2s	<0.5Mo	37					

Tableau 6.2: `Carburant` : Vérification de la propriété 1

Propriété 2

Nos expériences pour la vérification formelle de la propriété 2 sont synthétisées par le Tableau 6.3.

Les constatations que nous pouvons faire concernant la propriété 2 sont similaires à celles de la propriété 1, les temps de calculs et les consommations mémoire étant néanmoins notablement inférieurs.

méthode	temps	mem.	nombre total d'états atteignables à la profondeur				
			1	2	3	4	5
RSS = 1	≪0.1s	≈0					
RSS = regtree	0.5s	3Mo					
exact	>1mn	10Mo	8 749	3.02e8	1.34e13	3.34e13	3.68e13
inputization	1s	<0.5Mo					
inp. + rt as ics	1.5s	1.5Mo	37	341	3 738		
inp. + rt as orss	1.2s	0.5Mo					
inp. + rt as ics+orss	1.7s	1.5Mo					
abstraction	0.2s	≈0	37				
abs. + rt as orss		<0.5Mo					

Tableau 6.3: `Carburant` : Vérification de la propriété 2

Propriété 3

Nos expériences pour la vérification formelle de la propriété 3 sont synthétisées par le Tableau 6.4.

Cette propriété n'est pas combinatoire, mais tenter de la vérifier sans la moindre information de restriction est pratiquement instantané et sans consommation de mémoire.

Tenter de la vérifier avec les seules exclusions de variables d'états indiquées par l'arbre de sélection échoue aussi, mais en 6s et 48 Mo.

Le calcul exact de l'espace d'états atteignables du circuit spécialisé par *sweeping* à cette seule propriété se fait en 9mn, 73 Mo de mémoire et 5 itérations.

Remplacer les variables d'état des bidons et des réservoirs par des entrées libres permet de vérifier la propriété en seulement 3 secondes et 1.5 Mo de mémoire.

Utiliser les informations indiquées par l'arbre de sélection pour renforcer les relations entre les variables d'entrées et/ou comme borne supérieure de l'espace d'états atteignables n'a pas d'impact sur le cardinal de l'ensemble d'états et peut augmenter le temps de calcul ou la consommation mémoire.

Abstraire les variables d'état des bidons et des réservoirs à l'aide d'une logique trivaluée permet encore de réduire d'un tiers le temps de vérification formelle au prix d'une légère augmentation de la consommation de mémoire, bien que, du fait de la sur-approximation accrue, une itération supplémentaire soit nécessaire. Toujours du fait de cette sur-approximation accrue, le cardinal de l'espace d'états est supérieur de trois ordres de grandeur.

Utiliser les informations indiquées par l'arbre de sélection comme borne supérieure de l'espace d'états atteignables permet de ramener le cardinal de l'espace des états atteignable au même ordre de grandeur et de réduire la consommation de mémoire. Cette stratégie est optimale pour vérifier cette propriété.

<table>
<tr><th rowspan="2">méthode</th><th rowspan="2">temps</th><th rowspan="2">mem.</th><th colspan="5">nombre total d'états atteignables à la profondeur</th></tr>
<tr><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th></tr>
<tr><td>RSS = 1</td><td><0.1s</td><td>≈0</td><td colspan="5" rowspan="2"></td></tr>
<tr><td>RSS = regtree</td><td>6s</td><td>48Mo</td></tr>
<tr><td>exact</td><td>9mn</td><td>73Mo</td><td>8 749</td><td>3.02e8</td><td>1.34e13</td><td>3.34e13</td><td>3.68e13</td></tr>
<tr><td>inputization</td><td>3s</td><td>1.5Mo</td><td rowspan="5">37</td><td rowspan="3">341</td><td rowspan="3">3 738</td><td colspan="2" rowspan="3"></td></tr>
<tr><td>inp. + rt as ics</td><td>4.4s</td><td>3Mo</td></tr>
<tr><td>inp. + rt as orss</td><td>3.2s</td><td>1.5Mo</td></tr>
<tr><td>inp. + rt as ics+orss</td><td>4.8s</td><td>3.5Mo</td><td rowspan="3"></td></tr>
<tr><td>abstraction</td><td>1.9s</td><td><2Mo</td><td>2.71e5</td><td>9.49e6</td><td>9.52e6</td></tr>
<tr><td>abs. + rt as orss</td><td>2s</td><td><1.5Mo</td><td>1 670</td><td>6 807</td><td>7 407</td></tr>
</table>

Tableau 6.4: `Carburant` : Vérification de la propriété 3

Propriété 4

Nos expériences pour la vérification formelle de la propriété 4 sont synthétisées par le Tableau 6.5.

Les constatations que nous pouvons faire concernant la propriété 4 sont similaires à celles de la propriété 3, les temps de calculs et les consommations mémoire étant néanmoins notablement inférieurs.

Encore une fois l'abstraction de variable à l'aide d'une logique trivaluée est la solution optimale pour vérifier cette propriété. Utiliser les informations indiquées par l'arbre de sélection comme borne supérieure de l'espace d'états atteignables permet de réduire le cardinal de celui-ci au prix d'une très légère augmentation de la consommation mémoire, sans toutefois accélérer les calculs.

méthode	temps	mem.	nombre total d'états atteignables à la profondeur						
			1	2	3	4	5	6	7
RSS = 1	<0.1s	≈0							
RSS = regtree	0.8s	6Mo							
exact	2mn	15Mo		16 673	2.41e8	7.03e8	8.86e8	8.86e8	8.86e8
inputization	0.5s	≪0.5Mo							
inp. + rt as ics	1.2s	1.5Mo		70	229	245			
inp. + rt as orss	0.6s	<0.5Mo	2						
inp. + rt as ics+orss	1.2s	1.5Mo							
abstraction	0.4s	≪0.5Mo		4 337	1.08e6	2.42e6			
abs. + rt as orss		<0.5Mo		865	79 255	1.77e5			

Tableau 6.5: `Carburant` : Vérification de la propriété 4

6.4.5 Conclusions

Les deux premières propriétés de ce modèle sont purement combinatoires et trivialement prouvables sans calcul d'espace d'états atteignables. Notre technique d'abstraction de variables à l'aide d'une logique trivaluée échoue à vérifier ces propriétés mais en une fraction de seconde et avec une consommation de mémoire négligeable.

Pour les deux autres propriétés, notre technique d'abstraction de variables à l'aide d'une logique trivaluée permet leur vérification en temps minimum et avec une consommation mémoire minimale.

6.5 FWS

6.5.1 Description

Cette application modélise le moniteur d'alarme de l'A380, le futur avion de ligne d'Airbus à deux étages qui permettra de transporter plus de 500 passagers sur près de 15 000 kilomètres en 2006.

Dans les postes de pilotage modernes, les alarmes présentées aux équipages sont de natures très variées, des majeures (défaillance des réacteurs, du train d'atterrissage, du système de gestion du carburant, etc.) à d'autres dont la résolution peut être reportée.

Le système de gestion de ces alarmes vise à alerter les pilotes d'un dysfonctionnement et à les guider dans le traitement de ces alarmes. Leur réponse doit être fonction de l'importance relative de ces alarmes, qui sont susceptibles de survenir en même temps. Le système doit donc être capable de gérer une file d'attente dynamique d'alarmes à traiter, les plus importantes devant prendre le pas sur les autres. Dans les phases sensibles du vol —décollage et atterrissage— certaines alarmes peuvent être masquées et devront être de nouveau signalées dans les phases moins sensibles (croisière, etc.). Le système doit également permettre la gestion d'alarmes erronées.

Les alarmes transportent plusieurs niveaux d'information par le biais de sons, de signaux lumineux, d'affichage textuel sur des écrans ou de voix synthétisées décrivant la nature des problèmes. L'information doit être précise et concise, permettant ainsi aux équipages d'identifier et de confirmer les problèmes dans un premier temps, puis d'effectuer le traitement approprié dans un deuxième temps.

L'application `Esterel` générée comprend 6 modules et 10 instances de modules pour environ 900 lignes de code. Le modèle déclare 12 signaux d'entrée et 47 signaux de sortie. Aucune relation entre les signaux d'entrée n'est spécifiée. Le circuit `Esterel` généré comporte plus de 2 000 portes et 128 registres.

6.5.2 Calcul d'espace d'états complet

Le calcul de l'espace d'états atteignables du système intégral est relativement peu coûteux et s'effectue en 14 itérations, moins d'une demi-heure et en consommant 223 Mo de mémoire (cf. Tableau 6.6).

Le système complet a près de 45 millions d'états atteignables.

6.5.3 Vérification formelle

Ce modèle comporte 8 propriétés de sûreté correctes.

Les observateurs relatifs à ce modèle ont été développés séparément afin de permettre une vérification en boîte noire (cf. 3.6, p. 68). L'ajout des 8 observateurs, de leur logique et de leurs variables d'états n'augmente pas de manière sensible le coût de calcul de l'espace d'états atteignables (cf. Tableau 6.7).

La vérification en boîte noire, combinée avec une spécialisation par *sweeping* du circuit en fonction des propriétés à vérifier, ne permet néanmoins guère de supprimer de variables d'états : elles sont pratiquement toutes présentes dans les supports de chaque propriété. La décomposition de la vérification formelle en différentes sessions —chacune focalisée sur une seule propriété— n'apporte donc guère d'intérêt. La durée de chaque session n'étant inférieure que de quelques minutes à la durée de la session vérifiant toutes les propriétés à la fois, décomposer les vérifications nécessite pratiquement 8 fois plus de temps.

6.5.4 Abstraction du modèle

Afin de réduire le coût de vérification formelle du modèle, nous avons contacté son développeur, Lionel Blanc, qui, en quelques minutes, nous a indiqué quels étaient les modules dont le comportement ne devaient pas avoir d'impact sur les propriétés.

Ses indications étant différentes pour chacune des propriétés, nous présentons les résultats des expériences propriété par propriété.

Propriété 1

Nos expériences pour la vérification formelle de la propriété 1 sont synthétisées par le Tableau 6.8.

Le remplacement des variables d'états de certains modules par des entrées libres permet de réduire le temps de vérification formelle par un facteur supérieur à 90 (29mn → 19s), la mémoire étant réduite par un facteur supérieur à 45 (187 Mo → 4 Mo). Le nombre total d'itérations nécessaires est réduit de 13 à 9.

L'utilisation des informations provenant de l'arbre de sélection pour renforcer les relations entre les entrées ou borner la sur-approximation n'a aucun effet sur les cardinaux des ensembles d'états. Cela peut néanmoins augmenter légèrement le temps de calcul (19s → 19-22-23s) et cela double la consommation mémoire (4 → 9 Mo).

L'abstraction des variables à l'aide d'une logique trivaluée permet encore d'accélérer les calculs, mais réduit légèrement moins la mémoire consommée : le temps de calcul est divisé par 250 (25mn → 6s) et la mémoire est réduite par un facteur supérieur à 30 (187 Mo → 6 Mo).

L'utilisation des informations provenant de l'arbre de sélection pour borner la sur-approximation augmente légèrement le temps de calcul (6s → 8s) mais réduit le cardinal de l'espace d'états atteignables par un facteur proche de 5.

Propriété 2

Nos expériences pour la vérification formelle de la propriété 2 sont synthétisées par le Tableau 6.9.

Cette propriété est en fait purement combinatoire : elle ne dépend pas fonctionnellement des variables d'états. De plus, la construction du BDD de cette propriété s'avère

être triviale, même sans la moindre information de restriction. Cette propriété est donc immédiatement vérifiable avec une consommation de mémoire négligeable.

Par rapport à une vérification formelle avec calcul exact de l'espace d'états du système, le remplacement des variables d'états de certains modules par des entrées libres permet de réduire le temps de calcul par un facteur supérieur à 20 (25mn → 1mn 9s) et la consommation mémoire par un facteur supérieur à 13 (188 Mo → 14 Mo). Le nombre d'itérations nécessaires est réduit de 13 à 10.

L'utilisation des informations indiquées par l'arbre de sélection pour renforcer les relations entre les variables d'entrées augmente légèrement le temps de calcul et la consommation mémoire, mais divise pratiquement par 2 le cardinal de l'espace d'états atteignables. L'utilisation des informations indiquées par l'arbre de sélection pour borner la sur-approximation permet une amélioration très notable : le temps de calcul est divisé par 375 (25mn → 4s) et la consommation mémoire est divisée par 75 (188 Mo → 2.5 Mo) ; le nombre d'itérations est réduit de 13 à 4 et le cardinal de l'espace d'états est divisé par 65. L'utilisation conjointe des informations indiquées par l'arbre de sélection à la fois pour renforcer les relations entre les variables d'entrées et borner la sur-approximation offre les mêmes apports quant au temps de calcul et à la consommation mémoire, mais permet de réduire le cardinal de l'espace d'états atteignables de 2 ordres de grandeur.

L'abstraction des variables à l'aide d'une logique trivaluée a un intérêt mitigé : le temps de calcul est réduit d'un tiers mais la consommation mémoire est plus que doublée. Néanmoins, l'utilisation des informations indiquées par l'arbre de sélection pour borner la sur-approximation améliore cette apport : le temps de calcul est réduit par un facteur supérieur à 30 (25mn → 44s) et la consommation mémoire est divisée par 4 (188 Mo → 47 Mo).

Propriété 3_1

Nos expériences pour la vérification formelle de la propriété 3_1 sont synthétisées par le Tableau 6.10.

Le remplacement des variables d'états de certains modules par des entrées libres permet de réduire le temps de vérification formelle par 250 (25mn → 6s), la mémoire étant réduite par 90 (184 Mo → 2 Mo). Le nombre total d'itérations nécessaires est réduit de 13 à 10.

L'utilisation des informations provenant de l'arbre de sélection pour renforcer les relations entre les entrées triple pratiquement le temps de calcul, multiplie par 6 la consommation mémoire et ne réduit que d'un quart le cardinal de l'espace d'états. Néanmoins, l'utilisation des informations provenant de l'arbre de sélection pour borner la sur-approximation améliore notablement les résultats : le temps de calcul est divisé par plus de 400 (25mn → 3.5s) et la consommation mémoire par plus de 120 (184 Mo → 1.5 Mo), le cardinal de l'espace d'états étant plus que divisé par 2. L'utilisation conjointe des informations indiquées par l'arbre de sélection à la fois pour renforcer les relations entre les variables d'entrées et borner la sur-approximation est moins intéressante : le temps de calcul est divisé par 200, la consommation mémoire par 35 mais le cardinal de l'espace d'états est divisé par 3.

L'abstraction des variables à l'aide d'une logique trivaluée est moins performant : le temps de calcul est divisé par plus de 40 (25mn → 36s) et la mémoire par près de 6 (184 Mo → 31 Mo). Le cardinal de l'espace d'états est augmenté de trois ordres de grandeur.

L'utilisation des informations provenant de l'arbre de sélection pour borner la sur-approximation améliore légèrement ces performances : le temps de calcul est divisé par plus de 70 (25m → 21s), la consommation mémoire est divisée par 10 (184 Mo → 18 Mo) et le cardinal de l'espace d'états n'est multiplié "que" par 30.

Propriété 3_2

Nos expériences pour la vérification formelle de la propriété 3_2 sont synthétisées par le Tableau 6.11.

Le remplacement des variables d'états de certains modules par des entrées libres permet de diviser le temps de vérification formelle par près de 70 (25mn → 22s) en réduisant la consommation mémoire par plus de 40 (188 Mo → 4.5 Mo). Le nombre total d'itérations nécessaires est réduit de 13 à 10.

