UNIVERSITÉ PIERRE ET MARIE CURIE – PARIS VI
LABORATOIRE D'INFORMATIQUE DE PARIS 6 – LIP6
SYSTÉMES EMBARQUÉS SUR PUCE – SOC

Doctorat Spécialité Informatique

Conception Incrémentale, Vérification de Composants Matériels et Méthode d'Abstraction pour la Vérification de Systèmes Intégrés sur Puce

Cécile Braunstein

Soutenue le : 14 mai 2007

Devant le jury composé de :

Mme.	Béatrice Bérard	Rapporteur
M.	Hans Eveking	Rapporteur
Mme.	Dominique Borrione	Examinateur
M.	Alain Greiner	Examinateur
Mme.	Emmanuelle Encrenaz	Encadrante de Thèse
Mme.	Alix Munier-Kordon	Directrice de Thèse

A Bina et Maurice
(pépé et mémé)

Remerciements

Je tiens à exprimer ici ma reconnaissance à toutes les personnes qui m'ont aidé, encouragé ou tout simplement soutenu tout au long de mon travail de thèse. En premier lieu, le département SOC du laboratoire d'informatique de Paris 6 pour avoir soutenu mes travaux.

Je voudrais également remercier Béatrice Bérard et Hans Eveking d'avoir accepté d'évaluer mon manuscrit. J'ai particulièrement apprécié vos rapports détaillés et approfondis témoignant de l'intérêt que vous lui avez porté. Merci aussi à Dominique Borrione, Alix Munier et Alain Greiner pour avoir accepté de faire partie du jury de la soutenance. Merci à Emmanuelle Encrenaz pour m'avoir confié ce sujet de thèse, pour avoir su me guider et diriger mes travaux pendant 4 ans, merci aussi pour m'avoir donné le goût de la recherche et l'envie d'aller encore plus loin.

A titre personnel, je remercie toute ma famille pour leur inconditionnel soutient et leur confiance qui me tient à coeur. Merci à ma mère pour m'avoir transmis son caractère volontaire et m'avoir aidé à choisir ma tenue. Merci à mon père pour le goût des sciences et de l'enseignement. Merci à ma soeur, Sarah, qui s'occupe de notre misérable condition d'étudiant (Est ce que j'ai gagné ma quatrième dimension ?). Merci à mon grand père pour être le premier docteur de la famille. Merci à ma grand mère pour l'organisation du pot (le plus important !).

Je tiens à remercier Jean-Lou Desbarbieux pour m'avoir introduit dans ce labo et m'avoir présenté Emma. Merci à Michaël Sultan pour mes débuts à la fac. Un Big-up à Soph.

Merci à tous ceux qui me font croire en l'université et la recherche publique.

... et puis merci à Tim de m'avoir supporté et aidé, tu m'as ouvert la voix, c'est à nous de jouer maintenant !

Résumé

Cette thèse traite de la vérification formelle par model checking de systèmes intégrés sur puce. Nous proposons d'abord une méthode de conception incrémentale pour la vérification d'un composant matériel. Cette méthode est un cadre de conception par ajouts successifs de nouveaux comportements. Nous avons montré que cette méthode assure la non-régression d'un composant tout au long de sa conception. D'autre part, cette méthode permet aussi de faire évoluer la spécification d'un composant en prenant en compte les différentes fonctionnalités ajoutées au cours de la conception. Nous avons ensuite particularisé cette approche pour la conception et la vérification d'architectures pipelines. Cette méthode a été utilisée avec succès pour la conception de convertisseurs de protocole.

La vérification par model-checking d'un système intégré sur puce se confronte au problème d'explosion combinatoire. Les techniques d'abstractions sont des méthodes efficaces pour alléger ce problème. Nous exposons un algorithme d'abstraction basé sur la spécification de chaque composant. Cet algorithme construit une structure de Kripke représentant un sous-ensemble des formules CTL tirées de la spécification. Cette construction se place dans un contexte de raffinement d'abstraction guidé par l'étude du contre-exemple produit par le model checker. Les premières expérimentations que nous avons réalisées montrent un gain considérable en temps de vérification et un accroissement conséquent de la taille du système vérifié. Ces résultats nous confortent sur l'intérêt de cette méthode d'abstraction.

Mots Clés

Architecture matérielle, SoC, Vérification formelle, CTL, Abstraction, CEGAR, Convertisseur de protocole

Abstract

This thesis deals with formal verification of integrated system on chip by model checking. We propose an incremental design process for the verification of a hardware component. This method is a framework for designing a component by successively adding some new behaviours. The incremental design process guarantees, *by construction*, the non-regression of a component during the design process. Moreover, the specification is automatically derived with taking into account the new behaviours. We apply our method with success for the design of a protocol converter.

The formal verification process of system on chip suffers from the states explosion problem. Abstraction techniques aim at alleviating this problem. We state an abstraction algorithm based on the specification of each component. We construct a Kripke structure directly from a subset of the specification written with CTL formulas. This abstraction takes place in a counter-example guided abstraction refinement framework. The first experiments show an important benefit in terms of verification time and an increase in the size of the system we are now able to check. This result reinforces the interest of our abstraction method.

Keywords

Hardware design, SoC, formal verification, CTL, Abstraction, CEGAR, protocol converter

Table des matières

Remerciements **i**

Résumé **iii**

Abstract **v**

Table des matières **ix**

Table des figures **ix**

Liste des tableaux **xii**

Introduction **1**

1 La Vérification de SOC par model checking **5**
1.1 Modéliser les composants matériels 6
1.1.1 Les machines de Moore 6
1.1.2 Les structures de Kripke 8
1.1.3 La composition de structures 9
1.2 La spécification CTL 9
1.3 Vérification de composants et de leur composition 11
1.3.1 Relations entre les modèles 12
1.3.2 Abstraction-Raffinement par contre-exemple (CEGAR) 13
1.3.3 Paradigme "assume-guarantee" 14

I Une Méthode de Conception Incrémentale **17**

2 Conception de composant par ajout **19**
2.1 Idée générale 20
2.1.1 La conception incrémentale 20
2.1.2 Les méthodes par raffinement 21
2.1.3 Les méthodes d'intégrations de nouveaux services 23
2.2 Définition d'un incrément 24
2.3 Spécification CTL de l'ajout 29
2.4 Peut-on tout modéliser sous forme d'incrément ? 31

3 Couplage conception incrémentale et vérification incrémentale 33
3.1 Propriétés d'un composant incrémenté 35
3.1.1 Automate de Moore M_{i+1} 35
3.1.2 Structure de Kripke $K(M_{i+1})$ 36
3.2 Conséquence sur les spécifications 40
3.2.1 Transformation de la spécification de $K(M_i)$ 40
3.2.2 Incorporation de la spécification de l'incrément 42

4 Cas particulier du contrôle de flux 45
4.1 Formalisation du contrôle de flux 46
4.2 Ajout de bégaiement 49
4.2.1 L'événement *stall* 49
4.2.2 L'incrément de bégaiement 50
4.2.3 Propriétés de l'incrément *stall* 51
4.2.4 Ajout de destruction 54
4.3 Vérification du contrôle de flux 56
4.3.1 Les classes de propriétés **CPI** et **CPE** 56
4.3.2 Transformations et préservations de la spécification 57

5 Application aux convertisseurs de protocoles 61
5.1 La Plate-forme d'expérimentation 63
5.1.1 Les protocoles VCI et PI 63
5.1.2 La plate-forme 64
5.2 Hierarchie des wrappers VCI/PI 65
5.2.1 Le modèle initial 65
5.2.2 Les différents événements compatibles 68
5.3 Conception incrémentale des wrappers VCI/PI 69
5.3.1 Incrément 1 (de **A** à **B**) 69
5.3.2 Incrément 2 (de **B** à **B'**) 70
5.3.3 Incrément 3 (de **B'** à **C'**) 70
5.3.4 Incrément 4 (de **C'** à **C"**) 71
5.4 Évolution de la spécification des wrappers VCI/PI 71

II Formules CTL comme Abstraction de Composants 75

6 Démarche CEGAR pour la vérification de SoC 77
6.1 Abstraction de composants 78
6.2 Notre démarche d'abstraction à partir des formules CTL 79
6.3 Définition d'un composant abstrait 81
6.3.1 Structure de Kripke équitable 81
6.3.2 Structure de Kripke Abstraite (AKS) 81

7 Spécification comme abstraction 83
7.1 Construction d'une AKS à partir de formules CTL 83
7.1.1 Définitions Préliminaires 83
7.1.2 Composition de structures 84
7.2 Algorithme de construction 84
7.2.1 Quantificateur universel **A** 85
7.2.2 Quantificateur existentiel **E** 87
7.3 Propriétés d'un AKS construit à partir d'une formule 92
7.3.1 Propriétés de K_φ 92
7.3.2 Composition des abstractions 93
7.3.3 Vérification du système abstrait par model checking 94

8 Application à la plate-forme VCI-PI 97
8.1 Détails sur la boucle CEGAR de la propriété 1 97
8.2 Résultats de l'abstraction 99

Conclusions et perspectives 103

Annexes 104

A Theorem 1 partial proof : Proofs of each basic cases 105

B Wrappers VCI-PI 113
B.1 Wrapper maître A 113
B.2 Wrapper maître B 113
B.3 Wrapper maître B' 113
B.4 Wrapper maître C' 113

C Abstraction de composant 117

Bibliographie 121

Table des figures

1.1 Circuit séquentiel d'une machine de Moore 6
1.2 Représentation d'un automate de Moore 7
1.3 Représentation d'une structure Kripke 9
1.4 Boucle CEGAR 14

2.1 La méthode de conception incrémentale 21
2.2 Méthode de conception à la B 22
2.3 Composant M_{i+1} obtenu par incrément à partir de M_i 25
2.4 Exemple d'incrément valide 26
2.5 Frontière e_act 28
2.6 Les trois types d'incréments concernés par l'extension 30

3.1 Relation entre les spécifications des modèles 35
3.2 Règles incrémentales 37
3.3 Illustration des corollaires 38
3.4 Frontière des valeurs actives dans $K(M_{i+1})$ 40

4.1 Suite de traitements 46
4.2 Le flux optimal pour un composant à 2 étages 48
4.3 M_i incrémenté par $stall_2$ 50
4.4 Progression du préfixe 51
4.5 Progression bégayante du préfixe 53
4.6 Graphe d'exécution de la machine de Moore avec un kill à l'étage 1 55
4.7 Découplage commande résultat 57

5.1 Protocole BVCI 63
5.2 Protocole du PI-bus 64
5.3 Interface des wrappers VCI/PI 64
5.4 Illustration d'un transfert sur la plate-forme de traduction VCI-PI-VCI . . 65
5.5 Automate du wrapper maître 66
5.6 Chemin de donnée du wrapper maître 67
5.7 Chronogramme de la gestion du signal WAIT 69