L'utilisation des informations provenant de l'arbre de sélection pour renforcer les relations augmente légèrement le temps de calcul, triple la consommation mémoire et ne réduit que d'un quart le cardinal de l'espace d'états. Néanmoins, l'utilisation des informations provenant de l'arbre de sélection pour borner la sur-approximation améliore notablement les résultats : le temps de calcul est divisé par près de 140 (25mn → 11s) et la consommation mémoire par plus de 60 (188 Mo → 3 Mo), le cardinal de l'espace d'états étant divisé par 4. L'utilisation conjointe des informations indiquées par l'arbre de sélection à la fois pour renforcer les relations entre les variables d'entrées et borner la sur-approximation améliore le temps de calcul, désormais divisé par 150 (25mn → 10s), mais augmente la consommation de mémoire, divisée par près de 35 (188 Mo → 5.5 Mo).

L'abstraction des variables à l'aide d'une logique trivaluée ne permet pas de vérifier cette propriété, du fait d'une sur-approximation excessive : la propriété est trouvée violable dès la seconde itération. La session échoue néanmoins qu'en une fraction de seconde et pratiquement sans consommer de mémoire.

L'utilisation des informations provenant de l'arbre de sélection pour borner la sur-approximation améliore légèrement ces performances : le temps de calcul est divisé par plus de 70 (25m → 21s), la consommation mémoire est divisée par 10 (184 Mo → 18 Mo) et le cardinal de l'espace d'états n'est multiplié "que" par 30.

Propriété 4_1

Nos expériences pour la vérification formelle de la propriété 4_1 sont synthétisées par le Tableau 6.12.

Le remplacement des variables d'états de certains modules par des entrées libres permet de diviser le temps de calcul par plus de 200 (25mn → 7s) en réduisant la consommation

mémoire par près de 75 (186 Mo → 2.5 Mo). Le nombre total d'itérations nécessaires est réduit de 13 à 10.

L'utilisation des informations provenant de l'arbre de sélection pour renforcer les relations double le temps de calcul, triple la consommation mémoire et ne réduit que d'un quart le cardinal de l'espace d'états. Néanmoins, l'utilisation des informations provenant de l'arbre de sélection pour borner la sur-approximation améliore notablement les résultats : le temps de calcul est divisé par 375 (25mn → 4s) et la consommation mémoire par plus de 90 (186 Mo → 2 Mo), le cardinal de l'espace d'états étant plus que divisé par 2. L'utilisation conjointe des informations indiquées par l'arbre de sélection à la fois pour renforcer les relations entre les variables d'entrées et borner la sur-approximation ramène le temps de calcul à 7s et la consommation mémoire à 4 Mo.

L'abstraction des variables à l'aide d'une logique trivaluée ne permet pas de vérifier cette propriété, du fait d'une sur-approximation excessive : la propriété est trouvée violable à la troisième itération. La session échoue néanmoins qu'en 3.5s et en ayant consommé 11 Mo.

L'utilisation des informations provenant de l'arbre de sélection pour borner la sur-approximation permet la vérification de cette propriété, mais moins rapidement qu'en remplaçant les variables d'états par des entrées libres : le temps de calcul est divisé par 55 (25m → 27s), la consommation mémoire est divisée par 7 (186 Mo → 26 Mo) mais le nombre d'itérations nécessaires est réduit à 8.

Propriété 4_2

Nos expériences pour la vérification formelle de la propriété 4_2 sont synthétisées par le Tableau 6.13.

Le remplacement des variables d'états de certains modules par des entrées libres permet de diviser le temps de calcul par près de 190 (25mn → 8s) en réduisant la consommation mémoire par plus de 60 (192 Mo → 3 Mo). Le nombre total d'itérations nécessaires est réduit de 13 à 10.

L'utilisation des informations provenant de l'arbre de sélection pour renforcer les relations double le temps de calcul, triple la consommation mémoire et ne réduit que d'un quart le cardinal de l'espace d'états. Néanmoins, l'utilisation des informations provenant de l'arbre de sélection pour borner la sur-approximation améliore notablement les résultats : le temps de calcul est divisé par 375 (25mn → 4s) et la consommation mémoire par plus de 90 (192 Mo → 2 Mo), le cardinal de l'espace d'états étant pratiquement divisé par 4. L'utilisation conjointe des informations indiquées par l'arbre de sélection à la fois pour renforcer les relations entre les variables d'entrées et borner la sur-approximation ramène le temps de calcul à 7s et la consommation mémoire à 4 Mo.

L'abstraction des variables d'une logique trivaluée ne permet pas de vérifier cette propriété, du fait d'une sur-approximation excessive : la propriété est trouvée violable à la troisième itération. La session échoue néanmoins qu'en 4s et en ayant consommé 10 Mo.

L'utilisation des informations provenant de l'arbre de sélection pour borner la sur-approximation permet la vérification de cette propriété, mais moins rapidement qu'en

remplaçant les variables d'états par des entrées libres : le temps de calcul est divisé par près de 55 (25m → 28s), la consommation mémoire est divisée par près de 9 (192 Mo → 22 Mo) mais le nombre d'itérations nécessaires est réduit à 8.

Propriété 5

Nos expériences pour la vérification formelle de la propriété 5 sont synthétisées par le Tableau 6.14.

Le remplacement des variables d'états de certains modules par des entrées libres permet de réduire le temps de vérification formelle par un facteur supérieur à 70 (25mn → 21s), la mémoire étant réduite par un facteur supérieur à 55 (195 Mo → 3.5 Mo). Le nombre total d'itérations nécessaires est réduit de 13 à 9.

L'utilisation des informations provenant de l'arbre de sélection pour renforcer les relations et/ou borner la sur-approximation n'a pas d'impact sur le cardinal de l'espace d'états atteignables, ni sur le nombre d'itérations. On note néanmoins un léger ralentissement des calculs (21s → 25s / 22s / 27s), mais une augmentation éventuellement importante de la consommation mémoire (3.5 Mo → 12.5 Mo / 4 Mo / 13 Mo).

L'abstraction des variables à l'aide d'une logique trivaluée accélère encore plus les calculs, réduits d'un facteur 100 (25mn → 15s), la consommation mémoire étant réduite d'un facteur 14 (195 Mo → 14 Mo). Le nombre d'itérations nécessaire est encore plus réduit et passe de 13 à 7.

L'utilisation des informations provenant de l'arbre de sélection borner la sur-approximation divise le cardinal de l'espace d'états atteignables par près de 4, mais augmente d'un quart les temps de calculs et d'un septième la consommation de mémoire.

Propriété 6

Nos expériences pour la vérification formelle de la propriété 6 sont synthétisées par le Tableau 6.15.

Le remplacement des variables d'états de certains modules par des entrées libres permet de réduire le temps de vérification formelle par un facteur proche de 250 (29mn → 7s), la consommation mémoire par un facteur supérieur à 90 (231 Mo → 2.5 Mo) et le nombre total d'itérations nécessaires de 13 à 10.

L'utilisation des informations provenant de l'arbre de sélection pour renforcer les relations entre les variables d'état remplacées par des entrées libres réduit d'un quart le cardinal de l'espace d'états atteignables, mais double pratiquement les temps de calculs et quadruple la consommation mémoire. L'utilisation des informations provenant de l'arbre de sélection pour borner la sur-approximation permet néanmoins de réduire encore le temps de calcul, qui est alors divisé par plus de 400 (29mn → 4s), la mémoire étant réduite par un facteur supérieur à 115 (231 Mo → 2 Mo) et le cardinal de l'espace d'états atteignables par plus de 2. L'utilisation conjointe des informations provenant de l'arbre de sélection à la fois pour renforcer les relations et borner la sur-approximation n'accélère pas les calculs et

double presque la consommation mémoire, mais divise par 3 le cardinal de l'espace d'états atteignables.

L'abstraction des variables à l'aide d'une logique trivaluée ne permet pas la vérification formelle de cette propriété, du fait d'une sur-approximation excessive. Les calculs cessent dès la seconde itération, en 3s et en ayant consommé 12 Mo de mémoire.

L'utilisation des informations provenant de l'arbre de sélection pour borner la sur-approximation permet de réduire le cardinal de l'espace d'états atteignables d'un ordre de grandeur et de réduire légèrement la consommation mémoire (12 Mo → 11 Mo), mais sans toutefois rendre la propriété vérifiable.

Propriété 7

Nos expériences pour la vérification formelle de la propriété 7 sont synthétisées par le Tableau 6.16.

Le remplacement des variables d'états de certains modules par des entrées libres ne réduit le temps de calcul "que" d'un facteur 22 (25mn → 1mn 8s) et la mémoire d'un facteur 14 (184 Mo → 13 Mo), le nombre d'itérations nécessaires passant de 13 à 10.

L'utilisation des informations provenant de l'arbre de sélection pour renforcer les relations ralentit légèrement les calculs et augmente légèrement la consommation de mémoire. Néanmoins, l'utilisation des informations provenant de l'arbre de sélection pour borner la sur-approximation divise le temps de calcul par 375 (25mn → 4s), la consommation mémoire par plus de 70 (184 Mo → 2.5 Mo) et réduit le nombre d'itérations nécessaires de 13 à 4 en divisant le cardinal de l'espace d'états atteignables par 65. L'utilisation des informations provenant de l'arbre de sélection pour renforcer les relations, en plus de borner la sur-approximation, n'altère pas de manière significative le temps de calcul ou la consommation de mémoire, mais réduit le cardinal de l'espace d'états atteignables de deux tiers.

L'abstraction des variables à l'aide d'une logique trivaluée rend la propriété vérifiable en 16 minutes, soit en ne réduisant le temps de calcul "que" d'un tiers, mais en doublant la consommation mémoire. La sur-approximation est telle que le cardinal de l'espace d'états, trouvé en 8 itérations au lieu de 13, est près de 700 fois supérieur.

L'utilisation des informations provenant de l'arbre de sélection pour borner la sur-approximation permet de réduire le cardinal de l'espace d'états de 4 ordres de grandeur et de ramener le temps de calcul à 13s, soit une réduction d'un facteur supérieur à 115, mais avec une consommation mémoire de 47 Mo, soit une réduction d'un facteur proche de 4.

6.5.5 Conclusions

Nos expériences pour la vérification formelle de ce modèle sont synthétisées par le Tableau 6.17.

Sur ce modèle, le remplacement de certaines variables d'états par des entrées libres permet, quelles que soient les propriétés, de réduire les temps de calculs et la consommation mémoire par d'importants facteurs, allant respectivement de 22 à 250 et de 13 à 92.

Quelle que soit la propriété, utiliser les informations provenant de l'arbre de sélection pour renforcer les relations entre les variables d'état remplacées par des entrées libres ralentit les calculs de 7% à 115% en augmentant la mémoire de 20% à 500%. Le cardinal de l'espace d'états est parfois réduit, mais toutes les propriétés étaient déjà vérifiables.

Il est beaucoup plus intéressant d'utiliser les informations provenant de l'arbre de sélection pour borner la sur-approximation, ce qui n'augmente le temps de calcul et la consommation mémoire (respectivement de 4% et 14%) que pour 1 propriété sur 9. Pour toutes les autres propriétés, à la fois les temps de calculs et la consommation mémoire sont encore réduits jusqu'à des facteurs respectivement supérieurs à 17 et 5.

Utiliser les informations provenant de l'arbre de sélection pour renforcer en plus les relations n'améliore légèrement le temps de calcul que pour une seule propriété (de 10%) en doublant pratiquement la consommation mémoire. Dans la plupart des cas, on retrouve l'augmentation à la fois des temps de calculs et de la consommation mémoire, respectivement de 0% à 115% et de 0% à 233%.

Sur ce modèles, les apports de l'abstraction de variables à l'aide d'une logique trivaluée sont plus mitigés. Pour 4 propriétés sur 9, une sur-approximation excessive empêche la vérification des propriétés, mais les calculs terminent en peu de temps et en ayant consommé peu de mémoire. Pour les 2 autres propriétés (2 et 7), les gains en temps et en mémoire sont modestes (33%) alors que la consommation de mémoire est multipliée par 2. Pour la propriété 3_1, les gains sont plus notables (temps de calcul divisé par 40 et consommation mémoire divisée par 6). Enfin, pour les propriétés 1 et 5, les gains en temps sont maximaux et dépassent la méthode consistant à remplacer des variables d'état par des entrées libres, quoi que la consommation de mémoire soit plus importante.

Utiliser les informations provenant de l'arbre de sélection pour borner la sur-approximation rend toutes les propriétés vérifiables sauf la propriété 6. Cela permet aussi de réduire les temps de calculs dans 3 cas sur 4, en les divisant par 22 dans 2 cas. La consommation mémoire est aussi réduite dans 3 cas sur 4 (elle est alors divisée par des facteurs entre 8 et 10).

méthode	temps	mem.	nombre total d'états atteignables à la profondeur													
			1	2	3	4	5	6	7	8	9	10	11	12	13	14
exact	29mn	223Mo	257	16 513	4.43e5	3.39e6	1.27e7	2.49e7	3.49e7	3.98e7	4.27e7	4.42e7	4.46e7	4.48e7	4.48e7	4.48e7

Tableau 6.6: `FWS` : Calcul de l'espace d'états du système intégral

méthode	temps	mem.	nombre total d'états atteignables à la profondeur												
			1	2	3	4	5	6	7	8	9	10	11	12	13
RSS = 1	0.1s	0.5Mo													
RSS = regtree	0.4s	3.5Mo													
exact	29mn	223Mo	257	16 513	5.79e5	4.94e6	2.04e7	4.39e7	6.36e7	7.30e7	7.72e7	7.87e7	7.89e7	7.89e7	7.89e7

Tableau 6.7: `FWS` : Vérification des propriétés P1 à P7

méthode	temps	mem.	nombre total d'états atteignables à la profondeur												
			1	2	3	4	5	6	7	8	9	10	11	12	13
RSS = 1	≈ 0s	≈ 0Mo													
RSS = regtree	≈ 0s	<0.5Mo													
exact	25mn	187Mo	257	16 513	4.76e5	4.29e6	1.88e7	4.18e7	6.13e7	7.06e7	7.48e7	7.63e7	7.65e7	7.65e7	7.65e7
inputization	19s	4Mo													
inp. + rt as ics	22s	9Mo		7 937	29 921	79 585	1.11e5	1.27e5	1.35e5	1.36e5	1.36e5				
inp. + rt as orss	19s	4Mo													
inp. + rt as ics+orss	23s	9Mo	129												
abstraction	6s	6Mo		2.98e5	4.14e6	9.99e6	1.17e7	1.26e7	1.26e7						
abs. + rt as orss	8s	6Mo		1.26e5	8.81e5	2.13e6	2.48e6	2.67e6	2.68e6						

Tableau 6.8: `FWS` : Vérification de la propriété P1

méthode	temps	mem.	nombre total d'états atteignables à la profondeur												
			1	2	3	4	5	6	7	8	9	10	11	12	13
RSS = 1	≈ 0s	≈ 0Mo													
RSS = regtree	0.4s	3Mo													
exact	25mn	188Mo	257	16 513	4.76e5	4.29e6	1.88e7	4.18e7	6.13e7	7.06e7	7.48e7	7.63e7	7.65e7	7.65e7	7.65e7
inputization	1mn 9s	14Mo		30 977	3.97e5	2.65e6	6.24e6	9.02e6	1.09e7	1.15e7	1.15e7	1.15e7			
inp. + rt as ics	1mn 14s	17Mo		20 737	2.46e5	1.58e6	3.67e6	5.28e6	6.38e6	6.68e6	6.71e6	6.71e6			
inp. + rt as orss	4s	2.5Mo		15 553	35 777	1.03e5									
inp. + rt as ics+orss	4s	2.5Mo		9 409	22 625	63 665									
abstraction	16mn	448Mo		1.85e5	2.47e8	5.98e9	2.01e10	3.56e10	4.51e10	5.12e10					
abs. + rt as orss	44s	47Mo		92 449	4.19e5	9.75e5									

Tableau 6.9: `FWS` : Vérification de la propriété P2

méthode	temps	mem.	nombre total d'états atteignables à la profondeur												
			1	2	3	4	5	6	7	8	9	10	11	12	13
RSS = 1 RSS = regtree	≈ 0s	≈ 0Mo													
exact	25mn	184Mo	257	16 513	4.76e5	4.29e6	1.88e7	4.18e7	6.13e7	7.06e7	7.48e7	7.63e7	7.65e7	7.65e7	7.65e7
inputization	6s	2Mo	65	13 313	67 585	1.43e5	2.36e5	2.93e5	3.20e5	3.31e5	3.38e5	3.39e5			
inp. + rt as ics	16s	12Mo		9 985	50 689	1.08e5	1.77e5	2.20e5	2.40e5	2.48e5	2.53e5	2.55e5			
inp. + rt as orss	3.5s	1.5Mo		12 289	46 849	81 025	1.16e5	1.32e5	1.40e5	1.44e5	1.47e5	1.48e5			
inp. + rt as ics+orss	7.5s	5Mo		9 217	35 137	60 769	86 977	99 073	1.05e5	1.08e5	1.10e5	1.11e5			
abstraction	36s	31Mo		1.51e6	7.69e7	5.91e8	7.60e8	8.52e8	9.00e8	9.01e8					
abs. + rt as orss	21s	18Mo		6.22e5	3.64e6	7.02e6	8.79e6	9.90e6	1.02e7	1.03e7					

Tableau 6.10: `FWS` : Vérification de la propriété P3_1

méthode	temps	mem.	nombre total d'états atteignables à la profondeur												
			1	2	3	4	5	6	7	8	9	10	11	12	13
RSS = 1	≈ 0s	≈ 0Mo													
RSS = regtree		0.5Mo													
exact	25m	188Mo	257	16 513	4.76e5	4.29e6	1.88e7	4.18e7	6.13e7	7.06e7	7.48e7	7.63e7	7.65e7	7.65e7	7.65e7
inputization	22s	4.5Mo	65	14 337	95 489	2.47e5	4.18e5	4.94e5	5.30e5	5.43e5	5.48e5	5.49e5			
inp. + rt as ics	24s	14Mo		10 753	71 617	1.85e5	3.13e5	3.71e5	3.97e5	4.07e5	4.11e5	4.12e5			
inp. + rt as orss	11s	3Mo		12 289	46 849	81 025	1.16e5	1.32e5	1.40e5	1.44e5	1.47e5	1.48e5			
inp. + rt as ics+orss	10s	5.5Mo		9 217	35 137	60 769	86 977	99 073	1.05e5	1.08e5	1.10e5	1.11e5			
abstraction	0.3s	0.5Mo	65	4.13e6											
abs. + rt as orss	23s	12Mo	65	8.29e5	7.41e6	1.96e7	3.41e7	4.11e7	4.87e7	5.20e7					

Tableau 6.11: `FWS` : Vérification de la propriété P3_2

méthode	temps	mem.	nombre total d'états atteignables à la profondeur												
			1	2	3	4	5	6	7	8	9	10	11	12	13
RSS = 1	≈ 0s	≈ 0Mo													
RSS = regtree															
exact	25m	186Mo	257	16 513	4.76e5	4.29e6	1.88e7	4.18e7	6.13e7	7.06e7	7.48e7	7.63e7	7.65e7	7.65e7	7.65e7
inputization	7s	2.5Mo	65	13 345	86 273	1.77e5	3.07e5	3.88e5	4.28e5	4.42e5	4.51e5	4.54e5			
inp. + rt as ics	15s	8.5Mo		10 017	64 705	1.33e5	2.31e5	2.91e5	3.21e5	3.32e5	3.38e5	3.40e5			
inp. + rt as orss	4s	2Mo		12 321	60 673	98 689	1.53e5	1.74e5	1.85e5	1.90e5	1.94e5	1.96e5			
inp. + rt as ics+orss	7s	4Mo		9 249	45 505	74 017	1.15e5	1.31e5	1.39e5	1.42e5	1.46e5	1.47e5			
abstraction	3.5s	11Mo	65	2.02e6	1.50e8										
abs. + rt as orss	27s	26Mo	65	8.29e5	4.08e6	8.12e6	1.05e7	1.18e7	1.21e7	1.23e7					

Tableau 6.12: `FWS` : Vérification de la propriété P4_1

méthode	temps	mem.	nombre total d'états atteignables à la profondeur												
			1	2	3	4	5	6	7	8	9	10	11	12	13
RSS = 1	≈ 0s	≈ 0Mo													
RSS = regtree		1Mo													
exact	25m	192Mo	257	16 513	4.76e5	4.29e6	1.88e7	4.18e7	6.13e7	7.06e7	7.48e7	7.63e7	7.65e7	7.65e7	7.65e7
inputization	8s	3Mo	65	14 369	1.17e5	2.84e5	5.10e5	6.14e5	6.58e5	6.77e5	6.82e5	6.84e5			
inp. + rt as ics	16s	9.5Mo		10 785	87 553	2.13e5	3.82e5	4.61e5	4.94e5	5.07e5	5.12e5	5.13e5			
inp. + rt as orss	4s	2Mo		12 321	60 673	98 689	1.53e5	1.74e5	1.85e5	1.90e5	1.94e5	1.96e5			
inp. + rt as ics+orss	8s	5Mo		9 249	45 505	74 017	1.15e5	1.31e5	1.39e5	1.42e5	1.46e5	1.47e5			
abstraction	4s	10Mo	65	4.19e6	2.00e9										
abs. + rt as orss	28s	22Mo	65	1.11e6	8.40e6	2.26e7	4.00e7	4.84e7	5.72e7	6.16e7					

Tableau 6.13: `FWS` : Vérification de la propriété P4_2

méthode	temps	mem.	nombre total d'états atteignables à la profondeur												
			1	2	3	4	5	6	7	8	9	10	11	12	13
RSS = 1	≈ 0s	≈ 0Mo													
RSS = regtree		0.5Mo													
exact	25m	195Mo	257	16 513	5.79e5	4.94e6	2.04e7	4.39e7	6.36e7	7.30e7	7.72e7	7.87e7	7.89e7	7.89e7	7.89e7
inputization	21s	3.5Mo	129	8 001	53 281	1.21e5	1.85e5	2.10e5	2.20e5	2.22e5	2.22e5				
inp. + rt as ics	25s	12.5Mo													
inp. + rt as orss	22s	4Mo													
inp. + rt as ics+orss	27s	13Mo													
abstraction	15s	14Mo		3.97e5	4.49e6	1.09e7	1.29e7	1.37e7	1.37e7						
abs. + rt as orss	19s	16Mo		1.68e5	1.27e6	3.08e6	3.64e6	3.87e6	3.88e6						

Tableau 6.14: `FWS` : Vérification de la propriété P5

méthode	temps	mem.	nombre total d'états atteignables à la profondeur												
			1	2	3	4	5	6	7	8	9	10	11	12	13
RSS = 1	≈ 0s	≈ 0Mo													
RSS = regtree		0.5Mo													
exact	29m	231Mo	257	16 513	4.76e5	4.29e6	1.88e7	4.18e7	6.13e7	7.06e7	7.48e7	7.63e7	7.65e7	7.65e7	7.65e7
inputization	7s	2.5Mo	65	13 345	86 273	1.77e5	3.07e5	3.88e5	4.28e5	4.42e5	4.51e5	4.54e5			
inp. + rt as ics	13s	9Mo		10 017	64 705	1.33e5	2.31e5	2.91e5	3.21e5	3.32e5	3.38e5	3.40e5			
inp. + rt as orss	4s	2Mo		12 321	60 673	98 689	1.53e5	1.74e5	1.85e5	1.90e5	1.94e5	1.96e5			
inp. + rt as ics+orss	7s	4Mo		9 249	45 505	74 017	1.15e5	1.31e5	1.39e5	1.42e5	1.46e5	1.47e5			
abstraction	3s	12Mo	65	2.02e7	8.72e7										
abs. + rt as orss		11Mo		8.29e5	8.84e6										

Tableau 6.15: FWS : Vérification de la propriété P6

méthode	temps	mem.	nombre total d'états atteignables à la profondeur												
			1	2	3	4	5	6	7	8	9	10	11	12	13
RSS = 1	≈ 0s	≈ 0Mo													
RSS = regtree		2.5Mo													
exact	25m	184Mo	257	16 513	4.76e5	4.29e6	1.88e7	4.18e7	6.13e7	7.06e7	7.48e7	7.63e7	7.65e7	7.65e7	7.65e7
inputization	1mn 08	13Mo		30 977	3.97e5	2.65e6	6.24e6	9.02e6	1.09e7	1.15e7	1.15e7	1.15e7			
inp. + rt as ics	1m 13	16Mo		20 737	2.46e5	1.58e6	3.67e6	5.28e6	6.38e6	6.68e6	6.71e6	6.71e6			
inp. + rt as orss	4s	2.5Mo		15 553	35 777	1.03e5									
inp. + rt as ics+orss				9 409	22 625	63 665									
abstraction	16m	388Mo		1.85e5	2.47e8	5.98e9	2.01e10	3.56e10	4.51e10	5.12e10					
abs. + rt as orss	43s	47Mo		92 449	4.19e5	9.75e5									

Tableau 6.16: FWS : Vérification de la propriété P7

P	exact		inputization										abstraction					
			standard			+ rt as ics			+ rt as orss			+ rt as ics+orss	standard			+ rt as orss		
	t.	m.	temps		mem.	temps		mem.	temps		mem.	temps		mem.	temps		mem.	
1	25m	187	19	÷80	4	÷47	22	÷70	9	÷21	19	÷80	4	÷45	23	÷65	9	÷21	6	÷250	6	÷32	8	÷190	6	÷32
2	25m	188	1'09	÷22	14	÷13	1'14	÷20	17	÷11	4	÷375	2.5	÷75	4	÷375	2.5	÷75	16m	-33%	448	×2.4	44	÷35	47	÷4
3_1	25m	184	6	÷250	2	÷92	16	÷95	12	÷15	3.5	÷430	1.5	÷123	7.5	÷200	5	÷37	36	÷40	31	÷6	21	÷70	18	÷10
3_2	25m	188	22	÷70	4.5	÷42	24	÷63	14	÷13	11	÷135	3	÷63	10	÷150	5.5	÷34	0.3		0.5		23	÷65	12	÷16
4_1	25m	186	7	÷215	2.5	÷75	15	÷100	8.5	÷22	4	÷375	2	÷93	7	÷215	4	÷46	3.5		11		27	÷55	26	÷7
4_2	25m	192	8	÷190	3	÷64	16	÷95	9.5	÷20	4	÷375	2	÷96	8	÷190	5	÷38	4		10		28	÷55	22	÷9
5	25m	195	21	÷70	3.5	÷55	25	÷60	12.5	÷16	22	÷70	4	÷49	27	÷55	13	÷15	15	÷100	14	÷14	19	÷80	16	÷12
6	29m	231	7	÷250	2.5	÷92	13	÷135	9	÷26	4	÷435	2	÷115	7	÷250	4	÷57	3		11		3		11	
7	25m	184	1'08	÷22	13	÷14	1'13	÷21	16	÷11	4	÷375	2.5	÷74	4	÷375	2.5	÷74	16m	-33%	388	×2	43	÷35	47	÷4

Tableau 6.17: FWS : Synthèse des expériences de vérification formelle

6.6 TI

6.6.1 Description

Ce modèle correspond à une nouvelle architecture de bus de données de Texas Instruments. Ce bus a une topologie arborescente, les nœuds étant des commutateurs (*switches*) chargés d'orienter les données vers leur destination.

Ce bus ainsi que les différents commutateurs qui le composent sont destinés à être répartis dans un volume important, tel celui d'une voiture. Cet objectif impose des contraintes quant au nombre de câbles entre les commutateurs, qui doivent être en nombre réduit.

Les transferts de données se font sous la forme de *transactions* découpées en plusieurs étapes. La contrainte relative à la quantité de câble impose à chaque commutateur de scruter la transaction en cours, même si cette transaction ne le concerne pas, afin de conserver la synchronisation de l'ensemble des commutateurs : il n'était pas possible de rajouter des fils porteurs de signaux de synchronisation. On peut donc voir les commutateurs comme des automates cycliques, chacun des états correspondant à une étape de la transaction. Ces automates cycliques "tournent" en parallèle afin de demeurer synchronisés.

La version du bus que nous avons étudiée comporte 2 commutateurs à 4 voies de données. Elle est composée de 25 modules différents et 65 instances de modules pour près de 8 000 lignes de code `Esterel` générées à partir de diagrammes `SyncCharts`. Ce modèle comporte plus de 800 registres. Les transactions étant très longues —elles peuvent de plus être préemptées par des transactions de plus forte priorité—, ce modèle est très profond : son diamètre est égal à 181. Ce modèle a 652 948 états.

6.6.2 Vérification formelle

Pour la vérification formelle, ce modèle est englobé dans un module maître qui fournit des fonctionnalités de contraintes séquentielles d'environnement. Le modèle résultant a 7 signaux d'entrées et 15 signaux d'observateurs enfouis profondément dans le modèle. Ce modèle n'est donc pas vérifiable par une approche en boîte noire.

Les 15 propriétés ont toutes les mêmes supports, et il n'apporte rien de décomposer la vérification en différentes sessions pour chaque propriété. L'intrication des supports est *a priori* indépendante de la construction en boîte blanche ; l'extraction des observateurs et leur utilisation en boîte noire aurait les mêmes conséquences.

Les différents composants de ce modèle étant très fortement interdépendants, le concepteur de cette implémentation, Cédric Hyppolite, n'a pas jugé possible l'application de nos techniques d'abstraction de variables.

Approche implicite

Les commutateurs étant assimilable à des automates cycliques "tournant" de manière synchronisée, ce modèle comporte un nombre très important de registres redondants —

une petite moitié environ puisqu'il y a deux commutateurs. L'importante proportion de registres redondants fait échouer les techniques implicites à base de BDDs.

En effet, avec des BDDs, les calculs d'images intervenant lors de chaque étape du calcul de l'espace d'états atteignables se font en ne conservant qu'un seul représentant de chacune des classes d'équivalences des fonctions de registres (cf. 2.9.4, p. 42). Les calculs d'images n'impliquent donc pas la moindre fonction redondante mais, à la fin de chaque étape, les états images doivent être confrontés à l'ensemble des états découverts jusqu'alors afin de déterminer l'ensemble des nouveaux états. Cette étape nécessite de comparer des CBFs munis de listes d'équivalences de variables dont le contenu peut être notablement différent. La comparaison des CBFs implique alors de normaliser les deux CBFs afin qu'elles aient les mêmes listes d'équivalence de variables. Cette normalisation implique des substitutions de variables, processus exponentiel dans le pire des cas.

Avec des techniques implicites, seuls les 8 premiers niveaux de profondeur du modèle (sur 181) peuvent être vérifiés, en 10mn et en consommant 270 Mo de mémoire. L'analyse du 9ème niveau échoue après 7mn en ayant épuisé les 2 Go de mémoire disponibles. Les propriétés ne sont donc validées que pour à peine plus d'1% des états atteignables.

La Figure 6.2 présente, pour chaque niveau de profondeur du modèle, le temps passé et le nombre de nœuds dans les BDDs représentant l'ensemble des nouveaux états et celui de l'espace d'états atteignables découvert jusqu'alors.