6.1 Notre boucle de CEGAR 80

7.1 AKS de la formule $\mathbf{AF}p$ 86
7.2 Cas de base pour ACTL 88

7.3 Cas de base ECTL 90
7.4 Construction de l'AKS de la formule $\mathbf{EG}(\mathbf{AF}(p \vee q))$ 91
7.5 Comparaison entre une structure concrète et des structures abstraites . . . 92

8.1 Initialisation de la boucle CEGAR 98
8.2 Premier raffinement de la boucle CEGAR 99
8.3 Second raffinement de la boucle CEGAR 99

B.1 Automate du wrapper A 114
B.2 Automate du wrapper B 115
B.3 Automate du wrapper B' 116
B.4 Automate de l'incrément RETRACT et des connexions avec B' 116

Liste des tableaux

2.1 Exemple d'interactions de services 23

5.1 Hiérarchie des wrappers maîtres VCI/PI 68
5.2 Inclusion des wrappers maîtres VCI/PI 69
5.3 Temps de Vérification et mémoire requise 74

6.1 Information sur une proposition atomique p dans un état s 82

8.1 Résultats comparatifs des modèles concrets et des modèles abstraits 100

Introduction

Les avancées technologiques dans la conception et la fabrication des systèmes matériels amènent à une complexité croissante des circuits. On parle aujourd'hui de systèmes embarqués sur puce ou SoC (system on chip). Un SoC est un ensemble d'éléments interagissant les uns avec les autres, implémentés sur une même puce. Chacun des éléments concoure à l'élaboration d'un ensemble de tâches complexes en interaction les unes avec les autres. La conception de ces systèmes est réalisée à partir d'éléments préconçus (appelés coeurs ou "IP cores"), qui peuvent être réutilisés pour la conception de différents SoC. Ils sont de natures diverses : fonction analogiques (convertisseurs analogique/numérique, filtres...), fonctions numériques matérielles (décodeurs, filtres ...), fonctions logicielles (sur DSP, micro-contrôleur ...). La conception de tels systèmes est morcelée : souvent les différents IPs sont créés séparément, ils proviennent d'équipes de conception (académiques ou industriels) différentes. La difficulté est d'une part, de créer des IPs qui peuvent s'adapter facilement à différents environnements. D'autre part, une autre difficulté est de composer l'ensemble des éléments et de garantir que cette composition réalise les fonctionnalités attendues.

La validation d'un SoC peut être réalisée par simulation et/ou par vérification formelle. Il existe deux principaux aspects à la validation des SoCs. Le premier est la validation des éléments constitutifs du SoC. Chaque élément étant réutilisable, nous devons être capables de vérifier un composant dans des environnements variables. Ensuite, nous voulons vérifier l'assemblage des différents éléments, c'est à dire les interactions entre les différents composants, afin de vérifier le SoC complet. Une des stratégies les plus utilisées consiste à exploiter la validation de chacun des composants pour la validation du système entier. La vérification des composants doit donc être également réutilisable.

Traditionnellement, la méthode utilisée pour la validation de tels systèmes et de ses composants est la simulation. Outre le fait que cette approche est très coûteuse en temps, en pratique, il est impossible de simuler un circuit de manière exhaustive pour garantir la validité d'un ou plusieurs composants. La prise en compte des différents environnements possibles augmente considérablement le nombre de cas de test. Néanmoins, cette approche reste très utilisée par les concepteurs et permet de repérer des erreurs très tôt lors de la conception d'un composant.

Une alternative à la simulation est l'utilisation des méthodes formelles. La vérification formelle de systèmes (matériels ou logiciels) est définie par [Kro99] de la manière suivante : établir mathématiquement qu'une implémentation est correcte vis-à-vis de sa spécification. L'implémentation est le modèle de l'architecture décrit dans un langage de description matériel (VHDL, SystemC, Verilog). La spécification correspond à un ensemble de propriétés à vérifier pour statuer sur la validité de l'implémentation. Elle peut

être exprimée par des formules de logique, une description comportementale ...

Un aperçu des différentes techniques de vérification formelles est présenté dans les articles suivants [Gup92, KG99]. Dans cette thèse nous nous sommes intéressées à la vérification par model checking de propriétés CTL. Le model checking a l'avantage d'être complètement automatique. Un outil qui implémente un algorithme de model checking est appelé un model checker. Il permet de déterminer si une spécification du système est vraie ou sinon, pour une propriété non vérifiée par le système, il fournit un contre-exemple d'exécution du système. Un des problèmes majeurs de ces algorithmes est l'explosion combinatoire du nombre d'états du système à vérifier. Les systèmes sont trop grands et trop complexes pour pouvoir être vérifiés à l'aide du model checking. Grâce à différentes techniques comme l'abstraction [CGL94], la composition [Lon93], les raisonnements "assume-guarantee" ([HQR98]) et grâce à l'évolution des structures de données (BDD [Bry86], DDD[CEPA+02], SDD[CTM05]) utilisées par les algorithmes de model checking ([McM93, BCM+92]), la complexité des systèmes vérifiés automatiquement ne cesse d'augmenter.

Un autre problème majeur de la vérification formelle est lié au fait que jusqu'à présent le travail de vérification n'est pas fait par les concepteurs de composants matériels. Les concepts de modélisation imposés par la vérification formelle, ne sont pas toujours adaptés aux concepteurs. En effet, les modèles utilisés sont souvent trop abstraits pour amener facilement à la synthèse de circuit. Bien que de nombreux outils offrent maintenant la possibilité de synthèse automatique de modèle abstrait, le concepteur reste souvent très proche d'une description RTL (Register Transfer Level, niveau transfert de registres) pour la conception d'architecture.

Dans cette thèse nous nous attaquons aux deux problèmes de vérification formelle exposés précédemment. La première partie de ce manuscrit propose une méthode de conception incrémentale pour un composant, qui offre l'avantage de rester proche des méthodes de travail d'un concepteur tout en réalisant le lien entre conception et la vérification. La seconde partie propose une méthode d'abstraction de composant à partir de leur spécification pour la vérification de systèmes mettant en oeuvre plusieurs composants.

Partie I : Une méthode de conception incrémentale

La méthode de conception, que nous avons formalisée permet de réduire la distance entre les concepteurs et vérification formelle. Elle part d'une idée simple qui calque la méthode employée par de nombreux concepteurs. Au lieu de concevoir directement l'ensemble d'un système, le concepteur adopte une démarche incrémentale. Pour la réalisation d'un composant, il ne s'attaque pas à toutes les difficultés en même temps. Il avance par la résolution d'une difficulté à chaque étape jusqu'à la réalisation complète du système. De plus, pour faciliter encore la conception, son point de départ est une réalisation très simple du système. Cette première réalisation comprend l'ensemble des comportements de base du systèmes qui ne prend pas en compte tous les événements possible de l'environnement. Par exemple pour la réalisation d'un processeur, la première implémentation sera la description de la mise en oeuvre de l'ensemble des instructions mais sans prendre en compte les différents délais imposés par la hiérarchie mémoire, ni les exceptions ou les

interruptions possibles.

Ensuite, à chaque étape de conception, le modèle est enrichi par de nouveaux comportements. Une des priorités du concepteur est alors de s'assurer que les ajouts ne viennent pas détruire l'existant. Chaque étape est la résolution d'un problème, il ne faudrait pas avoir à tout refaire à chaque étape. La formalisation de notre méthode garantit la non-régression du composant *par construction*. Si les ajouts suivent notre formalisation alors le concepteur n'a plus à s'inquiéter de la destruction des anciens comportements lorsque de nouveaux sont ajoutés.

Une autre difficulté qui crée la distance entre vérification et conception est le travail difficile et laborieux d'écrire une spécification dans un langage formel. La méthode que nous proposons permet de dériver à partir de la spécification du composant initial et dans certains cas de l'ajout lui-même, une partie de la spécification du composant plus complexe. En effet, notre méthode garantit la non-régression des comportements, la spécification doit donc pouvoir être exportée pour faire partie de la spécification du système complet. En outre, l'ensemble de la spécification obtenue n'a pas besoin d'être revérifié par un outil de model checking, cela réduit aussi le temps de vérification d'un composant.

Partie II : Formules CTL comme abstraction de composants

Les SoCs sont composés d'éléments le plus souvent pré-vérifiés et munis d'une spécification, mais la vérification globale du système reste un problème majeur. L'interaction des différents composants peut amener à des fonctionnements incorrects du système. De plus, les systèmes composés sont beaucoup trop grands et trop complexes pour être vérifiés directement.

Nous tirons parti de la décomposition naturelle des SoCs et de la spécification déjà vérifiée de chaque composant afin de vérifier le système complet. Nous construisons un système plus abstrait où chaque composant est remplacé par une abstraction. L'abstraction de chaque composant est directement obtenue par la transformation d'un sous-ensemble de sa spécification, pertinente pour la vérification d'une propriété globale.

La construction d'une abstraction n'amène pas toujours directement à la résolution de la validité (ou non) d'une propriété globale. En effet, l'abstraction enlève des détails de l'implémentation et par conséquent, il est possible d'obtenir des propriétés erronées sur le modèle abstrait qui ne le sont pas sur le modèle concret. Il faut alors raffiner l'abstraction. Nous avons donc intégré notre algorithme d'abstraction dans un cadre de raffinement d'abstraction guidé par contre-exemple (CEGAR [Cla03]).

Chapitre 1

La Vérification de SOC par model checking

Ce chapitre présente de façon générale le processus de vérification de composants matériels et de leur composition par model checking. Le model checking sert à vérifier si des systèmes de transitions sont des modèles pour des propriétés. Dans cette thèse, nous nous limitons aux systèmes de transitions finis, obtenus par composition synchrone de composants élémentaires. L'avantage de cette technique est qu'elle est complètement automatisable. Vérifier une architecture matérielle en utilisant le model checking peut être décomposé en différentes tâches :

La modélisation : La première tâche consiste à convertir un système, décrit dans un langage de description de matériel (HDL), en un formalisme accepté par un outil de model checking. La plupart du temps, cette étape est automatiquement réalisée par un compilateur. Dans notre étude, le système sera décrit sous forme de machine de Moore et le formalisme utilisé par le model checker est une représentation en structure de Kripke.

La spécification : Le système à vérifier doit être conforme à une spécification. Une spécification décrit les propriétés comportementales attendues du système. Les logiques temporelles sont souvent utilisées pour exprimer les comportements qui évoluent avec le temps. Ici nous nous sommes concentrées sur la logique temporelle arborescente CTL. Le sens des formules logiques CTL est déterminé relativement à l'arbre d'exécution infini obtenue par "dépliage" d'une structure de Kripke.