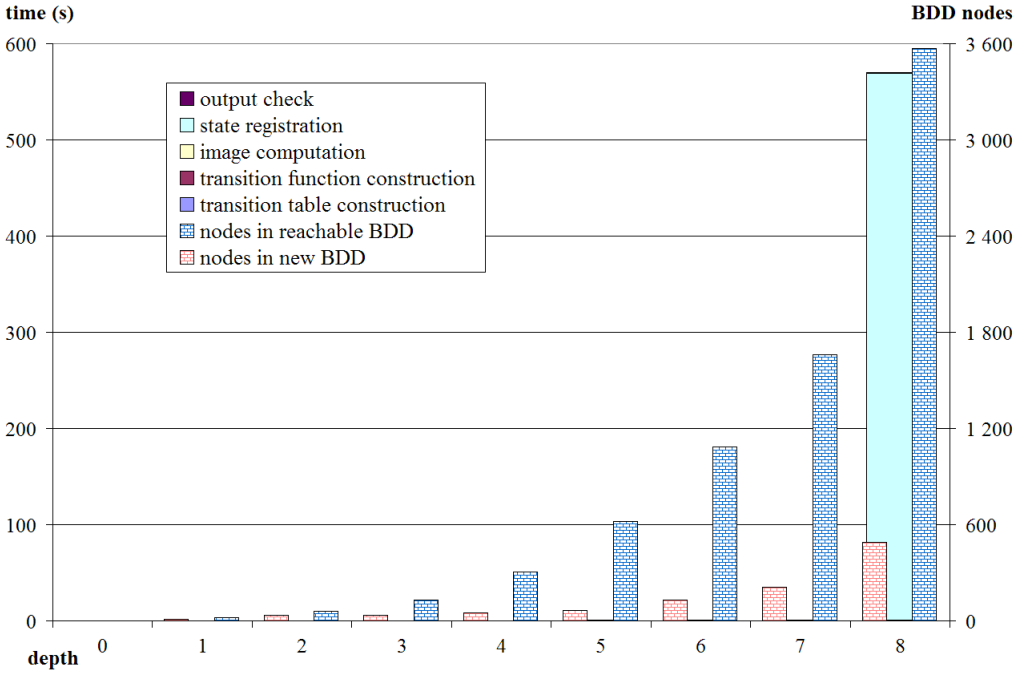


Figure 6.2: `TI` : temps et nœuds de BDDs par profondeur

On peut constater que les 7 premières profondeurs sont analysées en temps négligeable et que pour la 8ème profondeur, dont la durée d'analyse est proche des 10mn, seul le temps

passé à discriminer les nouveaux états des états déjà connus ressort. On voit aussi se dessiner une tendance exponentielle pour le nombre de nœuds de l'espace d'états atteignables.

La Figure 6.3 présente, pour chaque niveau de profondeur du modèle, la distribution du temps passé dans chacune des étapes du calcul d'espace d'états atteignables.

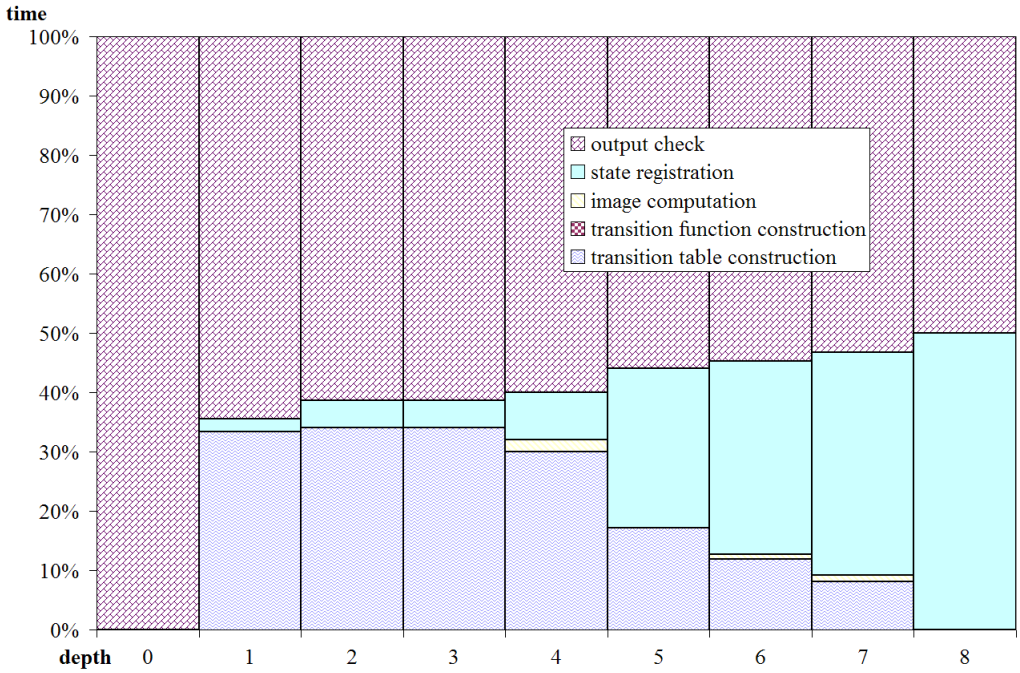


Figure 6.3: `TI` : distribution du temps par profondeur

L'analyse des premières profondeurs étant très rapide, les valeurs ne sont pas vraiment significative, mais une tendance nette se profile : à partir d'un temps d'analyse essentiellement partagé entre la confrontation des propriétés aux nouveaux états et la construction des 800 fonctions de transitions, la proportion du temps passer à discriminer les nouveaux états des états déjà connus devient prépondérante.

Utilisation de l'analyseur de satisfiabilité `Prover`

L'importante longueur des transactions confère à ce modèle un diamètre très important (181), qui fait échouer les technologies à base d'analyse de satisfiabilité (*SAT Solvers*). L'analyseur de satisfiabilité intégré à `Esterel Studio`, `Prover`, ne parvient pas à valider la moindre propriété. La consommation mémoire est toutefois très restreinte et son évolution très régulière, comme le montre la Figure 6.4 pour les 6 premières heures de calcul.

Nous avons laissé l'analyseur poursuivre ses calculs durant de nombreuses heures sans qu'aucun nouveau résultat n'émerge. L'outil ne donne ni signe de vie, ni d'indication concernant le nombre de niveaux de profondeurs vérifiés.

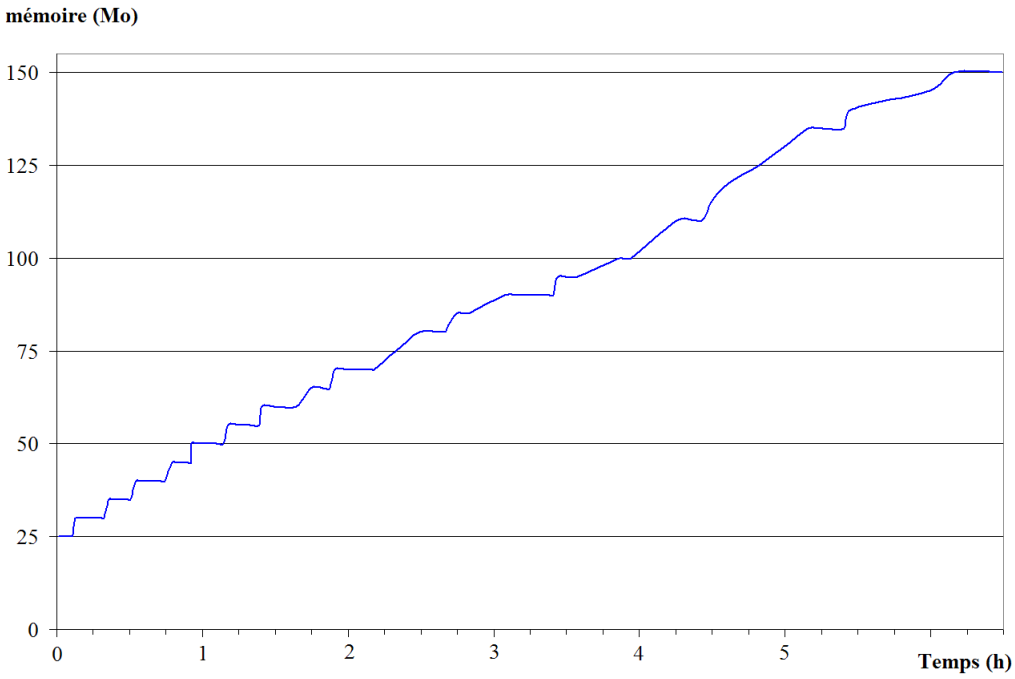


Figure 6.4: `TI` : vérification formelle par analyse de satisfiabilité

Approche purement explicite et hybride implicite/explicite

En fait, le comportement de ce modèle est fortement linéaire : dans la majorité des états, très peu de branchements sont nécessaires pour passer d'un état à un autre. De plus, à la fois le nombre d'états atteignables (652 948) et le nombre d'octets nécessaires pour représenter un état (une centaine) sont réduits. Ce modèle est donc tout à fait analysable par des techniques explicites.

Le tableau 6.18 synthétise les résultats de nos différentes tentatives de vérification formelle du modèle par des techniques implicites et explicites. Là où les techniques purement implicites échouent en ne validant qu'un peu plus d'1% des états atteignables, en 17mn et en consommant 2 Go de mémoire, les techniques purement explicites valident l'intégralité du modèle en 2h 33 et 104 Mo. Sur ce modèle très linéaire, la propagation de BDDs pour stabiliser le circuit génère un surcoût en temps de l'ordre de 25% : la vérification à l'aide de techniques hybrides implicites/explicites se fait alors en 3h 09 et 110 Mo.

Les Figures 6.5 à 6.8 mettent en valeur la linéarité, à la fois en temps et en mémoire, du comportement des techniques explicites sur ce modèle.

6.6.3 Génération d'automate

La génération d'un automate explicite à partir de ce modèle (cf. Tableau 6.19) ne nécessite que 44% de temps supplémentaire (2h 33 → 3h 40), mais avec plus de deux fois plus de mémoire (104 Mo → 239 Mo). L'automate résultant, pour 652 948 états, a 1 660 917

technique	diamètre couvert		états explorés		temps	mémoire
implicite	8	4%	94	1%	10mn	270Mo
					17mn	
explicite	181	100%	652 948	100%	2h 33	104Mo
hybride					3h 09	110Mo

Tableau 6.18: TI : Synthèse des expériences de vérification formelle

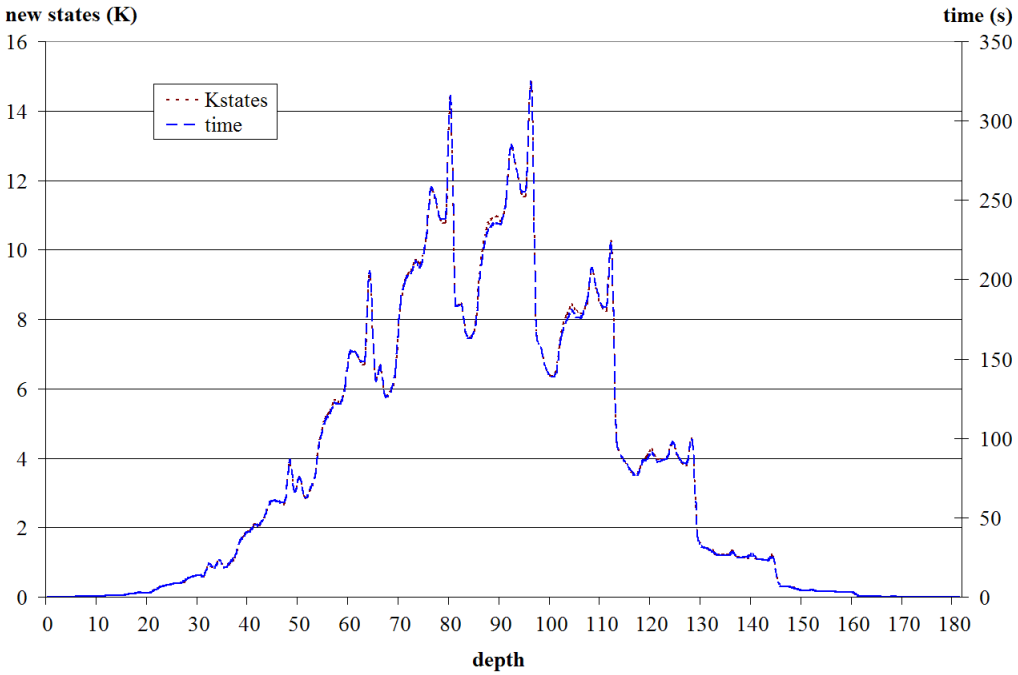


Figure 6.5: TI : nouveaux états découverts et temps d'analyse par profondeur

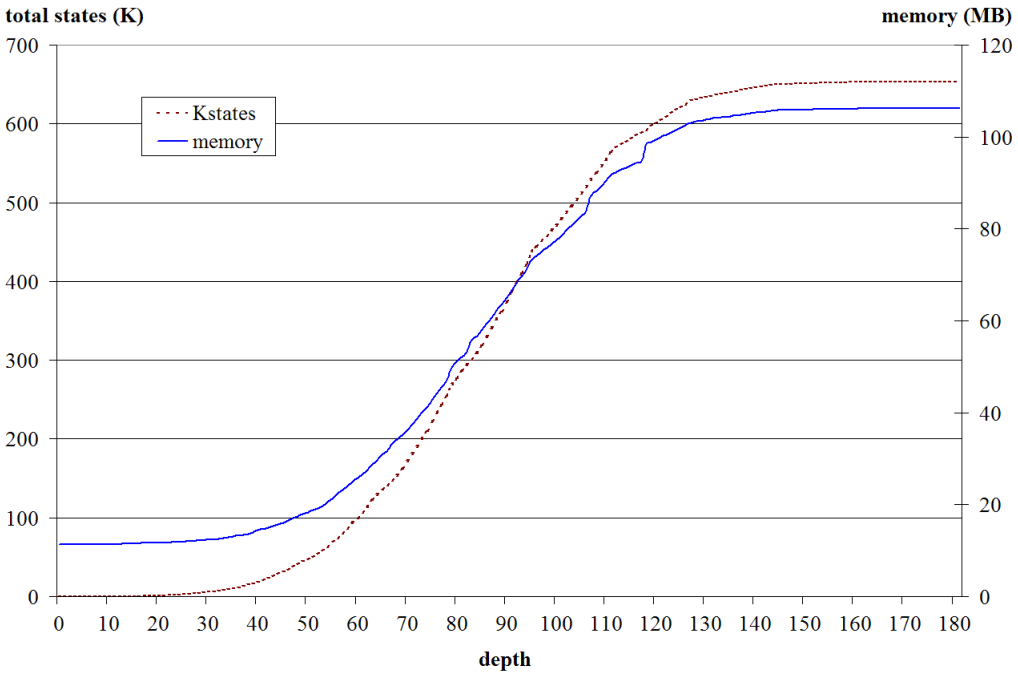


Figure 6.6: `TI` : progression des états totaux et du temps d'analyse avec la profondeur

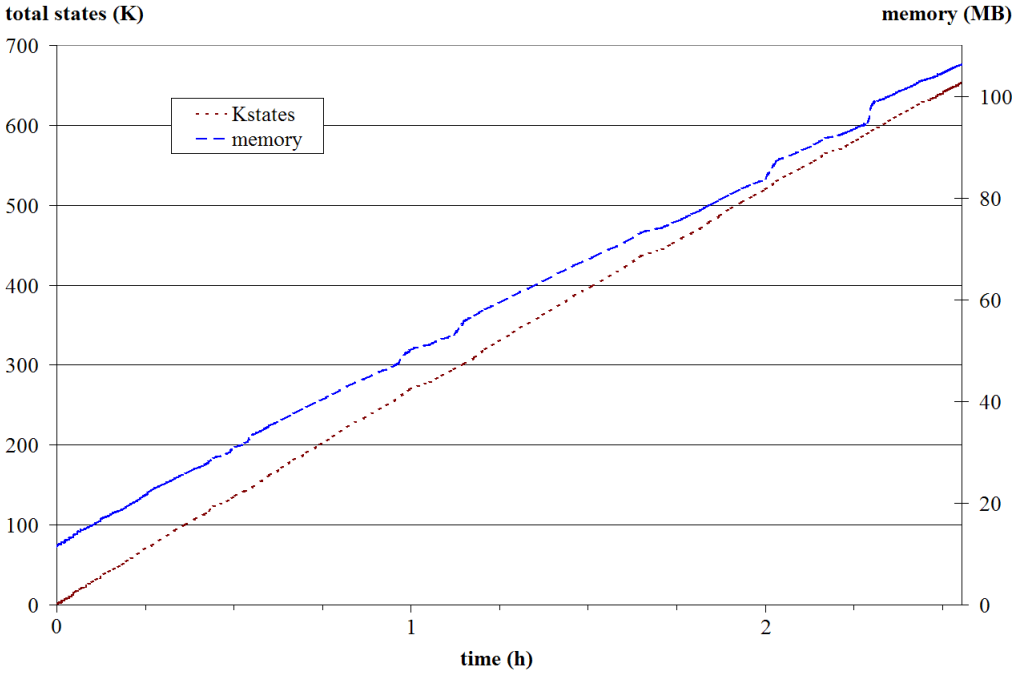


Figure 6.7: `TI` : progression des états analysés et de la consommation mémoire avec le temps

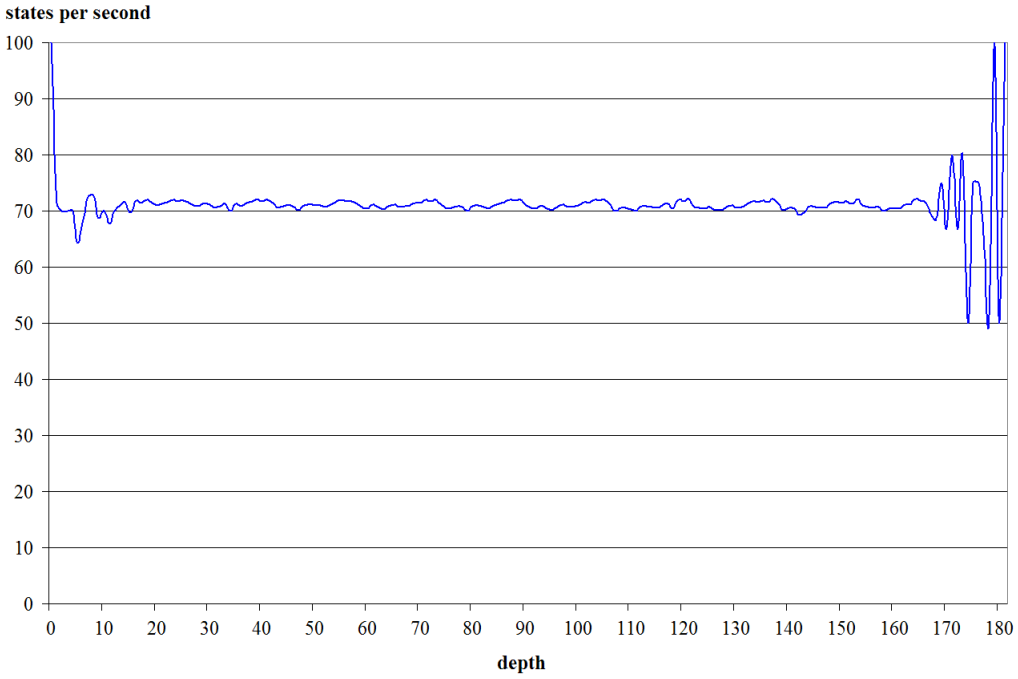


Figure 6.8: TI : débit d'analyse des états par profondeur

nœuds uniques. L'approche hybride implicite/explicite de la génération d'automates explicites est plus lente de 18% (3h 25 → 4h 03) pour une consommation mémoire supérieure de 2% (144 Mo → 147 Mo).

approche	temps	mémoire	états	nœuds
explicite	3h 25	144 Mo	652 949	1 007 969
implicite	4h 03	147 Mo		1 076 393

Tableau 6.19: TI : Génération d'automate

6.6.4 Arbre à 10 commutateurs

Nous avons appliqué notre outil de vérification formelle purement explicite sur une version beaucoup plus complexe de cette application : l'arbre du bus contenait alors 10 commutateurs pour 28 modules `SyncCharts`, 244 instances de modules, plus de 27 000 nets et près de 1 300 registres.

Sur une machine dotée de 2 Go de mémoire vive, notre outil parvient à analyser ce modèle jusqu'à la 69ème profondeur, en vérifiant plus de 15 000 000 d'états, en moins de 3 jours.

6.6.5 Conclusions

Parce qu'il a de nombreux registres redondants, ce modèle n'est pas vérifiable à l'aide de techniques purement implicites.

Parce qu'il est très profond, ce modèle ne peut être complètement vérifié à l'aide d'analyseurs de satisfiabilité.

Par contre, parce qu'il a un comportement très linéaire et qu'il comporte relativement peu d'états, ce modèle s'analyse parfaitement à l'aide de techniques purement explicites ou hybrides implicites/explicites.

6.7 Testbench

Ce modèle, plutôt anecdotique, stygmatise à l'extrême les différences de comportements entre les techniques purement implicites et les techniques purement explicites ou hybrides implicites/explicites lorsque le nombre de registres redondants est très important et le comportement très linéaire.

Cette application modélise un stimulateur parfaitement linéaire pour un modèle industriel. Quoique son implémentation soit assez complexe, elle se ramène à une séquence linéaire d'émissions de signaux. Tous les registres sont mutuellement exclusifs et, à chaque niveau de profondeur, tous les registres sauf un sont à 0. Cette application contient de nombreux compteurs, et il est indispensable d'expanser ces compteurs (cf. 4.1.4, p. 79), quelle que soit l'approche de vérification formelle adoptée.

Du fait des nombreux registres redondants, les techniques implicites nécessitent 39mn pour vérifier ce modèle absolument trivial. Du fait de son importante profondeur (243), les techniques à base d'analyseur de satisfiabilité ne donnent aucune information, même après 3 heures de calculs. Par contre, les techniques purement implicites ou hybrides implicites/explicites vérifient ce modèle en respectivement 1.6s et 1.8s.

approche	temps	mémoire
implicite	39mn	8.5Mo
`Prover` (aucun résultat)	>3h	<40Mo
explicite	1.6s	≈0 Ko
hybride	1.8s	≈0 Ko

Tableau 6.20: `Testbench` : Vérification formelle

Chapitre 7

Conclusion

Nous avons traité dans cette thèse des approches implicites et explicites de l'exploration d'espaces d'états atteignables de systèmes réactifs synchrones exprimés sous la forme de circuits logiques. Nous utilisons les résultats de ces explorations à des fins de vérification formelle ou de génération d'automates explicites.

Dans le cadre de l'approche implicite, nous avons développé un nouvel outil de vérification formelle qui intègre plusieurs stratégies visant à réduire le nombre de variables impliquées dans les calculs d'espaces d'états atteignables. La plus simple est la technique usuelle de remplacement de certaines variables d'états par des entrées libres. Nous proposons d'étendre cette technique par l'abstraction de variables à l'aide d'une logique trivaluée. Nous proposons également de gérer l'édition de lien du modèle à vérifier et des observateurs contenant les propriétés au sein même du vérifieur afin de s'affranchir des liaisons excessives générées par le compilateur `Esterel` v5 et ainsi obtenir des supports de fonctions plus réduits.

Dans le cadre de l'approche explicite, nous avons développé un nouveau moteur d'exploration d'espaces d'états atteignables. Ce moteur repose sur une simulation de la propagation des informations dans le circuit, lequel est stabilisé par branchements successifs sur les entrées. Ce moteur intègre de nombreuses stratégies exactes ou heuristiques visant à réduire le risque d'explosion en temps. Ce moteur a initialement été utilisé à des fins de génération d'automates. Le générateur d'automates obtenu se trouve être très performant et fait partie du compilateur `Esterel` commercialisé au sein de l'environnement de développement intégré `Esterel Studio`. Ce moteur a par la suite été utilisé à des fins de vérification formelle, ce qui a rendu possible ou notablement accéléré la vérification de modèles au comportement assez linéaire et comportant de nombreux registres redondants.

Enfin, dans le cadre de l'approche hybride implicite/explicite, nous avons simplifié notre moteur d'exploration d'espaces d'états purement explicite en propageant à travers le circuit des BDDs plutôt que des valeurs booléennes issues de branchements récursifs. L'utilisation de BDDs permet de réduire en pratique les risques d'explosion en temps que présente le moteur purement explicite. Les variables de BDDs manipulées étant limitées aux signaux d'entrées des modèles, nous évitons également les risques dûs à des BDDs référençant un nombre trop important de variables. Ce moteur n'est guère applicable en pratique à

la génération d'automates explicites, les contraintes dues à l'ordonnancement des actions étant trop pénalisantes. Ce moteur est plus approprié à la vérification formelle ou à la génération de séquences de tests exhaustives. Sur ce dernier point, nous avons vu qu'un prototype d'outil basé sur ce moteur hybride implicite/explicite s'avère déjà beaucoup plus performant qu'un outil similaire et plus mature basé sur une approche purement implicite.

Historiquement, les approches explicites ont été éclipsées par les méthodes implicites au début des années 90. En effet, les méthodes explicites présentent un risque intrinsèque d'explosion en temps et une consommation mémoire linéaire avec le nombre d'états atteignables. Les méthodes implicites présentent certes un risque similaire mais, en pratique, ce risque est beaucoup plus modéré. De ce fait, le domaine d'application des méthodes implicites est beaucoup plus large que celui des méthodes explicites.

Pour autant, les méthodes explicites ne sont pas à délaisser. Les méthodes implicites ne permettent par exemple pas la génération efficace d'automates pour lesquels les techniques explicites sont incontournable. Dans ce cadre, le risque d'explosion en temps n'est d'ailleurs pas le facteur rédhibitoire puisqu'il est alors également synonyme d'explosion en taille de l'automate produit : si un automate met trop de temps à être généré il ne pourra probablement pas être stockable ni donc exploitable.

De même, les méthodes explicites ou hybrides implicite/explicite facilitent aussi la génération de séquences de tests lorsque l'objectif de couverture requiert une connaissance précise du graphe de transitions du système. En effet, la représentation implicite du graphe détaillé des transitions du système nécessite de manipuler des BDDs avec un nombre important de variables. De plus, toutes les manipulations de base du graphe implicite de transitions ont des coûts non négligeables.

Enfin, comme nous l'avons vu, les techniques explicites ou hybrides implicites/explicites peuvent s'avérer plus performantes que les techniques implicites sur les modèles au comportement assez linéaire et comportant beaucoup de registres redondants, ou que les outils d'analyse de satisfiabilité sur les modèles profonds.

Nos travaux peuvent être étendus dans diverses directions.

Dans le cadre de l'approche implicite, notre technique d'abstraction de variables à l'aide d'une logique trivaluée doit encore être automatisée. Nous avons présenté des pistes de recherches pour cette automatisation, mais elles ne peuvent pas encore être appliquées aux circuits générés par le compilateur `Esterel` en version 5 et devront être retentées sur les circuits produits par la prochaine version du compilateur. De plus, il pourrait être intéressant d'appliquer notre technique d'abstraction de variables à des travaux reposant sur le remplacement de variables d'états par des entrées libres, comme les techniques de calculs d'espaces d'états basées sur le partitionnement du modèle à explorer.

Dans le cadre des approches purement explicites ou hybrides implicites/explicites, les contraintes liées à la génération d'automates étant extrêmement fortes, le champ ne semble guère libre pour de notables améliorations. Par contre, pour ce qui est de la vérification formelle ou de la génération de séquences de tests exhaustives, les degrés de liberté étant plus

nombreux, plusieurs axes de recherches sont envisageables. De nombreuses idées concernant par exemple la réduction de la taille de la table des états atteignables ont été proposées par les équipes de `SPIN` ou `Mur`φ. Néanmoins, l'approche purement explicite imposant un algorithme d'analyse d'état en pratique plus coûteux que celui mis en œuvre dans l'approche hybride implicite/explicite, nous pensons que les travaux futurs doivent être menés dans ce second cadre. Dans ce cadre, le moteur hybride d'exploration d'espaces d'états atteignables que nous avons développé s'avère déjà très performant, bien que son développement soit assez récent. Ce moteur pourra donc très probablement être encore amélioré.

Annexe A

Outils connexes

A.1 Autour de Lustre : Lesar, Lutess et Lurette

Plusieurs outils de vérification formelle ou de génération de tests sont disponibles pour le langage Lustre [HCRP91] :

- Lesar [RHR91, HLR92, HR99] est un vérificateur formel proposant à la fois une approche explicite basée sur l'énumération des états atteignables et une approche implicite à base de BDDs. Lesar est basé sur le paradigme des observateurs synchrones [HLR93] et permet la vérification en boîte noire des programmes. Initialement, Lesar était dédié à la vérification de la partie contrôle des modèles et seules les variables booléennes étaient reconnues par le vérifieur. Par la suite, Lesar a été étendu à l'aide de techniques d'interprétation abstraite [CC77, CC92] pour permettre la vérification de propriétés numériques linéaires [HPR97] ou la vérification de réseaux paramétrés de processus synchrones [LHR97].
- Lutess [dBOR+98, dBORZ99, dBZ99] et Lurette [RWNH98, HR99] sont des générateurs de tests basés sur une utilisation en boîte noire du modèle. Le modèle est composé avec des observateurs —plus précisemment des oracles— qui acceptent ou rejettent les séquences de tests soumises au modèle. Ces outils ne visent donc pas la génération de séquences de tests exhaustives. Alors que Lutess se limite à des oracles purement booléens, Lurette permet l'expression de contraintes numériques linéaires pour filtrer les séquences de tests générées.

A.2 SPIN

SPIN [Hol97a, Hol91] est un vérificateur formel basé sur l'énumération d'espace d'états atteignables de systèmes composés de processus communiquant de manière asynchrone. SPIN propose de décrire ces systèmes avec son propre langage, Promela (*A Process Meta Language*). Les propriétés à vérifier sont exprimées en Logique Temporaire Linéaire (LTL) et transformées à la volée [GPVW95] en automates de Büchi.

Les états connus peuvent être stockés de manière usuelle dans une table d'adressage

dispersée. Dans ce cadre, des techniques de compression des vecteurs de bits décrivant les états ont été analysées (*Bit-State Hashing*) : [Hol88, Hol91, Hol95, Hol97b].

Ces techniques reposent notamment sur un hachage des vecteurs de bits dans une table d'adressage dispersé sans détection de collision. L'idée est de considérer le code calculé par la fonction de hachage des vecteurs d'états comme un index dans un tableau de bit de très grande taille : si le bit est positionné, alors l'état est connu, sinon il est nouveau. Cette approche permet de ne stocker *strictement aucun* vecteur d'état et de n'utiliser qu'un bit par état connu. Ainsi, la table d'états connus d'une fonction de hachage renvoyant un résultat sur 32 bits n'occupe "que" 512Mo de mémoire et permet de référencer plus de 4 milliards d'états[1]. Les modèles à analyser ayant généralement beaucoup plus d'une trentaine de registres, il convient de prendre en compte les probabilités de collisions, d'autant plus que lorsqu'un nouvel état entre en collision avec un état déjà connu, il est non seulement omis, mais avec lui tous les états dont il est l'unique précurseur, etc.

Les collisions peuvent être réduites en utilisant plus d'une fonction de hachage : un état est alors considéré comme déjà analysé si tous les bits dont les index sont calculés par les différentes fonctions de hachage sont positionnés. Dans [Hol95], pour un double hachage des vecteurs d'états, Holzmann considère que chaque collision ne conduit à l'omission que d'un seul état, pour peu que le graphe d'états du modèle analysé soit suffisamment connecté. la plupart des modèles sont soutient que, pour deux fonctions de hachage, si la table des états est remplie à moins de 1%, alors le modèle est exploré à plus de 99%. Outre que cette analyse a été réfutée dans [SD96], une garantie de moins de 1% d'états omis est loin d'être suffisante pour valider un modèle.