La vérification Idéalement la vérification est automatique. Néanmoins, en pratique l'intervention humaine est souvent nécessaire. Lorsqu'une propriété n'est pas vérifiée, un contre-exemple est fourni, il guide l'utilisateur pour trouver les erreurs. D'autre part, beaucoup de systèmes sont encore trop grands en terme d'espace mémoire, pour être vérifiés. Dans ce cas il peut être nécessaire d'ajuster le modèle. C'est à dire simplifier ou abstraire le modèle ou encore adopter une démarche compositionnelle.

Dans une première partie, nous nous sommes concentrées sur les problèmes de modélisation et de spécification pour un composant matériel. Plus précisément, nous nous sommes intéressées aux liens entre les méthodes de conception d'un composant et l'évo-

lution de sa spécification. Dans une seconde partie, nous avons mis en place une méthode de vérification pour une composition d'éléments.

1.1 Modéliser les composants matériels

1.1.1 Les machines de Moore

Les composants que nous voulons concevoir sont des circuits séquentiels synchrones. Ils peuvent être représentés par des machines d'états finis. Ils présentent une interface composée d'un ensemble de signaux d'entrées et de sorties permettant de communiquer avec son environnement. L'état interne du système est défini par la valeur de ces éléments mémorisants (registres). Le comportement du système est alors décrit comme une succession d'états. Une horloge globale cadence le système, à chaque "top" d'horloge, l'état futur est calculé par un ensemble de fonctions booléennes -*fonction de transition*- qui dépend de la configuration des entrées et de l'état courant du système. La valeur des signaux de sortie est un ensemble de fonctions booléennes -*fonction de génération*- qui dépend aussi de la configurations des entrées et de l'état courant du système. Dans notre contexte, nous considérons un cas particulier des machines d'état fini, où la fonction de génération ne dépend que de l'état courant : les *Machines de Moore*. La figure 1.1 représente une machine de Moore ayant n entrées I_i et m sorties O_j.

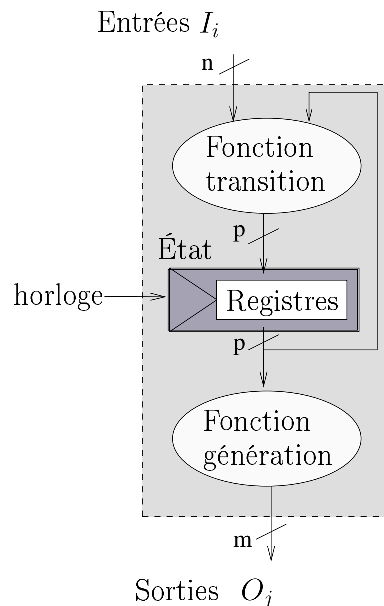


Fig. 1.1 – Circuit séquentiel d'une machine de Moore

Le modèle utilisé pour représenter la succession d'états d'une machine de Moore est un automate de Moore. Dans ce document nous ne considérons que les automates de Moore complets et déterministes. La définition formelle d'un signal, d'une configuration de signaux et d'un automate de Moore complet et déterministe est donnée dans la définition.

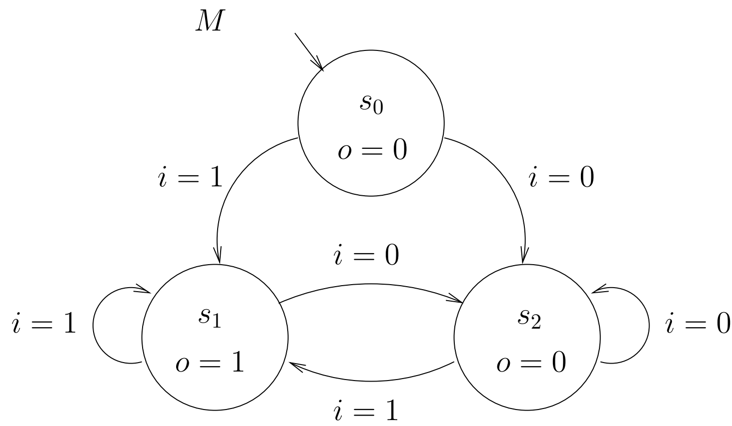


FIG. 1.2 – Représentation d'un automate de Moore

Définition 1.1 *Un* signal *est défini par un nom de variable α et un ensemble fini de valeurs possibles associées $\mathcal{D}om(\alpha)$,* domaine de définition *de α.*

Définition 1.2 *Soit un ensemble de signaux $E = \{\alpha_1, \ldots, \alpha_n\}$. L'*ensemble de toutes les configurations *possibles de E, noté $\mathcal{C}(E)$ est l'ensemble : $\mathcal{D}om(\alpha_1) \times \ldots \times \mathcal{D}om(\alpha_n)$. Une* configuration *$c(E)$ est une conjonction d'affectation d'une valeur pour chaque signal de E.*

Définition 1.3 *Un* automate de Moore *complet et déterministe $M = \langle S, I, O, T, G, s_0 \rangle$ est un 6-uplet tel que :*

S : Ensemble fini d'états ;

I : Ensemble fini des signaux d'entrées avec leur domaine de définition ;

O : Ensemble fini des signaux de sorties avec leur domaine de définition ;

$T \subseteq S \times \mathcal{C}(I) \times S$: Ensemble fini de transitions tel que,
$\forall s \in S$, $\forall c \in \mathcal{C}(I)$, $\exists!$ $s' \in S$ tel que. $(s, c, s') \in T$;

$G = \{g_0, \ldots, g_{|O|-1}\}$: Vecteur de fonctions de génération, chaque fonction définit la valeur d'un unique signal de sortie pour chaque état ; pour tous les signaux de sorties o_j $0 \leq j < |O|$ on a $g_j : S \to \mathcal{D}om(o_j)$;

$s_0 \in S$: un état initial.

Remarque 1 Le vecteur de fonctions de génération pour un état donné produit une configuration $c(O)$. Les signaux de sortie de l'automate de Moore ne dépendent que de l'état des registres (pas des signaux d'entrées), par conséquent $c(O)$ définit l'état courant de la machine de Moore.

Notation On introduit les notations suivante :

$s \to s'$ où s' est un successeur de s : s et $s' \in S$, il existe $c \in \mathcal{C}(I)$ tel que $(s, c, s') \in T$.

$\pi = s_0 \stackrel{*}{\to} s_n$ est un chemin à partir de s_0 tel que pour tout $i \in [0, n-1]$ $s_i \to s_{i+1}$.

La figure 1.2 représente un automate de Moore M tel que

- $S = \{s_0, s_1, s_2\}$;
- $I = \{i\}$ avec $Dom(i) = \{0, 1\}$;
- $O = \{o\}$ avec $Dom(o) = \{0, 1\}$;
- $T = \{(s_0, (i = 1), s_1), (s_1, (i = 1), s_1), (s_0, (i = 0), s_2), (s_2, (i = 0), s_2), (s_1, (i = 0), s_2), (s_2, (i = 1), s_1)\}$;
- $G = \{g_0\}$, g_0 est définie telle que $g_0(s_0) = 0$, $g_0(s_1) = 1$, $g_0(s_2) = 0$.

Dans tous les schémas, les états pointés par une flèche sans état source sont les états initiaux.

1.1.2 Les structures de Kripke

Les outils de model checking ont pour la plupart un traducteur automatique de description en automate de Moore en une description en structure de Kripke. L'outil VIS [gro96] accepte du Verilog et l'outil RuleBase [gro05] du VHDL et du Verilog. De façon informelle la traduction est obtenue en incorporant dans les états de la structure de Kripke, la configuration des entrées qui étiquettent les transitions de l'automate de Moore.

Définition 1.4 *Une* structure de Kripke *est un 5-uplet* $K = \langle AP, \Sigma, \Sigma_0, \mathcal{L}, R\rangle$ *tel que :*

AP *: Ensemble fini de propositions atomiques.*

Σ *: Ensemble fini d'états ;*

$\Sigma_0 \subseteq \Sigma$ *: Ensemble fini d'états initiaux ;*

$\mathcal{L} : \Sigma \rightarrow 2^{AP}$ *: fonction d'étiquetage qui associe à chaque état l'ensemble des propositions atomiques vraies dans cet état ;*

$R \subseteq \Sigma \times \Sigma$ *: Relation de transition.*

Définition 1.5 *(adaptation de [CLM89]) Soit M un automate de Moore, on peut déduire la* structure de Kripke $K(M) = \langle AP_{K(M)}, \Sigma_{K(M)}, \Sigma_{K(M),0}, \mathcal{L}_{K(M)}, R_{K(M)}\rangle$

$AP_{K(M)} = I \cup O$;

$\Sigma_{K(M)} = S \times \mathcal{C}(I)$;

$\Sigma_{K(M),0} = \{s_0\} \times \mathcal{C}(I)$;

$\mathcal{L}_{K(M)}$ *:* $p \in \mathcal{L}_{K(M)}(\sigma), \sigma = (s, c) \in \Sigma_{K(M)}$ *ssi* $\{p \in c \vee \exists d \in Dom(O), p = (g_o(s) = d)\}$

$R_{K(M)} \subseteq \Sigma_{K(M)} \times \Sigma_{K(M)}$ *et* $\forall\ (s, c_i) \in \Sigma_{K(M)}$, $\forall (s', c'_i) \in \Sigma_{K(M)}$, *on a* $((s, c_i), (s', c'_i)) \in R_{K(M)}$ *ssi* $(s, c_i, s') \in T$.

La figure 1.3 illustre la transformation de l'automate de Moore M figure 1.2 en une structure de Kripke $K(M)$. L'état s_0 est transformé en 2 états σ_0 et σ'_0, chacun est maintenant étiqueté par s_0 et une configuration du signal d'entrée i. L'état $(s_0, i = 1)$ est le prédécesseur de tous les états étiquetés par s_1, alors que l'état $(s_0, i = 0)$ est le prédécesseur de tous les états étiquetés par s_2.

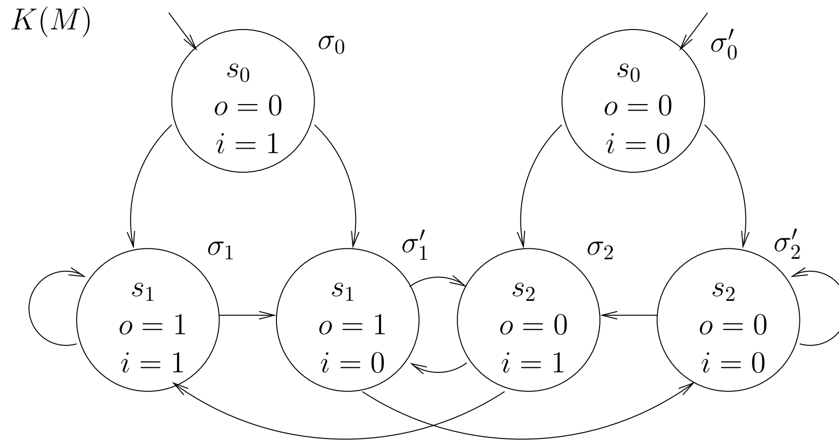


Fig. 1.3 – Représentation d'une structure Kripke

1.1.3 La composition de structures

La composition parallèle de deux structures de Kripke a été définie par Clarke et al. dans [CLM89]. Cette définition modélise des comportements synchrones. Les états sont des paires d'états de chaque composant qui sont cohérents vis-à-vis de leur propositions atomiques : pour un état de la composition, il est impossible d'avoir p et sa négation $\overline{p}$ vraies simultanément. Chaque transition du modèle composé correspond à une transition valide dans chacun des composants.