Alternativement, les états connus peuvent être représentés un BDD [Vis96], une structure d'arbres partagés [Gré96] ou par un automate d'états finis déterministe [HP99]. Dans la plupart des cas, ces techniques augmentent sensiblement les coûts en temps des accès et des mises à jour de l'ensemble des états connus, mais elles permettent de réduire la consommation mémoire.

A.3 Murφ

Murφ [Dil96] est un vérifieur formel utilisant à la fois des techniques explicites et implicites, développé à l'Université de Stanford par l'équipe de David L. Dill. Murφ propose son propre langage de description de modèles, le langage Murφ et son évolution Murφ++, assez proche de langages de descriptions de hardware, notamment Verilog. Initialement destiné à la spécification de protocoles, le langage Murφ permet plus généralement la description de machines d'états finis non déterministes, ainsi que leur composition asynchrone. La communication entre processus concurrents se fait uniquement par variables partagées ; il n'y a pas d'autre construction de communication. Les modèles décrits sont paramétrables par des constantes symboliques. Enfin, le langage Murφ supporte la déclaration d'invariants sous la forme de propriétés booléennes, et a eu supporté la spécification de propriétés de

[1] $512 * 1024 * 1024 * 8 = 2^{32} = 4\ 294\ 967\ 296$.

vivacité.

En Murφ, comme avec SPIN, la sémantique de la concurrence correspond à l'exécution en parallèle de processus évoluant à des vitesses arbitraires. L'équipe de Murφ a donc aussi travaillé sur :

- la prise en compte des symétries (*symmetry reduction*), pour les modèles dont les propriétés à vérifier sont indifférentes à certaines permutations de processus [ID93a, ID93b] ;
- la réduction de sous-graphes de l'espace d'états atteignables par des règles réversibles (*reversible rules*) en états générateurs de ces sous-graphes [ID96a] ;
- les constructeurs de répétitions [ID96b].

Similairement à notre outil basé sur une approche hybride implicite/explicite, Murφ analyse les états atteignables individuellement, en les stockant sous la forme d'un vecteur de bits dans une table d'états connus. Murφ travaillant sur des programmes *valués*, à chaque variable du programme est associée une liste de couples $(valeur, condition)$, où *condition* est un BDD représentant le domaine pour lequel la variable prend la valeur *valeur*. Les blocs d'instructions dépendant de conditions sont simplement évalués avec un contexte –le BDD *condition*– différent, selon que l'on soit dans une branche *then* ou *else*.

Outre ses fonctionnalités de vérification formelle, Murφ peut générer le graphe d'états explicite de machines d'états finis. Ce graphe peut par la suite être utilisé pour générer des tests couvrants, ou indiquer le taux de couverture de vecteurs de tests spécifiés.

L'exploration des modèles en largeur (*breadth-first*) étant très peu appropriée à la recherche d'erreurs (*bug tracking*), Murφ propose un mode d'analyse *ciblé*. Dans ce mode, les états ne sont plus analysés en fonction de l'ordre dans lequel ils ont été découverts, mais dans un ordre déterminé par une fonction de pondération[2]. Cette fonction est une heuristique qui favorise les états ayant une forte propension à atteindre rapidement un état d'erreur, dans lequel une ou des propriétés sont violables.

Une heuristique de base consiste à utiliser la distance de Hamming d'un état à un autre, soit le nombre de bits qui diffèrent entre les deux vecteurs de bits représentant ces états [YSAA97, Yan98]. Le calcul de la distance de Hamming entre un état et un ensemble d'états se généralise en calculant la distance minimale de chacune des distances.

L'utilisation d'heuristiques guidant la recherche vers les états d'erreurs est renforcée dans Murφ par l'élargissement des cibles (*Target Enlargement*), soit des calculs successifs d'images inverses à partir des états d'erreur [Yan98]. Cette technique explore encore une fois le compromis explicite/implicite : des calculs d'images inverses symboliques sont effectués jusqu'à une certaine profondeur, afin de ne pas trop consommer de mémoire, puis les techniques énumératives entrent en jeu.

Le système Murφ est aussi disponible en version distribuée, Parallel Murφ [SD97, Ste97]. Cette version est destinée aux machines multi-processeurs à mémoire distribuée, ou, plus généralement, aux réseaux d'ordinateurs (*computer farms*). Parallel Murφ distribue non seulement les analyses d'états aux différents serveurs de calculs, mais aussi la

[2]En fait, en mode *breadth-first*, cette fonction est une constante.

maintenance de la table des états connus. En terme d'accélération des calculs, cette approche permet en pratique des gains à peu près linéaires : doubler le nombre de machines permet de diviser par deux les temps de calculs. De plus, la répartition de la table des états connus entre les différents serveurs de calculs permet aussi d'accroître très sensiblement le nombre d'états analysables.

Une approche consistant à stocker la table des états connus dans un fichier plutôt qu'en mémoire a été étudiée dans [Ste97, SD98]. Cette approche consiste à ne distinguer les nouveaux états des états déjà connus qu'à la fin de chaque analyse de profondeur (voire plus fréquemment si nécessaire), par lecture linéaire du fichier contenant la table des états connus, le fichier étant augmenté par la suite. Cette approche évite donc l'écueil des accès aléatoires à un fichier, généralement très coûteux. Selon les expériences, le ralentissement de l'analyse du modèle dû aux accès disque demeure inférieur à 30%, pour une réduction de la consommation mémoire allant jusqu'à un facteur 50. Cette approche semble difficilement applicable aux analyses en profondeur des modèles.

Enfin, des méthodes de compression de la table des états connus (*Hash Compaction*) on été proposées en extension des travaux menés sur `SPIN` [SD95b, SD96].

Annexe B

Sémantique opérationnelle du langage **Esterel**

Cette annexe reprend les règles de la sémantique opérationnelle du langage `Esterel`, détaillée dans [Ber99].

$$k \xrightarrow[E]{\emptyset, k} 0 \qquad (compl)$$

$$!s \xrightarrow[E]{\{s^+\}, 0} 0 \qquad (emit)$$

$$\frac{s^+ \in E \quad p \xrightarrow[E]{E', k} p'}{s\,?\,p\,,q \xrightarrow[E]{E', k} p'} \qquad (present+)$$

$$\frac{s^- \in E \quad q \xrightarrow[E]{F', l} q'}{s\,?\,p\,,q \xrightarrow[E]{F', l} q'} \qquad (present-)$$

$$\frac{p \xrightarrow[E]{E', k} p' \quad k \neq 0}{s \supset p \xrightarrow[E]{E', k} s \supset p'} \qquad (susp1)$$

$$\frac{p \xrightarrow[E]{E', 0} p'}{s \supset p \xrightarrow[E]{E', 0} 0} \qquad (susp2)$$

$$\frac{p \xrightarrow[E]{E',k} p' \quad k \neq 0}{p\,;\,q \xrightarrow[E]{E',k} p'\,;\,q} \qquad (seq1)$$

$$\frac{p \xrightarrow[E]{E',0} p' \quad q \xrightarrow[E]{F',l} q'}{p\,;\,q \xrightarrow[E]{E' \cup F',l} q'} \qquad (seq2)$$

$$\frac{p \xrightarrow[E]{E',k} p' \quad k \neq 0}{p* \xrightarrow[E]{E',k} p'\,;\,p*} \qquad (loop)$$

$$\frac{p \xrightarrow[E]{E',k} p' \quad q \xrightarrow[E]{F',l} q'}{p \mid q \xrightarrow[E]{E' \cup F',max(k,l)} p' \mid q'} \qquad (parallel)$$

$$\frac{p \xrightarrow[E]{E',k} p' \quad k = 0 \text{ or } k = 2}{\{p\} \xrightarrow[E]{E',0} 0} \qquad (trap1)$$

$$\frac{p \xrightarrow[E]{E',k} p' \quad k = 1 \text{ or } k > 2}{\{p\} \xrightarrow[E]{E',\downarrow k} \{p'\}} \qquad (trap2)$$

$$\frac{p \xrightarrow[E]{E',k} p'}{\uparrow p \xrightarrow[E]{E',\uparrow k} \uparrow p'} \qquad (shift)$$

$$\frac{p \xrightarrow[E*s^+]{E'*s^+,k} p' \quad \mathcal{S}(E') = \mathcal{S}(E) \backslash s}{p \backslash s \xrightarrow[E]{E',k} p' \backslash s} \qquad (sig+)$$

$$\frac{p \xrightarrow[E*s^-]{E'*s^-,k} p' \quad \mathcal{S}(E') = \mathcal{S}(E) \backslash s}{p \backslash s \xrightarrow[E]{E',k} p' \backslash s} \qquad (sig-)$$

Liste des algorithmes

2.1 Calcul du point fixe de la suite RSS 40
4.1 Algorithme de base du moteur d'évaluation explicite de circuits 77
5.1 Algorithme de base du moteur d'évaluation hybride de circuits 104

Liste des figures

2.1 L'automate AB_O . . . 23
2.2 Le circuit AB_O . . . 24
2.3 Bloc de base des circuits Esterel . . . 26
2.4 Le circuit "Hamlet" . . . 27
2.5 La chaîne de compilation d'Esterel v5_21 . . . 30
2.6 La chaîne de compilation d'Esterel v5_9x . . . 33
2.7 Diagrammes de décisions binaires de la formule $(x_1 \Leftrightarrow y_1) \wedge (x_2 \Leftrightarrow y_2)$ . . . 37
2.8 Structure en oignon de l'espace d'états atteignables . . . 39
2.9 Exemple de circuit séquentiel . . . 43
2.10 Observateur exécuté en parallèle du modèle à observer . . . 44

3.1 L'interface graphique Xeve . . . 48
3.2 Cône d'influence des fonctions d'un circuit . . . 50
3.3 Fragment de circuit généré par l'expression **present** I **then** ... **else** ... 52
3.4 Ensembles f^0, f^1 et $f^d = \overline{f^0 + f^1}$ . . . 54
3.5 Elargissement de la fonction trivaluée f . . . 55
3.6 Circuit généré pour les constructions parallèles (3 composants) . . . 59
3.7 BDD de l'arbre de sélection du Programme 2.6 . . . 62
3.8 Flots de création de Tgr_Networks . . . 64
3.9 Têtes de nœuds de BDDs . . . 65

4.1 Graphe de relations entre entrées . . . 79
4.2 Le circuit du Programme 4.1 sans expansion des compteurs . . . 80
4.3 Arbitre de bus (à 6 cellules) . . . 82
4.4 Automate de l'arbitre de bus (à 3 cellules) . . . 83
4.5 Circuit du programme Esterel 4.2 . . . 85
4.6 Début du graphe de transition d'initialisation de variables . . . 85
4.7 Circuit du programme Esterel 4.3 (ReconvergentBranches) . . . 87
4.8 Sous-circuit correspondant à une instruction de test . . . 88
4.9 Chaînage des piles d'UndoBlocks . . . 90
4.10 Automates avec et sans partage du programme Esterel 4.4 . . . 95
4.11 Structures de tables d'adressage dispersé . . . 97
4.12 Structures des *pools* de notre gestionnaire de tas . . . 99

5.1 Analyse des vecteurs de BDDs des registres 106

6.1 Interface graphique de l'application `Carburant` 119
6.2 `TI` : temps et nœuds de BDDs par profondeur 140
6.3 `TI` : distribution du temps par profondeur 141
6.4 `TI` : vérification formelle par analyse de satisfiabilité 142
6.5 `TI` : nouveaux états découverts et temps d'analyse par profondeur 143
6.6 `TI` : progression des états totaux et du temps d'analyse avec la profondeur 144
6.7 `TI` : progression des états analysés et de la consommation mémoire avec le temps 144
6.8 `TI` : débit d'analyse des états par profondeur 145

Liste des programmes **Esterel**

2.1 Quelques instructions Esterel dérivées 21
2.2 Runner 22
2.3 AB_O 23
2.4 BadCycle 27
2.5 GoodCycle 27
2.6 SelectionTree 29
2.7 ABRO 34
2.8 AB_O avec observateur 45
4.1 Counters 79
4.2 ResetActions 85
4.3 ReconvergentBranches 86
4.4 UnrelatedParallelBlocks 95

Liste des tableaux

4.1 Apports de la pondération des entrées . . . 83
4.2 Apports de la constructivité faible . . . 84
4.3 Apports du partage des nœuds isomorphes . . . 93
4.4 Apports de la suppression des branchements inutiles . . . 96
4.5 Apports du gestionnaire de tas *ad hoc* . . . 100

5.1 Génération d'automates avec les moteurs purement explicite et hybride . . . 109
5.2 Couverture d'états purement implicite ou hybride implicite/explicite . . . 112

6.1 `Carburant` : Vérification des propriétés 1, 2, 3 et 4 . . . 121
6.2 `Carburant` : Vérification de la propriété 1 . . . 122
6.3 `Carburant` : Vérification de la propriété 2 . . . 122
6.4 `Carburant` : Vérification de la propriété 3 . . . 123
6.5 `Carburant` : Vérification de la propriété 4 . . . 124
6.6 `FWS` : Calcul de l'espace d'états du système intégral . . . 133
6.7 `FWS` : Vérification des propriétés P1 à P7 . . . 133
6.8 `FWS` : Vérification de la propriété P1 . . . 133
6.9 `FWS` : Vérification de la propriété P2 . . . 134
6.10 `FWS` : Vérification de la propriété P3_1 . . . 134
6.11 `FWS` : Vérification de la propriété P3_2 . . . 135
6.12 `FWS` : Vérification de la propriété P4_1 . . . 135
6.13 `FWS` : Vérification de la propriété P4_2 . . . 136
6.14 `FWS` : Vérification de la propriété P5 . . . 136
6.15 `FWS` : Vérification de la propriété P6 . . . 137
6.16 `FWS` : Vérification de la propriété P7 . . . 137
6.17 `FWS` : Synthèse des expériences de vérification formelle . . . 138
6.18 `TI` : Synthèse des expériences de vérification formelle . . . 143
6.19 `TI` : Génération d'automate . . . 145
6.20 `Testbench` : Vérification formelle . . . 147

Bibliography

[ABD98] Charles André, Hedi Boufaïed, and Sylvan Dissoubray.
`SyncCharts` : un modèle graphique synchrone pour systèmes réactifs complexes.
In *Real-Time and Embedded Systems, RTS'98*, pages 175–194. Teknea, January 1998.
2.6.2

[Ake78] Sheldon B. Akers.
Binary decision diagrams.
IEEE Transactions on Computers, C-27(6):509–516, June 1978.
2.8

[AM00] Pablo Argon and Ken L. McMillan.
Deriving a Special-Purpose Prover for Compositional Model-Checking in `Coq`.
In *Supplemental Proceedings of the 13th International Conference on Theorem Proving and Higher Order Logics, TPHOLS'00*, 2000.
3.8.2

[And96a] Charles André.
Representation and Analysis of Reactive Behaviors: A Synchronous Approach.
In *Proceedings of the IEEE-SMC Computational Engineering in Systems Applications Conference, CESA'96*, pages 777–782, July 1996.
Also available as I3S Technical Report RR-96-28, Sophia Antipolis.
2.6.2

[And96b] Charles André.
`SyncCharts`: A Visual Representation of Reactive Behaviors, 1996.
I3S Technical Report RR 95-52, Sophia Antipolis.
2.6.2

[ATB94] Adnan Aziz, Serdar Tasiran, and Robert K. Brayton.
BDD Variable Ordering for Interacting Finite State Machines.
In *Proceedings of the 31st Design Automation Conference, DAC'94*, pages 283–288. ACM Press, June 1994.
2.8

[BCL91] Jerry R. Burch, Edmund M. Clarke, and David E. Long.
Symbolic Model Checking with Partitioned Transition Relations.
In A. Halaas and P. B. Denyer, editors, *Proceedings of the International Conference on Very Large Scale Integration*, pages 49–58. North-Holland, 1991.
2.9.3

[BCL+94] Jerry R. Burch, Edmund M. Clarke, David E. Long, Ken L. McMillan, and David L. Dill.
Symbolic Model Checking for Sequential Circuit Verification.
IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 13(4):401–424, 1994.
2.9.3

[BdS92] Amar Bouali and Robert de Simone.
Symbolic Bisimulation Minimisation.
In *Proceedings of the 4th Workshop on Computer Aided Verification, CAV'92*, volume 663 of *Lecture Notes in Computer Science*, pages 96–108. Springer Verlag, 1992.
3.1, 4.1.4

[Ber] Gérard Berry.
The `Esterel` Language Primer.
CMA, Ecole des Mines and INRIA and Esterel Technologies.
Available with the `Esterel` system and updated for each release.
2.1, 2.1

[Ber98] Gérard Berry.
The Foundations of `Esterel`.
MIT Press, 1998.
Edited by C. Stirling, G. Plotkin and M. Tofte.
2.3.2

[Ber99] Gérard Berry.
The Constructive Semantics of Pure `Esterel`.
Draft / Not yet published, July 1999.
1, 2.3.2, 2.3.4, 2.3.5, 2.5, 3.3.4, 4, 4.1.4, 4.1.5, B

[BG92] Gérard Berry and Georges Gonthier.
The `Esterel` Synchronous Programming Language : Design, Semantics, Implementation.
Science of Computer Programming, 19(2):87–152, 1992.
2.2

[BLI98] University of California at Berkeley.
Berkeley Interchange Logic Format (BLIF), December 1998.
2.4

[BMdST96] Amar Bouali, Jean-Paul Marmorat, Robert de Simone, and Horia Toma.
Verifying Synchronous Reactive Systems Programmed in `Esterel`.
In *Proceedings of the 4th International School and Symposium on Formal Techniques in Real Time and Fault Tolerant Systems, FTRTFT'96*, volume 1135 of *Lecture Notes in Computer Science*, September 1996.
4.1.4

[BMZ02] Yves Bertot, Nicolas Magaud, and Paul Zimmermann.
A proof of GMP square root.
Journal of Automated Reasoning, June 2002.
Also available as INRIA Research Report RR-4475.
3.8.2

[BNLD00] Yann Le Biannic, Eric Nassor, Emmanuel Ledinot, and Sylvan Dissoubray.
UML Object Specification for Real-Time Software.
RTS 2000 Show, March 2000.
3.1.1, 6.4.1

[Bou97] Amar Bouali.
`Xeve`: an `Esterel` Verication Environment (Version v1_3).
Technical Report 0214, INRIA, December 1997.
2.4, 3.1

[Bou98] Amar Bouali.
`Xeve`: an `Esterel` Verification Environment.
In *Proceedings of the 10th International Conference on Computer Aided Verification, CAV'98*, volume 1427 of *Lecture Notes in Computer Science*, pages 500–504. Springer-Verlag, June 1998.
2.4, 3.1

[Bra93] Daniel Brand.
Verification of Large Synthesized Designs.
In *Proceedings of the International Conference on Computer-Aided Design, ICCAD'93*, pages 534–537. IEEE Computer Society Press, November 1993.
2.10.3

[Bry86] Randal E. Bryant.
Graph-Based Algorithms for Boolean Function Manipulation.
IEEE Transactions on Computers, C-35(8):677–691, August 1986.
2.8, 3

[Bry92] Randal E. Bryant.
Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams.
ACM Computing Surveys, 24(3):293–318, 1992.
2.8

[CBM89] Olivier Coudert, Christian Berthet, and Jean-Christophe Madre.

Verification of Synchronous Sequential Machines Based on Symbolic Execution.
In *Proceedings of the Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, June 1989.
2.9.6, 3.3.2

[CC77] Patrick Cousot and Radhia Cousot.
Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints.
In *Proceedings of the 4th Symposium on Principle of Programming Languages, POPL'77*, pages 238–252. ACM Press, January 1977.
A.1

[CC92] Patrick Cousot and Radhia Cousot.
Abstract Interpretation and Application to Logic Programs.
Journal of Logic Programming, 13(2–3):103–179, July 1992.
A.1

[CCJ+01a] Pankaj Chauhan, Edmund M. Clarke, Somesh Jha, Jim Kukula, Tom Shiple, Helmut Veith, and Dong Wang.
Non-linear Quantification Scheduling in Image Computation.
In *Proceedings of the International Conference on Computer-Aided Design, ICCAD'01*, pages 293–298, November 2001.
2.9.3

[CCJ+01b] Pankaj Chauhan, Edmund M. Clarke, Somesh Jha, Jim Kukula, Helmut Veith, and Dong Wang.
Using Combinatorial Optimization Methods for Quantification Scheduling.
In *Proceedings of the Conference on Correct Hardware Design and Verification Methods, CHARME'01*, volume 2144 of *Lecture Notes in Computer Science*, pages 293–309, September 2001.
2.9.3

[CCLQ97] Gianpiero Cabodi, Paolo Camurati, Luciano Lavagno, and Stefano Quer.
Disjunctive Partitioning and Partial Iterative Squaring: An Effective Approach for Symbolic Traversal of Large Circuits.
In *Proceedings of the 34th Design Automation Conference, DAC'97*, pages 728–733. ACM Press, June 1997.
2.9.3

[CCQ96] Gianpiero Cabodi, Paolo Camurati, and Stefano Quer.
Improved Reachability Analysis of Large Finite State Machines.
In *Proceedings of the International Conference on Computer-Aided Design, ICCAD'96*, pages 354–360. IEEE Computer Society Press, November 1996.
2.9.3

[CFG94] Paul Caspi, Jean-Claude Fernandez, and Alain Girault.
An Algorithm for Reducing Binary Branchings: Implementation and Formal Proof.
Technical report, INRIA, 1994.
4.2.3

[CFG95] Paul Caspi, Jean-Claude Fernandez, and Alain Girault.
An Algorithm for Reducing Binary Branchings.
In *Proceedings of the 15th Conference on the Foundations of Software Technology and Theoretical Computer Science, FST&TCS'95*, volume 1026 of *Lecture Notes in Computer Science*, pages 279–293. Springer Verlag, December 1995.
4.2.3

[CGH+93] Edmund M. Clarke, Orna Grumberg, Hiromi Hiraishi, Somesh Jha, David E. Long, Ken L. McMillan, and Linda A. Ness.
Verification of the Futurebus+ Cache Coherence Protocol.
In *Proceedings of the 11th International Symposium on Computer Hardware Description Languages and their Applications*, pages 5–20. Elsevier Science Publishers B.V., April 1993.
1

[CGJ+00] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith.
Counterexample-guided Abstraction Refinement.
In *Proceedings of the 11th International Conference on Computer Aided Verification, CAV'00*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169, July 2000.
3.3.4

[CHM+93] Hyunwoo Cho, Gary Hatchel, Enrico Macii, Bernard Plessier, and Fabio Somenzi.
Algorithms for Approximate FSM Traversal based on State Space Decomposition.
In *Proceedings of the 30th Design Automation Conference, DAC'93*, pages 25–30. ACM Press, June 1993.
3.3.5

[CM90] Olivier Coudert and Jean-Christophe Madre.
A Unified Framework for the Formal Verification of Sequential Circuits.
In *Proceedings of the International Conference on Computer-Aided Design, ICCAD'90*. IEEE Computer Society Press, November 1990.
2.8, 2

[CM91] Olivier Coudert and Jean-Christophe Madre.
Symbolic Computation of the Valid States of a Sequential Machine: Algorithms and Discussion.

In *Proceedings of International Worshop on Formal Methods in VLSI Design*, January 1991.
2.9.2, 2.9.6, 3.3.2, 3.3.2

[CMT93] Olivier Coudert, Jean-Christophe Madre, and Hervé Touati.
TiGeR Version 1.0 User Guide.
Digital Paris Research Lab, December 1993.
3.1, 3.5

[CYF94] Ben Chen, Michihiro Yamazaki, and Masahiro Fujita.
Bug Identification of a Real Chip Design by Symbolic Model Checking.
In *Proceedings of the European Design and Test Conference, EDAC'94*, pages 132–136. IEEE Computer Society Press, March 1994.
1

[dBOR+98] Lydie du Bousquet, Farid Ouabdesselam, Jean-Luc Richier, JeanLuc Richier, and Nicolas Zuanon.
Lutess: a Testing Environment for Synchronous Software.
Tool support for System Specification Development and Verification, pages 48–61, 1998.
A.1

[dBORZ99] Lydie du Bousquet, Farid Ouabdesselam, Jean-Luc Richier, and Nicolas Zuanon.
Lutess: A Specification-Driven Testing Environment for Synchronous Software.
In *Proceedings of the 21st International Conference on Software Engineering*, pages 267–276. ACM Press, May 1999.
A.1

[dBZ99] Lydie du Bousquet and Nicolas Zuanon.
An Overview of Lutess: A Specification-based Tool for Testing Synchronous Software.
In *Proceedings of the 14th International Conference on Automated Software Engineering*, pages 208–215. IEEE, October 1999.
A.1

[Dil96] David L. Dill.
The Murφ Verification System.
In *Proceedings of the 8th International Conference on Computer Aided Verification, CAV'96*, volume 1102 of *Lecture Notes in Computer Science*, pages 390–393. Springer Verlag, July 1996.
A.3

[GB94] Daniel Geist and Ilan Beer.
Efficient Model Checking by Automated Ordering of Transition Relation.

In *Proceedings of the 6th International Conference on Computer-Aided Verification, CAV'94*, volume 818 of *Lecture Notes in Computer Science*, pages 299–310. Springer-Verlag, June 1994.
2.9.3

[GD98] Shankar G. Govindaraju and David L. Dill.
Verification by Approximate Forward and Backward Reachability.
In *Proceedings of the International Conference on Computer-Aided Design, ICCAD'98*, pages 366–370, November 1998.
3.3.5

[GD00] Shankar G. Govindaraju and David L. Dill.
Counterexample-Guided Choice of Projections in Approximate Symbolic Model Checking.
In *Proceedings of the International Conference on Computer-Aided Design, ICCAD'00*, pages 115–119. IEEE Press, November 2000.
3.3.4, 3.3.5

[GDB00] Shankar G. Govindaraju, David L. Dill, and Jules P. Bergmann.
Improved Approximate Reachability using Auxiliary State Variables.
In *Proceedings of the 36th Design Automation Conference, DAC'99*, pages 312–316. ACM Press, June 2000.
3.3.5

[GDHH98] Shankar G. Govindaraju, David L. Dill, Alan J. Hu, and Mark A. Horrowitz.
Approximate Reachability with BDDs using Overlapping Projections.
In *Proceedings of the 35th Design Automation Conference, DAC'98*, pages 451–456. ACM Press, June 1998.
3.3.5

[GK00] Malay K. Ganai and Andreas Kuehlmann.
On-the-fly Compression of Logical Circuits.
Technical Report RC 21704, IBM Research Division, T. J. Watson Research Center, March 2000.
3.5.3

[GPVW95] Rob Gerth, Doron Peled, Moshe Y. Vardi, and Pierre Wolper.
Simple On-the-fly Automatic Verification of Linear Temporal Logic.
In *Proceedings of the 15th Workshop on Protocol Specification Testing and Verification*. Chapman & Hall, 1995.
A.2

[Gré96] Jean-Charles Grégoire.
State Space Compression in `SPIN` with GETSs.
In *Proceedings of the 2nd `SPIN` Workshop*, August 1996.
A.2

[Hal93] Nicolas Halbwachs.

Synchronous Programming of Reactive Systems.
Kluwer Academic Publishers, 1993.
2.1

[Hal98] Nicolas Halbwachs.
Synchronous Programming of Reactive Systems: A Tutorial and Commented Bibliography.
In *Proceedings of the 10th International Conference on Computer Aided Verification, CAV'98*, volume 1427 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, June 1998.
2.1

[Har87] David Harel.
`StateCharts`: A Visual Formalism for Complex Systems.
Science of Computer Programming, 8(3):231–274, June 1987.
2.6.2

[HCRP91] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud.
The Synchronous Data-Flow Programming Language `Lustre`.
Proceedings of the IEEE, 79(9):1305–1320, September 1991.
A.1

[HKB96] Ramin Hojati, Sriram C. Krishnan, and Robert K. Brayton.
Early Quantification and Partitioned Transition Relations.
In *Proceedings of the International Conference on Computer Design, ICCD'96*, pages 12–19, October 1996.
2.9.3

[HLR92] Nicolas Halbwachs, Fabienne Lagnier, and Christophe Ratel.
Programming and Verifying Real-Time Systems by Means of the Synchronous Data-Flow Language `Lustre`.
IEEE Transactions on Software Engineering, Special Issue on the Specification and Analysis of Real-Time Systems, 18(9):785–793, September 1992.
A.1

[HLR93] Nicolas Halbwachs, Fabienne Lagnier, and Pascal Raymond.
Synchronous Observers and the Verification of Reactive Systems.
In *Proceedings of the 3rd International Conference on Algebraic Methodology and Software Technology, AMAST'93*, pages 83–96. Springer Verlag, June 1993.
2.10, 3.6, A.1

[Hol88] Gerard J. Holzmann.
An Improved Protocol Reachability Analysis Technique.
Software - Practice and Experience, 18(2):137–161, February 1988.
4.5, A.2

[Hol91] Gerard J. Holzmann.
Design and Validation of Computer Protocols.
Prentice-Hall, 1991.
A.2

[Hol95] Gerard J. Holzmann.
An Analysis of Bitstate Hashing.
In *Proceedings of the 15th International Conference on Protocol Specification, Testing, and Verification*, pages 301–314, June 1995.
4.5, A.2

[Hol97a] Gerard J. Holzmann.
Memory efficient storage in `SPIN`.
IEEE Transactions on Software Engineering, 23(5), May 1997.
A.2

[Hol97b] Gerard J. Holzmann.
State Compression in `SPIN`: Recursive Indexing and Compression Training Runs.
In *Proceedings of the 3rd `SPIN` Workshop*, April 1997.
A.2

[HP99] Gerard J. Holzmann and Anuj Puri.
A Minimized Automaton Representation of Reachable States.
Software Tools for Technology Transfer, 3(1), 1999.
A.2

[HPR97] Nicolas Halbwachs, Yann-Erick Proy, and Patrick Roumanoff.
Verification of Real-Time Systems using Linear Relation Analysis.
Formal Methods in System Design, 11(2):157–185, August 1997.
A.1

[HR99] Nicolas Halbwachs and Pascal Raymond.
Validation of Synchronous Reactive Systems: from Formal Verification to Automatic Testing.
In *Proceedings of the Asian Computing Science Conference, ASIAN'99*, volume 1742 of *Lecture Notes in Computer Science*. Springer Verlag, December 1999.
A.1

[ID93a] C. Norris Ip and David L. Dill.
Better Verification Through Symmetry.
In *Proceedings of the 11th International Conference on Computer Hardware Description Languages and their Applications*, pages 87–100. Elsevier Science Publishers, April 1993.
A.3

[ID93b] C. Norris Ip and David L. Dill.