Définition 1.6 *Soient* $K_1 = \langle AP_1, \Sigma_1, \Sigma_{0_1}, \mathcal{L}_1, R_1 \rangle$ *et* $K_2 = \langle AP_2, \Sigma_2, \Sigma_{0_2}, \mathcal{L}_2, R_2 \rangle$ *deux structures de Kripke. La* composition synchrone *de* K_1 *et* K_2 *notée* $K_1 \parallel K_2$ *est la structure* K *défini de la façon suivante :*

$AP = AP_1 \cup AP_2$

$\Sigma = \{(\sigma_1, \sigma_2) \mid \mathcal{L}_1(\sigma_1) \cap AP_2 = \mathcal{L}_2(\sigma_2) \cap AP_1\}$

$\Sigma_0 = (\Sigma_{0_1} \times \Sigma_{0_2}) \cap \Sigma$

$\mathcal{L}((\sigma_1, \sigma_2)) = \mathcal{L}_1(\sigma_1) \cup \mathcal{L}_2(\sigma_2)$

$R \subseteq \Sigma_1 \times \Sigma_2 \rightarrow \Sigma_1 \times \Sigma_2$, *c'est à dire* $((\sigma_1, \sigma_2), (\sigma_1', \sigma_2')) \in R$ *ssi* $(\sigma_1, \sigma_1') \in R_1$ *et* $(\sigma_2, \sigma_2') \in R_2$

1.2 La spécification CTL

Les logiques temporelles permettent d'exprimer l'ordre des événements dans le temps. Nous utilisons la logique temporelle CTL (Computational tree logic) [CE81]. Les formules CTL décrivent des propriétés sur une arborescence infinie obtenue par le dépliage d'une structure de Kripke. Les formules CTL sont construites à partir des :

- Formules atomiques composées de propositions atomiques. Elles expriment l'information pour un ensemble d'états donnés du système ;
- Opérateurs booléens usuels :$\neg$, $\wedge$, $\vee$, $\rightarrow$;
- Quantificateur de chemins : **A**,**E**

– Opérateurs temporels qui décrivent des propriétés sur les chemins de l'arbre.

Tous les opérateurs sont interprétés par rapport à un "état courant", et chaque opérateur est composé de deux parties. La première partie correspond à un quantificateur de chemin **A** ou **E**. **A** exprime le fait qu'un événement est vrai pour tous les chemins à partir de l'état courant. En revanche **E** spécifie l'existence d'un chemin à partir de l'état courant qui vérifie une certaine propriété. La seconde partie contient un opérateur qui décrit l'ordre des événements le long d'un ou des chemins du dépliage de la structure de Kripke. La définition intuitive de ces opérateurs est donnée de la manière suivante :

- $\mathbf{X}\varphi$: **X** est l'opérateur *Next*. La formule $\mathbf{X}\varphi$ est vraie sur un chemin si la formule φ est vraie pour l'état successeur de l'état courant.
- $\mathbf{F}\varphi$: **F** est l'opérateur *Finally*. La formule $\mathbf{F}\varphi$ est vraie sur un chemin si la formule φ est vraie pour un état du chemin.
- $\mathbf{G}\varphi$: **G** est l'opérateur *Globally*. La formule $\mathbf{G}\varphi$ est vraie sur un chemin si la formule φ est vraie pour tous les états du chemin.
- $\psi\mathbf{U}\varphi$: **U** est l'opérateur *Until*. La formule $\psi\mathbf{U}\varphi$ est vraie sur un chemin si
  1. il existe un état du chemin qui satisfait φ et
  2. pour tous les prédécesseurs de cet état ψ est vraie.
- $\psi\mathbf{W}\varphi$: **W** est l'opérateur *Weak until*. La formule $\psi\mathbf{W}\varphi$ est vraie sur un chemin si
  1. il existe un état du chemin qui satisfait φ et
  2. pour tous les prédécesseurs de cet état ψ est vraie.
  3. ou φ n'est jamais satisfait et ψ est toujours vraie.

Les opérateurs **AF**, **EF**, **AG** et **EG** peuvent être exprimés de la façon suivante :

- $\mathbf{AF}(\varphi_1) = \mathbf{A}[\mathbf{true}\,\mathbf{U}\,\varphi_1]$,
- $\mathbf{EF}(\varphi_1) = \mathbf{E}[\mathbf{true}\,\mathbf{U}\,\varphi_1]$,
- $\mathbf{AG}(\varphi_1) = \neg\,\mathbf{EF}(\neg\varphi_1)$,
- $\mathbf{EG}(\varphi_1) = \neg\,\mathbf{AF}(\neg\varphi_1)$.

De façon plus formelle, la syntaxe d'une formule CTL est définie par la définition 1.7

Définition 1.7 *Pour un ensemble donné de propositions atomiques AP, la* logique CTL *est l'ensemble des formules défini de façon inductive par les règles suivantes :*

- *Si $p \in AP$ alors p est une formule CTL.*
- *Si φ_1 et φ_2 sont des formules CTL alors $\neg\varphi_1$, $\varphi_1 \vee \varphi_2$, et $\varphi_1 \wedge \varphi_2$ sont des formules CTL.*
- *Si φ_1 et φ_2 sont des formules CTL alors $\mathbf{AX}\,\varphi_1$, $\mathbf{A}[\varphi_1\,\mathbf{U}\,\varphi_2]$, $\mathbf{A}[\varphi_1\,\mathbf{W}\,\varphi_2]$, $\mathbf{EX}\,\varphi_1$, $\mathbf{E}[\varphi_1\,\mathbf{U}\,\varphi_2]$ et $\mathbf{E}[\varphi_1\,\mathbf{W}\,\varphi_2]$ sont des formules CTL.*

Nous pouvons distinguer deux sous-ensembles de la logique CTL : ACTL est un sous-ensemble de CTL tel que les opérateurs sont uniquement quantifiés par des **A** et les négations ne s'appliquent qu'aux feuilles; ECTL est un sous-ensemble de CTL tel que les opérateurs sont uniquement quantifiés par des **E** et les négations ne s'appliquent qu'aux feuilles.

Exemple

- $\mathbf{AG}(\mathit{Req} \Rightarrow \mathbf{AF}\ \mathit{Ack})$: Si une requête est émise alors un acquittement sera un jour reçu. Cette formule appartient au sous-ensemble ACTL.

– **AG**(**EF** *Reset*) : De n'importe quel état, il est toujours possible d'accéder à un état de ré-initialisation. C'est une formule CTL, elle n'appartient ni à ACTL ni à ECTL.

Dans la suite de ce document, les formules CTL telles que l'ensemble des négations ne s'appliquent qu'aux propositions atomiques sont appelées l'ensemble des formules CTL *positives*.

La sémantique de la logique CTL est définie sur l'arbre d'exécution infini de la structure de Kripke. Soit une structure de Kripke $K : \langle AP, \Sigma, \Sigma_0, \mathcal{L}, R\rangle$. On note $[K, \sigma \models \varphi]$ si la formule φ est satisfaite par $\sigma \in \Sigma$ dans la structure K. Une structure de Kripke K valide une formule φ, dénoté $K \models \varphi$, si et seulement si pour tout $\sigma_0 \in \Sigma_0$, on a $K, \sigma_0 \models \varphi$.

Définition 1.8 *Soit K une structure de Kripke, p une proposition atomique et φ_1 et φ_2 des formules CTL, la relation $\models$ est définie par récurrence de la manière suivante.*

$K, \sigma \models p$	$\Leftrightarrow$	$p \in \mathcal{L}(\sigma)$
$K, \sigma \models \neg\varphi_1$	$\Leftrightarrow$	$K, \sigma \not\models \varphi_1$
$K, \sigma \models \varphi_1 \vee \varphi_2$	$\Leftrightarrow$	$K, \sigma \models \varphi_1$ *ou* $K, \sigma \models \varphi_2$
$K, \sigma \models \mathbf{AX}\varphi_1$	$\Leftrightarrow$	*Pour tout* σ_1 *tel que* $(\sigma, \sigma_1) \in R$, $K, \sigma_1 \models \varphi_1$
$K, \sigma \models \mathbf{EX}\varphi_1$	$\Leftrightarrow$	*Il existe* σ_1 *tel que* $(\sigma, \sigma_1) \in R$, $K, \sigma_1 \models \varphi_1$
$K, \sigma \models \mathbf{A}[\varphi_1 \mathbf{U}\, \varphi_2]$	$\Leftrightarrow$	*Pour tout chemin* $\pi = \sigma_0 \xrightarrow{*} \sigma_k \ldots$ *tel que* $\sigma = \sigma_0$ *il existe un* $k \geq 0$ *tel que* $K, \sigma_k \models \varphi_2$ *et pour tout* $0 \leq i < k$, $K, \sigma_i \models \varphi_1$
$K, \sigma \models \mathbf{E}[\varphi_1 \mathbf{U}\, \varphi_2]$	$\Leftrightarrow$	*Il existe un chemin* $\pi = \sigma_0 \xrightarrow{*} \sigma_k \ldots$ *tel que* $\sigma = \sigma_0$ *il existe un* $k \geq 0$ *tel que* $K, \sigma_k \models \varphi_2$ *et pour tout* $0 \leq i < k$, $K, \sigma_i \models \varphi_1$
$K, \sigma \models \mathbf{A}[\varphi_1 \mathbf{W}\; \varphi_2]$	$\Leftrightarrow$	*Pour tout chemin* $\pi = \sigma_0 \xrightarrow{*} \sigma_k \ldots$ *tel que* $\sigma = \sigma_0$ *et pour tout* $k \geq 0$, *si pour chaque* $i < k$ $K, s_i \not\models \varphi_1$ *alors* $K, s_k \models \varphi_2$
$K, \sigma \models \mathbf{E}[\varphi_1 \mathbf{W}\; \varphi_2]$	$\Leftrightarrow$	*Pour tout chemin* $\pi = \sigma_0 \xrightarrow{*} \sigma_k \ldots$ *tel que* $\sigma = \sigma_0$ *et pour tout* $k \geq 0$, *si pour chaque* $i < k$ $K, s_i \not\models \varphi_1$ *alors* $K, s_k \models \varphi_2$

La vérification de formules CTL a une complexité en temps, déterminée par la taille du modèle et la taille de la formule, de $O(\mid K \mid \times \mid \varphi \mid)$ ([CES86]).