Efficient Verification of Symmetric Concurrent Systems.
In *Proceedings of the International Conference on Computer Designs: VLSI in Computers and Processors*, pages 230–234. IEEE Computer Society, October 1993.
A.3

[ID96a] C. Norris Ip and David L. Dill.
State Reduction Using Reversible Rules.
In *Proceedings of the 33rd Design Automation Conference, DAC'96*, pages 564–567. ACM Press, June 1996.
A.3

[ID96b] C. Norris Ip and David L. Dill.
Verifying Systems with Replicated Components in `Mur`φ.
In *Proceedings of the 8th International Conference on Computer Aided Verification, CAV'96*, volume 1102 of *Lecture Notes in Computer Science*, pages 147–158. Springer Verlag, July 1996.
A.3

[JPO95] Lalita Jategaonkar Jagadeesan, Carlos Puchol, and James Von Olnhausen.
Safety Property Verification of `Esterel` Programs and Applications to Telecommunications Software.
In *Proceedings of the 7th International Conference On Computer Aided Verification, CAV'95*, volume 939 of *Lecture Notes in Computer Science*, pages 127–140. Springer Verlag, 1995.
2.10.1

[KF98] Gila Kamhi and Limor Fix.
Adaptive Variable Reordering for Symbolic Model Checking.
In *Proceedings of the International Conference on Computer-Aided Design, ICCAD'98*. ACM Press, November 1998.
2.8

[KGP01] Andreas Kuehlmann, Malay K. Ganai, and Viresh Paruthi.
Circuit-based Boolean Reasoning.
In *Proceedings of the 38th Design Automation Conference, DAC'01*, pages 232–237. ACM Press, June 2001.
3.5.3

[KK97] Andreas Kuehlmann and Florian Krohm.
Equivalence Checking Using Cuts and Heaps.
In *Proceedings of the 34th Design Automation Conference, DAC'97*, pages 263–268. ACM Press, June 1997.
3.5.3

[Lee59] C. Y. Lee.
Representation of Switching Circuits by Binary-Decision Programs.

Bell System Technical Journal, 38:985–999, July 1959.
2.8

[LHR97] David Lesens, Nicolas Halbwachs, and Pascal Raymond.
Automatic Verification of Parameterized Linear Networks of Processes.
In *Proceedings of the 24th Symposium on Principles of Programming Languages, POPL'97*, pages 346–357. ACM Press, January 1997.
A.1

[LS99] Luciano Lavagno and Ellen M. Sentovich.
`ECL`: A Specification Environment for System-Level Design.
In *Proceedings of the 36th Design Automation Conference, DAC'99*. ACM Press, June 1999.
2.6.1

[Mal94] Sharad Malik.
Analysis of Cyclic Combinational Circuits.
IEEE Transactions on Computer-Aided Design, 13(7):950–956, July 1994.
3.3.2

[Mar91] Florence Maraninchi.
The `Argos` Language: Graphical Representation of Automata and Description of Reactive Systems.
In *Proceedings of the IEEE Workshop on Visual Languages*, October 1991.
2.6.2

[McM92] Ken L. McMillan.
Symbolic Model Checking: an Approach to the State Explosion Problem.
PhD thesis, Carnegie Mellon University, May 1992.
A revised edition is available in hardback from Kluwer Academic Publishers.
4.1.4

[Mea55] George H. Mealy.
A method for synthesizing sequential circuits.
Bell System Technical Journal, 34(5):1045–1079, 1955.
2.7

[Mey88] Bertrand Meyer.
Object-Oriented Software Construction.
Prentice-Hall, 1988.
3.8.2

[Mey91] Bertrand Meyer.
Eiffel: The Language.
Prentice-Hall, 1991.
3.8.2

[Mig] Frédéric Mignard.

Le processeur `sscoc` du compilateur `Esterel` v4_4x.
Technical report, CMA, Ecole des Mines and INRIA, ???
2

[Mig94] Frédéric Mignard.
Compilation du langage `Esterel` en systèmes d'équations booléennes.
PhD thesis, Ecoles des Mines de Paris, October 1994.
2

[MP92] Zohar Manna and Amir Pnueli.
The Temporal Logic of Reactive and Concurrent Systems.
Springer-Verlag, 1992.
2.10.1

[MS00a] Christoph Meinel and Christian Stangier.
Speeding Up Image Computation by Using RTL Information.
In *Formal Methods in Computer-Aided Design, FMCAD'00*, volume 1954 of *Lecture Notes in Computer Science*, pages 443–454. Springer, November 2000.
2.9.3

[MS00b] In-Ho Moon and Fabio Somenzi.
Border-Block Triangular Form and Conjunction Schedule in Image Computation.
In *Proceedings of the 3rd International Conference on Formal Methods in Computer-Aided Design, FMCAD'00*, volume 1954 of *Lecture Notes in Computer Science*, pages 79–90. Springer, November 2000.
2.9.3

[MS01] Christoph Meinel and Christian Stangier.
A New Partitioning Scheme for Improvement of Image Computation.
In *Proceedings of Asia South Pacific Design Automation Conference (ASP-DAC'01)*, pages 97–102. ACM Press, January 2001.
2.9.3

[MWBSV88] Sharad Malik, Albert Wang, Robert Brayton, and Alberto Sangiovanni-Vincentelli.
Logic Verification Using Binary Decision Diagrams in a Logic Synthesis Environment.
In *Proceedings of the International Conference on Computer Aided Design, ICCAD'88*, November 1988.
2.8

[oc598] *The `Lustre-Esterel` portable format, version oc5*, September 1998.
4.2

[PB01] Dumitru Potop-Butucaru.

Fast Redundancy Elimination Using High-Level Structural Information from `Esterel`.
Technical Report 4330, INRIA, 2001.
2.3.5, 2.5, 3.3.4

[PK00] Viresh Paruthi and Andreas Kuehlmann.
Equivalence Checking Combining a Structural SAT-Solver, BDDs, and Simulation.
In *Proceedings of the International Conference on Computer Design: VLSI in Computers & Processors, ICCD'00*. IEEE Computer Society Press, September 2000.
3.5.3

[RAB+95] Rajeev K. Ranjan, Adnan Aziz, Robert K. Brayton, Bernard Plessier, and Carl Pixley.
Efficient BDD Algorithms for FSM Synthesis and Verification.
In *Proceedings of the International Workshop on Logic Synthesis, IWLS'95*, May 1995.
2.9.3

[RHR91] Christophe Ratel, Nicolas Halbwachs, and Pascal Raymond.
Programming and Verifying Real-Time Systems by Means of the Synchronous Data-Flow Language `Lustre`.
In *Proceedings of the Conference on Software for Critical Systems*, pages 112–119. ACM Press, December 1991.
A.1

[RS95] Kavita Ravi and Fabio Somenzi.
High Density Reachability Analysis.
In *Proceedings of the International Conference on Computer Aided Design, ICCAD'95*. IEEE Computer Society Press, November 1995.
2.9.1, 2.9.3

[Rud93] Richard Rudell.
Dynamic Variable Ordering For Binary Decision Diagrams.
In *Proceedings of the International Conference on Computer-Aided Design, ICCAD'93*. IEEE Computer Society Press, November 1993.
2.8

[RWNH98] Pascal Raymond, Daniel Weber, Xavier Nicollin, and Nicolas Halbwachs.
Automatic Testing of Reactive Systems.
In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, December 1998.
A.1

[SBT96] Thomas R. Shiple, Gérard Berry, and Hervé Touati.
Constructive Analysis of Cyclic Circuits.

In *Proceedings of the International Design and Testing Conference, IDTC'96*. IEEE Computer Society Press, March 1996.
2.3.4, 1, 3.3.2

[SD95a] Ulrich Stern and David L. Dill.
Automatic Verification of the SCI Cache Coherence Protocol.
In *Proceedings of the Conference on Correct Hardware Design and Verification Methods, CHARME'95*, volume 987, pages 21–34. Springer-Verlag, 1995.
1

[SD95b] Ulrich Stern and David L. Dill.
Improved Probabilistic Verification by Hash Compaction.
In *Proceedings of the Conference on Correct Hardware Design and Verification Methods, CHARME'95*, volume 987, pages 206–224. Springer-Verlag, 1995.
4.5, A.3

[SD96] Ulrich Stern and David L. Dill.
A New Scheme for Memory-Efficient Probabilistic Verification.
In *Proceedings of the Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification*, volume 69 of *IFIP Conference Proceedings*, pages 333–348. Kluwer, October 1996.
4.5, A.2, A.3

[SD97] Ulrich Stern and David L. Dill.
Parallelizing the `Mur`φ Verifier.
In *Proceedings of the 8th International Conference on Computer Aided Verification, CAV'97*, volume 1254 of *Lecture Notes in Computer Science*, pages 256–278. Springer-Verlag, 1997.
A.3

[SD98] Ulrich Stern and David L. Dill.
Using Magnetic Disk Instead of Main Memory in the `Mur`φ Verifier.
In *Proceedings of the 9th International Conference on Computer Aided Verification, CAV'98*, volume 1427 of *Lecture Notes in Computer Science*, pages 172–183. Springer-Verlag, June 1998.
4.5, A.3

[SHM00] Christian Stangier, Ulrich Holtmann, and Christoph Meinel.
Optimizing Partitioning of Transition Relations by Using High-Level Information.
In *Proceedings of the International Workshop on Logic Synthesis, IWLS'2000*, May 2000.
2.9.3

[Som99] Fabio Somenzi.
Binary decision diagrams.
Calculational System Design, 173:303–366, 1999.
2.8, 3

[SSL+92] Ellen M. Sentovich, Kanwar Jit Singh, Luciano Lavagno, Cho Moon, Rajeev Murgai, Alexander Saldanha, Hamid Savoj, Paul R. Stephan, and Robert K. Brayton *et al.*.
`SIS`, A System for Sequential Circuit Synthesis.
Technical report, Dept. of Electrical Engineering and Computer Science, University of California, Berkeley, May 1992.
2.4

[STB96] Ellen M. Sentovich, Horia Toma, and Gérard Berry.
Latch Optimization in Circuits Generated from High-level Descriptions.
In *Proceedings of the International Conference on Computer Aided Design, ICCAD'96*, pages 428–435. IEEE Computer Society Press, November 1996.
Also available as INRIA Research Report RR-2943.
2.4

[STB97] Ellen M. Sentovich, Horia Toma, and Gérard Berry.
Efficient Latch Optimization Using Incompatible Sets.
In *Proceedings of the 34th Design Automation Conference, DAC'97*, pages 8–11. ACM Press, June 1997.
2.3.5

[Ste97] Ulrich Stern.
Algorithmic Techniques in Verification by Explicite State Enumeration.
PhD thesis, Technical University of Munich, 1997.
4.5, A.3

[Tea02] The Coq Development Team.
The Coq Proof Assistant Reference Manual.
INRIA and LRI, January 2002.
Available at `http://coq.inria.fr/doc/main.html`.
3.8.2

[Thi02] Xavier Thirioux.
Simple and Efficient Translation from LTL Formulas to Büchi Automata.
In *Proceedings of the 7th International Workshop on Formal Methods for Industrial Critical Systems, FMICS'02*, July 2002.
2.10.2

[THY93] Seiichiro Tani, Kiyoharu Hamaguchi, and Shuzo Yajima.
The Complexity of the Optimal Variable Ordering Problems of Shared Binary Decision Diagrams.

In *Proceedings of the 4th International Symposium on Algorithms and Computation, ISAAC'93*, volume 762 of *Lecture Notes in Computer Science*. Springer, December 1993.
2.8

[TSLaASV90] Herve J. Touati, Hamid Savoj, Bill Lin, and Robert K. Brayton an Alberto Sangiovanni-Vincentelli.
Implicit State Enumeration of Finite State Machines using BDD's.
In *Proceedings of the International Conference on Computer-Aided Design, ICCAD'90*. IEEE Computer Society Press, November 1990.
2.8

[Vis96] Willem Visser.
Memory efficient storage in `SPIN`.
In *Proceedings of the 2nd `SPIN` Workshop*, August 1996.
A.2

[WHL+01] Dong Wang, Pei-Hsin Ho, Jiang Long, James Kukula, Yunshan Zhu, Tony Ma, and Robert Damiano.
Formal Property Verification by Abstraction Refinement with Formal, Simulation and Hybrid Engines.
In *Proceedings of the 38th Design Automation Conference, DAC'01*, pages 35–40. ACM Press, 2001.
3.3.4

[Yan98] Chiang Han Yang.
Prioritized Model Checking.
PhD thesis, Stanford University, December 1998.
2.9.1, A.3

[YSAA97] Jun Yuan, Jian Shen, Jacob A. Abraham, and Adnan Aziz.
On Combining Formal and Informal Verification.
In *Proceedings of the 9th International Conference on Computer Aided Verification, CAV'97*, volume 1254 of *Lecture Notes in Computer Science*, pages 376–387. Springer-Verlag, 1997.
2.9.1, A.3

— Résumé —

Cette thèse traite des approches implicites et explicites, ainsi que de leur convergence, de l'exploration d'espace d'états atteignables de circuits logiques provenant de programmes réactifs synchrones écrits en `Esterel`, `ECL` ou `SyncCharts`. Nos travaux visent à réduire les coûts de ces explorations à l'aide de techniques génériques ou spécifiques à notre cadre de travail. Nous utilisons les résultats de ces explorations à des fins de vérification formelle de propriétés de sûreté, de génération d'automates explicites ou de génération de séquences de tests exhaustives. Nous décrivons trois outils.

Le premier outil est un vérificateur formel implicite, à base de Diagrammes de Décisions Binaires (BDDs). Ce vérificateur présente plusieurs techniques permettant de réduire le nombre de variables impliquées dans les calculs d'espace d'états. Nous proposons notamment l'abstraction de variables à l'aide d'une logique trivaluée. Cette nouvelle méthode étend la technique usuelle de remplacement de variables d'états par des entrées libres. Ces deux méthodes calculant des sur-approximations de l'espace d'états atteignables, nous proposons différentes techniques utilisant des informations concernant la structure du modèle et permettant de réduire la sur-approximation.

Le deuxième outil est un moteur d'exploration explicite, basé sur l'énumération des états accessibles. Ce moteur repose sur la simulation de la propagation du courant électrique dans les portes du circuit et supporte les circuits cycliques. Ce moteur comporte de nombreuses optimisations et fait appel à différentes heuristiques visant à éviter les explosions en temps ou en espace inhérentes à cette approche, ce qui lui confère de très bonnes performances. Ce moteur a été appliqué à la génération d'automates explicites et à la vérification formelle.

Enfin, le troisième outil est une évolution hybride implicite/explicite du moteur purement explicite. Dans cette évolution, les états sont toujours analysés individuellement mais symboliquement à l'aide de BDDs. Ce moteur a également été appliqué à la génération d'automates explicites, mais il est plutôt destiné à la vérification formelle ou la génération de séquences de tests exhaustives.

Nous présentons des résultats d'expérimentations de ces différentes approches sur plusieurs exemples industriels.

— Abstract —

This thesis deals with implicit and explicit approaches, as well as the convergence of these approaches, to the reachable state space exploration of logical circuits generated from synchronous reactive programs written in `Esterel`, `ECL` or `SyncCharts`. Our work aim at reducing the cost of these explorations either by the way of generic techniques or techniques that are specific to our context. We apply the results of these explorations to formal verification of safety properties, explicit automaton generation or exhaustive test sequence generation. We describe three tools.

The first tool is an implicit formal verifier based on Binary Decision Diagrams (BDDs). This verifier provide several techniques aiming at reducing the number of variables that are involved in reachable state space computations. We provide in particular a variable abstration technique based on the use of a trivalued logic. This new technique extends the usual technique in which state variables are replaced by free inputs. As these two techniques compute over-approximations of the reachable state space, we provide several methods aiming at reducing this over-approximation by using structural information concerning the model.

The second tool is an explicit exploration engine based on the enumeration of reachable states. This engine is based on the simulation of the electric current propagation within the circuit and it provides transparent support for cyclic circuits. This engine includes numerous optimisations and uses several heuristics aiming at avoiding explosions in time or space which are inherent to this approach, thus providing very good performances. This engine has been applied to explicit automaton generation and formal verification.

Finally, the third tool is an hybrid implicit/explicit evolution of the pure explicit engine. In this version, states are still analyzed one by one but in a symbolic way, using BDDs. This engine has also been applied to explicit automaton generation and formal verification as well as exhaustive test sequence generation.

We present experiment results of these different approaches on several industrial examples.