1.3 Vérification de composants et de leur composition

En général, la vérification formelle ne permet pas de vérifier des systèmes trop complexes. Plusieurs techniques permettent de repousser les limites de taille et de complexité des systèmes à vérifier. Une des techniques les plus utilisées est l'abstraction de composant. Cette technique permet d'alléger le système à vérifier en enlevant les détails non pertinents pour la vérification. Pour cela, il faut pouvoir construire une abstraction qui préserve un certain nombre de comportements du système. Nous allons voir qu'il existe de nombreuses relations entre les modèles qui permettent de préserver plus ou moins de comportements du composant.

1.3.1 Relations entre les modèles

Nous rappelons ici les définitions des relations liant des systèmes de transitions. Nous donnons les définitions pour les structures de Kripke mais celles-ci s'appliquent à tous les systèmes de transitions (dont les machines de Moore).

La première relation est la relation de simulation [Mil71]. Intuitivement, K_2 simule K_1 si K_2 peut réaliser toutes les actions de K_1 (la réciproque n'est pas forcément vraie). Cette relation peut-être vue comme la relation de base entre une implémentation et sa spécification (décrite par un système de transitions finis).

Définition 1.9 *Soient K_1 et K_2 deux structures de Kripke avec un même ensemble de proposition atomiques AP. Une relation $H \subseteq \Sigma_1 \times \Sigma_2$ est une* relation de simulation *entre K_1 et K_2 si les conditions suivantes sont vérifiées :*

1. *Pour tout $\sigma_1 \in \Sigma_{0_1}$, il existe $\sigma_2 \in \Sigma_{0_2}$ tel que $H(\sigma_1, \sigma_2)$.*
2. *Pour tout $(\sigma_1, \sigma_2) \in H$,*
   - $\mathcal{L}_1(\sigma_1) = \mathcal{L}_2(\sigma_2)$ *et*
   - *Pour tout σ_1' tel que $\sigma_1 \rightarrow \sigma_1'$ il existe σ_2' $\sigma_2 \rightarrow \sigma_2'$ et $H(\sigma_1', \sigma_2')$.*

Il est possible d'étendre cette relation à K_1 et K_2 avec $AP_1 \subseteq AP_2$ en s'intéressant à $AP_1 \cap AP_2$ uniquement. La relation de simulation est une relation de pré-ordre entre les structures. Si $K_1 \preceq K_2$ alors l'ensemble des propriétés ACTL satisfaites par K_2 sont aussi satisfaites par K_1 [GL91].

Grumberg et Long [GL91] ont montré les relations suivantes pour la composition de structure de Kripke (ils ont par la suite étendu ses résultats aux automates de Moore). Elles sont basées sur la relation de simulation entre des structures de Kripke.

- $\preceq$ est un pré-ordre.
- Pour tout K et K', $K \parallel K' \preceq K$.
- Pour tout K, K' et K'', si $K \preceq K'$ alors $K \parallel K'' \preceq K' \parallel K''$.
- Pour tout K, $K \preceq K \parallel K$.

La seconde relation est la bisimulation forte [Par81].

Définition 1.10 *Soient K_1 et K_2 deux structures de Kripke avec un même ensemble de proposition atomiques AP. Une relation $B \subseteq \Sigma_1 \times \Sigma_2$ est une* relation de bisimulation forte *entre K_1 et K_2 si les conditions suivantes sont vérifiées :*

1. $K_1 \preceq K_2$ *et*
2. $K_2 \preceq K_1$.

Si deux structures sont fortement bisimilaires alors ces deux structures satisfont le même ensemble de formules CTL [BCG88].

La troisième relation est la relation d'équivalence bégayante ([NV95, BCG88]). Deux systèmes K_1 et K_2 sont dit *équivalents bégayants* s'il est possible d'associer à une transition d'un des systèmes, une séquence de transitions de l'autre système, telle que l'ensemble des états de cette séquence sont étiquetés par le même ensemble de propositions atomiques.

Définition 1.11 *Soient K_1 et M_2 deux structures de Kripke avec un même ensemble de proposition atomiques AP. Une relation $B_w \subseteq \Sigma_1 \times \Sigma_2$ est une* relation d'équivalence bégayante *entre K_1 et K_2 si les conditions suivantes sont vérifiées :*

1. *Pour tout* $\sigma_1 \in \Sigma_{0_1}$*, il existe* $\sigma_2 \in \Sigma_{0_2}$ *tel que* $B_w(\sigma_1, \sigma_2)$*.*
2. *Pour tout* $(\sigma_1, \sigma_2) \in H$*,*
   - $\mathcal{L}_1(\sigma_1) = \mathcal{L}_2(\sigma_2)$
   - *Pour tout* σ_1' *tel que* $\sigma_1 \rightarrow \sigma_1'$ *il existe* σ_2' *tel que* $\pi = \sigma_2 \xrightarrow{*} \sigma_2'$ *et* $B_w(\sigma_1', s_2')$*.*
   - *Pour tout* σ_2' *tel que* $\sigma_2 \rightarrow \sigma_2'$ *il existe* σ_1' *tel que* $\pi = \sigma_1 \xrightarrow{*} \sigma_1'$ *et* $B_w(\sigma_1', s_2')$*.*

Si deux structures sont équivalents bégayants alors ces deux structures satisfont le même ensemble de formules CTL privé de l'opérateur *Next* (CTL\X) [BCG88].

1.3.2 Abstraction-Raffinement par contre-exemple (CEGAR)

Un des enjeux majeurs est de rendre automatique la construction d'une abstraction directement à partir d'un code ou d'une description synthétisable. En outre, nous aimerions être capable d'automatiser les phases de raffinement d'une abstraction. En effet, en raison du manque d'information contenue dans le modèle abstrait, il est parfois impossible de statuer sur la validité d'une propriété. Dans ce cas l'abstraction doit être raffinée afin d'obtenir un modèle abstrait plus contraint.

Clarke et al. [COJ+00] proposent une méthode automatique pour générer des abstractions. Cette méthode est itérative, une itération se décompose en trois phases :

1. Abstraction du modèle ;
2. Model checking sur le modèle abstrait ;
3. Raffinement du modèle abstrait.

La première étape construit une abstraction, puis celle-ci est analysée par un model checker, les contre-exemples fournis sont utilisés pour raffiner l'abstraction si cela est nécessaire. Le contre-exemple fourni par le model checker est en général une trace d'exécution du modèle invalidant la propriété. Dans le cas des sur-approximations, le contre-exemple peut représenter un chemin qui n'existe pas dans le modèle concret (*spurious counter-example* ou *faux contre-exemple*). Il faut alors déterminer si le contre-exemple est pertinent : il existe une exécution correspondante dans le modèle concret. Si ce n'est pas le cas il faut raffiner l'abstraction afin de ne pas avoir de telle exécution possible. L'idée est d'extraire les informations du *faux contre-exemple* pour contraindre les exécutions du modèle abstrait.

La méthodologie est présentée figure 1.4. A partir de la propriété globale à vérifier, une première abstraction est générée. Puis par l'analyse de la trace d'exécution produite par le contre-exemple, l'abstraction est raffinée. Le raffinement ajoute des détails à l'abstraction afin de pouvoir statuer sur la validité (ou non) de la formule.

Cette technique est largement utilisée dans le cas du model checking de logiciel. Les outils SLAM [BR02], BLAST [HJMS03], YASM [GC06] ... mettent en oeuvre cette technique combinée avec différents types d'abstraction et d'extraction du raffinement à partir du contre-exemple pour des programmes C ou Java multithread. Plus récemment, l'outil VCEGAR [JKSC07] met en oeuvre l'approche CEGAR pour la vérification de matériel décrit en Verilog, les auteurs ont vérifié des propriétés de sûreté pour les caches instructions du PicoJava II de Sun [JKSC05].

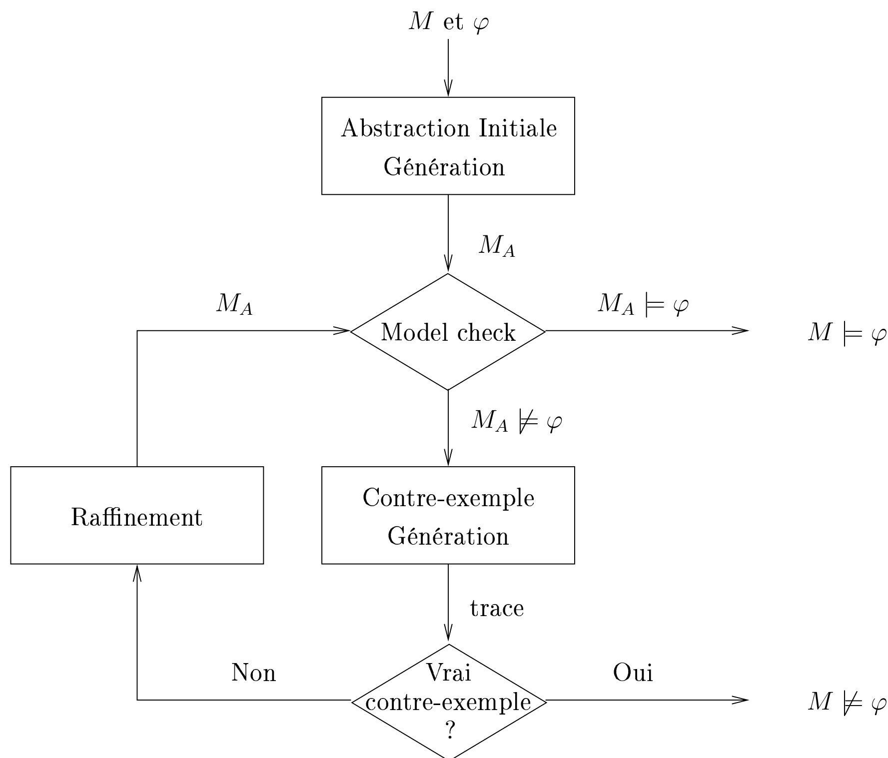


Fig. 1.4 – Boucle CEGAR

1.3.3 Paradigme "assume-guarantee"

La vérification compositionnelle permet d'étendre l'application des méthodes de vérification formelle à des systèmes plus grands et plus complexes. L'idée est de décomposer la vérification du système. Le système peut être décomposé de manière plus ou moins naturelle en composants. Le but est alors de vérifier des propriétés *locales* pour chaque composant individuellement, d'en déduire qu'elles sont aussi valides pour le système complet et enfin de les utiliser pour démontrer des propriétés *globales* du système. L'objectif est de décomposer aussi la spécification en sous-spécifications, si nous pouvons montrer que la vérification de toutes les sous-spécifications impliquent la spécification complète alors nous pouvons conclure que le système complet vérifie sa spécification.

Les SOC offrent une décomposition naturelle en composants puisqu'ils sont conçus par composition de différents IP's. L'objectif est alors de montrer que les comportements de l'ensemble des composants sont corrects vis à vis de la spécification. Les comportements des composants et leurs interactions forment l'ensemble des comportements du système.

Il y a un certain nombre de difficultés à résoudre pour réaliser ce genre de raisonnement. Il faut d'abord s'assurer que les propriétés locales à un composant sont toujours vraies lorsque celui-ci est intégré à un système. Le plus souvent, une propriété locale est vérifiée avec des hypothèses sur le comportement de l'environnement. Ces hypothèses représentent des conditions sur les comportements des autres composants qui doivent être vérifiées afin d'obtenir la vérification du système complet. De plus, nous devons montrer que la conjonction des propriétés locales implique les propriétés globales. Une des méthodes les plus utilisées est le raisonnement "assume guarantee" [McM00, GL91, Lon93, HQR98,

AH99, McM97].

Le raisonnement assume-guarantee se déroule de la façon suivante : pour vérifier une propriété ϕ_1, on suppose que ϕ_2 est vraie, puis pour vérifier ϕ_2, on suppose que ϕ_1 est vraie. Pour montrer que $I_1 \parallel I_2 \preceq P$, Henzinger et al. proposent de contraindre l'environnement des implémentations par une abstraction. Pour réduire encore la taille du modèle à vérifier les contraintes sur l'environnement peuvent être encore abstraites par A_1 et A_2, tel que A_1 modélise la partie de P utile pour contraindre I_2 et de la même façon A_2 modélise la partie de P utile pour contraindre I_1 . Le raisonnement assume-garantee procède alors de la façon suivante :

$$\begin{array}{rcl} I_1 \parallel A_2 & \preceq & A_1 \\ I_2 \parallel A_1 & \preceq & A_2 \\ A_1 \parallel A_2 & \preceq & P \\ \hline I_1 \parallel I_2 & \preceq & A_1 \parallel A_2 \quad \preceq P \end{array}$$

Dans ce chapitre nous avons présenté un ensemble de principes et de techniques utilisés dans la mise en oeuvre de méthodes formelles. A l'aide de ces méthodes nous allons proposer de nouvelles stratégies pour d'une part, l'utilisation plus simple et plus efficace des méthodes formelles lors de la conception de composant matériel. D'autre part nous nous sommes inspirés des techniques d'abstraction pour créer un nouvel algorithme d'abstraction visant à être intégré dans une boucle CEGAR, et permettant ainsi la vérification de systèmes composés.

Première partie

Une Méthode de Conception Incrémentale

Chapitre 2

Conception de composant par ajout

La conception des SOC fait aujourd'hui largement appel à la réutilisation de composants pré-existants. Ainsi il est possible de produire des dispositifs de grande complexité tout en maîtrisant l'effort de conception et de validation. En outre, plus de 60% de la durée de conception est occupée par la vérification et la validation. Il est donc nécessaire que d'une part, chacun des composants soit fiable et adaptable, d'autre part que leur conception et leur validation soient les plus rapides possibles. Puisque l'on souhaite pouvoir ré-utiliser au maximum les composants existants, un concepteur doit être capable de faire évoluer un composant et sa spécification. Mais par souci de rapidité de conception et de rétro-compatibilité, il est préférable que le concepteur n'ait pas tout à redéfinir (ni leur architecture, ni leur spécification). Enfin, l'intégration dans un environnement de vérification à chaque étape de conception devient incontournable. On peut trouver des outils industriels pour la vérification de composants logiciels basés sur l'assemblage de composants (software component based design) comme SLAM de Microsoft [BR02]. La plupart des environnements de conception/vérification formelles sont basés sur des techniques de raffinement comme la méthode B [STE98] ou le model checker SMV [McM00]. Il existe aussi des outils industriels de vérification de composants matériels, la plupart basés sur SMV comme RuleBase de IBM [BBDEL96] ou FormalCheck de Cadence [Cad99], qui incluent la dimension compositionnelle et l'adaptabilité des composants. Les techniques d'intégration de services visent plus l'évolution des composants et fournissent eux-aussi un environnement de conception et de vérification à chaque étape de développement du composant. Nous proposons une méthode de conception proche du concepteur qui permet de faire le lien entre la vérification et la conception.

Ce chapitre présente une méthode de conception de composants matériels par ajout successifs de nouvelles fonctionnalités. Cette méthode décompose la conception d'un composant en différentes étapes successives.

1. Le concepteur se donne un composant initial réalisant les comportements de base.
2. Le concepteur ajoute au composant de nouveaux événements et leur traitement correspondant un à un.

Le composant initial est un modèle simple qui ne prend pas en compte tous les comportements possibles de l'environnement. Ensuite, le concepteur enrichit le composant petit à petit, en ne considérant qu'un seul (ou un petit nombre) d'événement à chaque étape, afin d'obtenir le composant final.

Nous avons voulu rester très proche de la façon dont les concepteurs de matériel travaillent. C'est pourquoi la représentation des composants utilisée est un automate de Moore auquel nous allons ajouter des fonctionnalités afin d'obtenir une nouvelle machine de Moore plus complexe, jusqu'à obtenir l'implémentation complète du composant.

Le point essentiel de cette méthode est la *décomposition* de la conception. Le système est obtenu de manière incrémentale. A chaque nouvelle étape, le système est incrémenté pour devenir de plus en plus complexe. La première chose à définir est la nature de l'incrément. Cela revient à répondre aux questions suivantes :

- Que représente-t-il ? (Un nouveau traitement)
- Quand survient-il ? (Une nouvelle configuration des signaux d'entrées)
- Comment est-il représenté ? (Une suite de traitements élémentaires)
- Comment est-il relié au composant existant ? (Un mode de connexion adéquat)

Ensuite, nous pourrons déterminer si tous les ajouts peuvent être modélisés sous forme d'incrément :

- Existe-t-il des traitements incompatibles ?
- Existe-t-il un "bon" ordre d'ajout des incréments ?

La suite de ce chapitre apporte une réponse à chacune de ces questions, nous commençons par définir notre méthode générale de conception et nous la situons par rapport aux autres techniques de conception par ajout que l'on peut trouver dans la littérature. Puis, nous définissons les types d'incréments qui interviennent dans notre démarche de conception et nous montrons comment les composer pour obtenir une implémentation réelle.

2.1 Idée générale

2.1.1 La conception incrémentale

La méthode de conception que nous proposons se fait par ajouts successifs de nouveaux comportements, qui viennent enrichir un composant initial. Nous l'appelons *la méthode incrémentale*. L'objectif de cette méthode est d'une part de simplifier le travail de conception d'un composant en fournissant un cadre de conception. D'autre part, elle favorise l'ajout de nouveaux services de façon simple et uniforme, dans le cadre de l'évolution d'un composant. Enfin, nous montrerons dans le chapitre suivant que cette méthode facilite la démarche de vérification d'un composant.

Un aspect primordial de notre démarche est l'association de la conception d'un composant avec sa vérification formelle. En suivant cette démarche, il est possible de dériver la spécification du composant simple afin de l'intégrer dans la spécification du composant plus complexe. Pour un concepteur d'architecture matérielle, la méthode incrémentale est naturelle et simple à suivre. Bien que l'écriture d'une spécification ne soit pas une tâche aisée, l'utilisation de notre méthode permet d'en intégrer une partie automatiquement, et donc d'alléger le travail d'écriture de la spécification. De la même façon qu'un concepteur ne veut pas récrire des parties d'une architecture lors de l'évolution du composant, celui-ci ne veut pas avoir à récrire entièrement la spécification du composant à chaque étape de la conception. Ce qui a déjà été vérifié pour un composant à une étape i de développement reste vrai aux étapes $i + k$, si les règles d'ajout de nouveaux comportements sont respectées.

Les ajouts sont appelés des *incréments*. Ce chapitre introduit donc cette notion d'incrément et montre quels sont les différents types d'incrément que l'on peut ajouter à un composant initial sans détruire les comportements existants. La figure 2.1 donne l'idée générale de la conception incrémentale. La première implémentation 0 est incrémentée par l'incrément *Inc* 1 afin d'obtenir une implémentation 1 qui contient l'implémentation précédente avec des fonctionnalités supplémentaires. Ensuite, l'implémentation 1 est elle-même incrémentée et ainsi de suite, jusqu'à obtenir l'implémentation la plus complexe qui représente l'architecture complète du composant.

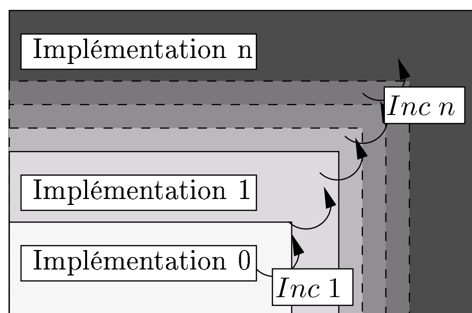


Fig. 2.1 – La méthode de conception incrémentale

2.1.2 Les méthodes par raffinement

Notre méthode est complémentaire aux stratégies de raffinement comme la *méthode B* [Lan96] ou le *refinement checking* de McMillan [McM00]. Le développement d'application matérielle ou logicielle à l'aide de la méthode B est basé sur cette technique de raffinement. Le raffinement est utilisé pour transformer une spécification abstraite en une spécification concrète qui pourra être automatiquement transformée en code exécutable [B-C97] ou en code synthétisable [MBA02] implémenté dans l'outil BHDL [ADT+03]. La méthode B est décrite figure 2.2. La vérification lors de la conception d'un composant se fait à l'aide d'un assistant de preuve. Le concepteur commence par décrire une *machine abstraite* qui est une traduction de la spécification. Cette machine définit les variables et les constantes sous la forme d'ensembles, et elle décrit les opérations qui manipulent les variables sous la forme de relations. Enfin elle définit les invariants qui permettent d'énoncer les preuves des opérations et des variables. Ce sont les propriétés vérifiées par la machine abstraite, elles représentent les attentes du système.

Le raffinement sert alors, d'une part à passer de la structure de données abstraite (ensemble) à des structures de données concrètes (entiers, tableaux...), d'autre part à adapter les opérations aux nouvelles structures de données, mais aussi à rendre déterministe les algorithmes. Au fur et à mesure du raffinement, le concepteur fait des choix sur les structures et sur les algorithmes utilisés. Dans une première étape un algorithme peut avoir certaines opérations non déterministes, puis plus le concepteur se rapproche de l'implémentation, plus il précise les différents comportements. De nouveaux invariants sont ajoutés pour assurer le lien entre les différentes données et permettre de prouver que la nouvelle machine vérifie la spécification. Les preuves sont réalisées à l'aide d'un

assistant de preuves. Le raffinement se fait ici *sans changer la signature (ou en restreignant la signature) des opérations ni ajouter de nouvelles opérations.* Les modifications d'une opération ne doivent pas être perceptibles par un utilisateur de la machine. Tous les comportements possibles sont définis dans la première machine.

La méthode B a été utilisée avec succès dans des applications industrielles comme le pilote automatique de METEOR [BBFM99] et plus récemment le Roissy Val [BA05].

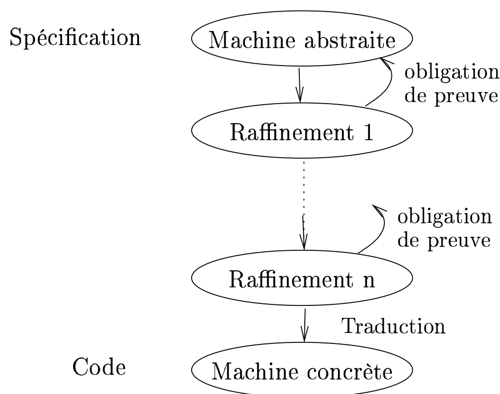


FIG. 2.2 – Méthode de conception à la B

La différence de la méthode B avec des méthodes de refinement checking (McMillan [McM00]) est l'utilisation d'un model checker plutôt que d'un assistant de preuves à chaque étape de la conception. Le but est aussi de vérifier qu'un modèle abstrait, se comportant comme la spécification du système, est implémenté par un modèle plus détaillé. L'idée est de réduire la taille du système en sous problèmes qui peuvent être vérifiés par model checking. L'implémentation est définie en terme de relations de raffinement, ces relations permettent de décomposer le problème de vérification en plus petites parties pour une vérification séparée et localisée. L'implémentation, la machine abstraite et les relations de raffinement sont décrites en HDL (hardware description langage).

Comme dans le cas de la méthode B, le raffinement est plus une *spécialisation* du comportement. Le concepteur décrit de manière abstraite le système puis spécialise le traitement à effectuer en unités fonctionnelles reliées au modèle abstrait par les relations de raffinements. Chaque unité peut être vérifiée séparément en utilisant les opérations abstraites à la place des autres unités fonctionnelles. La difficulté de cette méthode se trouve dans l'écriture des relations de raffinement. Il faut pouvoir trouver des points d'observation judicieux pour comparer une machine abstraite, à une implémentation. Souvent les opérations abstraites et concrètes ont des délais différents, ou les types des opérateurs ne sont pas les mêmes.

La méthode de *refinement checking* est incluse dans l'outil SMV de Cadence. Elle a servi à implémenter et vérifier des systèmes tel qu'un microprocesseur [JM01] et l'algorithme de Tomasulo [McM98] utilisé pour le réordonnancement des instructions dans un processeur superscalaire.

La force de ces méthodes réside dans la préservation des propriétés générales du système le long du processus de conception : si une propriété est vraie sur un modèle, si le raffinement est bien défini alors le modèle raffiné préserve cette propriété. L'inconvénient

majeur de ces méthodes est que tous ce que l'on ajoute au modèle initial ne doit pas induire de nouveaux comportements. Tout ce que l'on peut ajouter n'est qu'une spécialisation de comportements déjà définis.

2.1.3 Les méthodes d'intégrations de nouveaux services

L'interaction entre différents incréments a été largement étudiée dans le contexte de la détection d'incohérences lors de l'intégration de nouveaux services dans des applications de télécommunication ou dans des applications logicielles. Plath et Ryan [PR99] ont réalisé un outil permettant l'intégration automatique de nouveaux services. Cette outil est couplé à un model checker ([PR01] pour SPIN [Hol97], [PR99] pour SMV [McM93], [CRS01] pour MOCHA [AHM+98]). Il permet de vérifier *a posteriori* la cohérence (ou l'incohérence) des fonctionnalités d'un système conçu par intégration automatique de nouveaux services.

L'idée vient de l'analyse du développement des nouveaux systèmes téléphoniques. De plus en plus d'options comme le transfert d'appel, le rappel automatique ou encore les e-mail vocaux sont à la disposition de l'utilisateur. Mais l'intégration de ces nouveaux services peut ne pas fonctionner correctement.

Exemple 1 :	Exemple 2 :
(Téléphone + Rappel automatique) + Transfert d'appel	(Téléphone + Transfert d'appel si occupé) + Boîte vocale si occupé
Bonne interaction	Mauvaise interaction

TAB. 2.1 – Exemple d'interactions de services

Le tableau 2.1 montre deux exemples d'ajouts de services. Dans le premier exemple, le système de base est composé d'un téléphone et du service de rappel automatique. Le rappel automatique consiste à être rappelé dès que notre correspondant est disponible. À ce système de base est ajouté le service de transfert d'appel, Tous les appels entrants sont transférés sur un autre téléphone. Dans ce cas l'intégration de ce nouveau service fonctionne correctement : X appelle Y qui n'est pas disponible puis transfère ses appels sur Z. Lorsque Y est disponible l'appel sera transmis sur Z. Le deuxième exemple montre une interaction incompatible des différents ajouts. Un utilisateur s'abonne au service de transfert à une boîte vocale si il est indisponible. Puis, il rajoute le service de transfert d'appel sur un autre téléphone, si il est indisponible. Dans ce cas l'intégration de ces deux nouveaux services provoquent une incohérence dans le système : lorsque la personne n'est pas disponible le système exécute les deux actions différentes ce qui entraîne un conflit des deux services. La méthode proposée par Plath et Ryan permet de détecter les différents conflits qui peuvent survenir (ou non) lors de l'ajout de services.

L'idée générale de cette approche est dans un premier temps de décrire le modèle de base avec sa spécification écrite en CTL, LTL, ATL (Alternating Time Logic [AHK97]), puis de décrire les nouveaux services comme des unités fonctionnelles avec leur propre spécification, sans hypothèse sur le système auquel ils vont s'intégrer. L'intégration se fait automatiquement et le système résultant est lui vérifié. Les incohérences entre les différents

ajouts de service sont détectées par la violation de propriétés du système de base et des différents services. La vérification se fait *a posteriori* de l'intégration. Lorsqu'un ajout est réalisé, il n'y aucune garantie sur le fonctionnement du système enrichi ni sur le fait que ce qui marchait avant l'intégration d'un nouveau service continuera à marcher après l'ajout effectif. Leur méthode ne garantit pas la non-régression du composant au cours des ajouts successifs.

D'autres comme Cansell et Méry [CM01] ont pour leur part intégré la détection d'incohérence dans l'Atelier B [STE98]. L'implémentation est comparée à un système abstrait qui inclut les nouveaux services. La démonstration d'un assemblage de services incorrect se fait aussi à posteriori et est déterminée par la non prouvabilité des obligations de preuves. Leur but est plus de comprendre quand un service interfère avec un autre.

Notre approche est différente, d'une part car notre incrément est plus simple : il doit respecter nos règles de conception. D'autre part les services ajoutés ne sont pas complètement indépendants du système de base. D'abord parce qu'ils doivent garantir l'accès à tout ce qui était existant. En outre, nos incréments représentent la gestion de nouveaux événements de l'environnement qui compliquent le traitement existant plutôt que l'ajout de nouvelles fonctionnalités indépendantes.

2.2 Définition d'un incrément

Notre incrément diffère de ceux proposés par les méthodes de raffinement, dans le sens où il introduit des comportements qui n'existaient pas dans le modèle initial. De plus il est beaucoup plus simple que ceux proposés dans l'intégration de services. En effet, notre incrément est *monotone* : il n'existe pas de destruction de comportements, tous les comportements qui existent dans le cas simple sont préservés dans le composant plus complexe. En outre, l'incrément que nous proposons permet de décrire de façon uniforme l'ajout de comportements. Nous fixons un cadre pour faire évoluer un système lors de sa conception. Ce cadre assure l'absence d'incohérences lors des ajouts, il garantit la non-régression d'un composant par construction.

Un composant est défini par une partie contrôle qui dirige un chemin de donnée. Son espace d'états est représenté par un automate de Moore complet et déterministe (cf. définition 1.3). En suivant notre démarche de conception, le concepteur commence par définir l'automate de Moore de la manière la plus simple possible. La première version du composant est le composant initial qu'il va ensuite enrichir en ajoutant des nouveaux comportements définis sous forme d'incréments.

Un incrément est un ensemble de modifications appliquées à un composant dans le but d'obtenir un composant plus complexe. Il représente l'apparition d'un nouvel événement sur l'interface d'un composant. Les comportements ajoutés sont donc la gestion du nouvel événement qui n'était pas pris en compte dans le cas le plus simple. Le nouvel événement entraîne une modification du comportement et peut induire des modifications sur l'interface de sortie mais il *préserve* tous les comportements existants dans le modèle plus simple. L'exemple figure 2.3 montre un modèle à une étape i du développement, l'interface d'entrée est composée des signaux a et b et l'interface de sortie contient le signal α_0. Le modèle incrémenté à l'étape $i+1$ considère un nouvel événement qui s'exprime

par l'intermédiaire du nouveau signal d'interface c et qui induit un nouveau signal de sortie α_1.

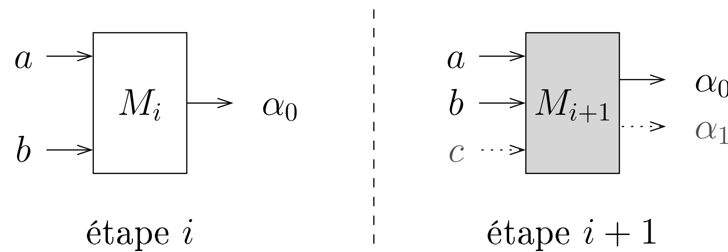


FIG. 2.3 – Composant M_{i+1} obtenu par incrément à partir de M_i

Le modèle M_{i+1} contient tous les comportements du modèle M_i plus les comportements réalisant la gestion du nouvel événement. Si le nouvel événement n'est pas activé, le composant M_{i+1} se comporte exactement comme M_i. Il existe une certaine configuration des signaux d'entrée de M_{i+1}, qui détermine si le nouvel événement est activé. Dans ce cas, le modèle M_{i+1} ne se comporte plus comme M_i, le nouveau comportement correspond au traitement du nouvel événement. La nouvelle configuration des entrées peut être obtenue de deux façons différentes :

- Le domaine de définition d'un signal est étendu. L'interface d'entrée d'un composant reste fixe mais de nouvelles valeurs de signaux peuvent être prises en compte. C'est le cas, par exemple, lorsque l'on étend un signal d'acquittement avec des valeurs d'erreurs.
- Un ou plusieurs signaux sont ajoutés (avec leur propre domaine de définition). C'est le cas, par exemple, lorsque l'on complexifie le chemin de données.

Dans les deux cas, le nouvel événement peut être modélisé par l'ajout d'un nouveau signal avec son propre domaine de définition. Par exemple, on souhaite étendre le domaine de définition d'un signal a. Dans le cas le plus simple, le domaine de définition de a est l'ensemble $\{i, j\}$. Le composant plus complexe peut gérer une nouvelle valeur k pour le signal a. Cela peut s'exprimer par l'ajout d'un nouveau signal b tel que son domaine est l'ensemble $\{0, 1\}$. La valeur 0 exprime le fait que le signal a peut prendre une valeur du modèle simple (i ou j), la valeur 1 quand à elle exprime le fait que le signal a prend la nouvelle valeur k. Le domaine de a reste inchangé. Le nouveau signal b n'est pas introduit dans l'implémentation du composant, c'est une modélisation de l'incrément qui est cohérente pour les deux cas de figure.

La conception incrémentale impose, dans le cas où l'événement n'arrive pas, que le composant M_{i+1} se comporte exactement comme M_i. Il faut pouvoir distinguer le cas où l'événement est activé ou non. Pour cela, nous avons défini deux configurations particulières des signaux d'entrée :

- Une valeur *muette* : l'événement ne s'exprime pas. Le composant se comporte comme le composant plus simple.
- Une valeur *active* : l'événement est actif. Le composant prend en compte cet événement et réalise les nouveaux traitements associés.

Lorsque l'interface d'entrée est à une configuration *active* alors le composant passe dans les nouveaux comportements. Grâce à cette valeur particulière nous pouvons repérer l'instant

où le composant *quitte* l'ensemble des anciens comportements. Tant que l'événement n'est pas actif (configuration muette de l'interface), M_{i+1} est dans les comportements de M_i, dès que l'événement est actif (configuration active de l'interface), M_{i+1} est dans les nouveaux comportements.

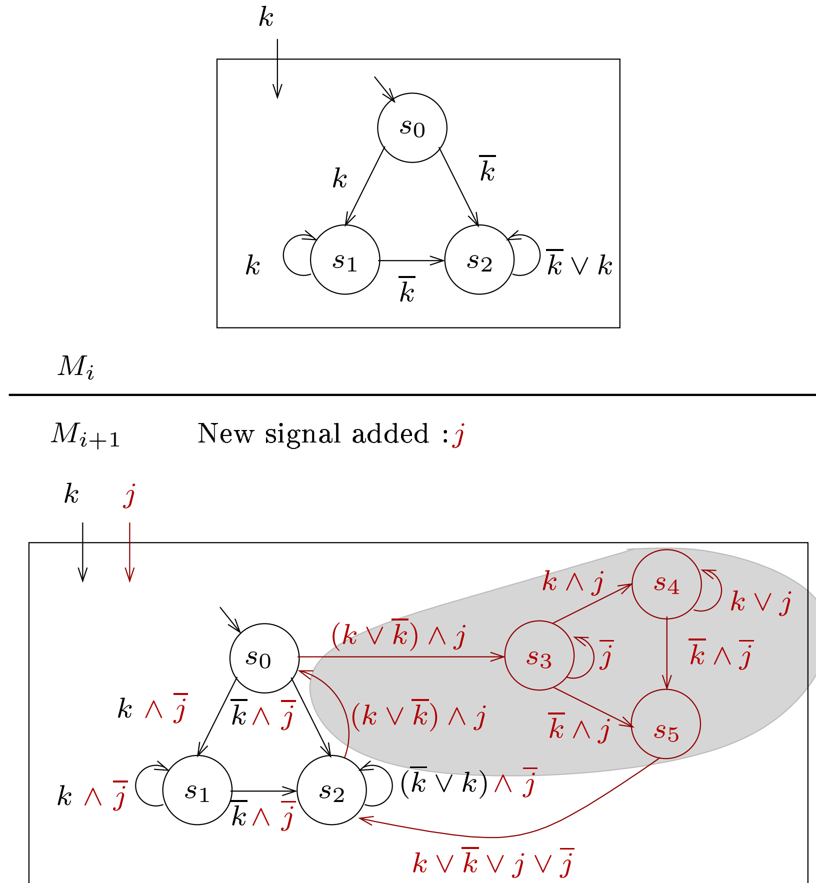


Fig. 2.4 – Exemple d'incrément valide

La figure 2.4 représente un incrément possible. L'automate de Moore en haut décrit un composant M_i à l'étape i de conception. Il contient trois états et un signal d'entrée k tel que $\mathcal{D}om(k) = \{0, 1\}$. Sur la figure, les littéraux k et $\overline{k}$ représentent respectivement le positionnement du signal k à la valeur 1 et 0. Le composant M_{i+1} en bas est obtenu à partir du composant M_i auquel nous avons ajouté un incrément. Le nouvel événement s'exprime par l'intermédiaire du nouveau signal j tel que $\mathcal{D}om(j) = \{0, 1\}$. Le littéral $\overline{j}$ représente une configuration muette et le littéral j une configuration d'entrée active. On peut remarquer que dans l'automate M_{i+1} toutes les transitions qui existaient dans M_i sont étiquetées par la configuration muette du nouvel événement, c'est la préservation des comportements existants. A partir de l'état s_0 de M_{i+1}, il existe une transition étiquetée par la valeur active du signal j. Cette transition amène sur des nouveaux états et de nouvelles transitions, qui représentent la gestion du nouvel événement. L'ensemble des transitions étiquetées par la valeur active de l'événement quittant M_i pour aller dans l'ensemble des nouveaux comportements représente une *frontière* entre les anciens et les nouveaux comportements. A partir des états préexistants, il ne peut pas exister une tran-

sition étiquetée par une configuration muette et faisant partie de cette *frontière*. Les nouvelles transitions (dans la partie grisée), peuvent quand à eux être étiquetés par n'importe quelle configuration des signaux d'entrée. A partir de ces comportements, on peut revenir sur les anciens mais ce n'est pas toujours le cas.

Plus formellement, un composant est modélisé par un automate de Moore $M = \langle S, I, O, T, L, s_0 \rangle$ (cf. def. 1.1), un incrément ajoute de nouveaux signaux d'entrée et peut aussi ajouter des nouveaux états, des nouvelles transitions, et des nouveaux signaux de sortie. Les valeurs spéciales (muette et active) sont des ensembles de configuration des signaux d'entrée du composants (def. 1.2). L'ensemble des configurations de signaux d'entrée du nouvel événement est décomposé en deux sous-ensembles disjoints $\mathcal{C}_{QT}$ et $\mathcal{C}_{ACT}$. Ils représentent respectivement l'ensemble des configurations muettes et actives.

Définition 2.1 *Un* événement e *est un triplet* $e = \langle I_+, C_{ACT}(I_+), C_{QT}(I_+) \rangle$ *tel que :*

I_+ *= L'ensemble de nouveaux signaux et leur domaine de définition ;* $I \cap I_+ = \emptyset$.

$C_{ACT}(I_+) \subset \mathcal{C}(I_+)$ *: L'ensemble des configurations où le nouvel événement s'exprime. Si l'ensemble des nouveaux signaux est positionné avec l'une de ces configurations, l'événement est dit actif.*

$C_{QT}(I_+) \subset \mathcal{C}(I_+)$ *: L'ensemble des configurations où le nouvel événement ne s'exprime pas. Si l'ensemble des nouveaux signaux est positionné avec l'une de ces configurations, l'événement est dit muet.*

Le nouvel événement peut induire de nouveaux signaux de sortie. De la même manière que les signaux d'entrées, ils possèdent une configuration muette et une configuration active.

Définition 2.2 *Soit* O_+*, l'ensemble des nouveaux signaux de sortie avec leur domaine de définition :*

$C_{ACT}(O_+)$ *: L'ensemble des configurations représentant l'activation des sorties.*

$C_{QT}(O_+)$ *: L'ensemble des configurations représentant l'inactivation des sorties.*

Remarque 2 On a $C_{ACT}(I_+) \cup C_{QT}(I_+) = \mathcal{C}(I_+)$ et $C_{ACT}(I_+) \cap C_{QT}(I_+) = \emptyset$ et
On a $C_{ACT}(O_+) \cup C_{QT}(O_+) = \mathcal{C}(O_+)$ et $C_{ACT}(O_+) \cap C_{QT}(O_+) = \emptyset$.

Notation e_qt est une configuration des signaux de I_+ appartenant à $C_{QT}(I_+)$. e_act est une configuration des signaux de I_+ appartenant à $C_{ACT}(I_+)$. Nous écrirons de manière abusive $\neg(e_qt) \in C_{ACT}(I_+)$ et $\neg(e_act) \in C_{QT}(I_+)$

Pour la définition de l'incrément nous avons besoin de définir au préalable une fonction de projection sur les ensembles de signaux.

Définition 2.3 *Soient deux ensembles de signaux* E_1 *et* E_2 *tel que* $E_2 \subset E_1$*, et soit* c *une configuration appartenant à* $\mathcal{C}(E_1)$*,* $c' = proj(c, E_2)$ *est la sous configuration de* c *restreinte aux signaux de* E_2.

Définition 2.4 Définition de l'incrément
Soit M_i *un automate de Moore, un incrément qui s'applique à* M_i *selon le nouvel événement* $e = \langle I_+, \mathcal{C}_{ACT}(I_+), \mathcal{C}_{QT}(I_+) \rangle$ *et l'ensemble des nouveaux signaux de sortie* O_+*, l'incrément est un 4-uplet* $INC = \langle M_{INC}, T_{i \to INC}, T_{INC \to i}, T_{i \to i} \rangle$.

- *M_{INC} est un automate de Moore tel que :*

  S_{INC} : l'ensemble des nouveaux états, $S_{INC} \cap S_i = \emptyset$;

  $I_{INC} = I_i \cup I_+$: l'ensemble des signaux d'entrée ;

  $O_{INC} = O_i \cup O_+$: l'ensemble des signaux de sortie ;

  $T_{INC} = (S_{INC} \times C(I_i \cup I_+) \times S_{INC})$: L'ensemble des nouvelles transitions reliant les nouveaux états accessibles.

  $\mathcal{G}_{INC} = \{l_0, \dots, l_{|O_i \cup O_+|-1}\}$: Vecteurs de fonctions de génération tel que pour tous les signaux de sortie $o_j \in O_+$ et pour tous $s \in S_i$, $g_j(s) \in \mathcal{C}_{QT}(o_j)$;

  $S_{0_{INC}} \subseteq S_{INC}$: l'ensemble des états initiaux.

- *La connexion entre M_i et M_{INC} est telle que :*

  $T_{i \to INC} \subseteq (S_i \times C(I_i \cup I_+) \times S_{INC})$: l'ensemble des transitions de M_i à M_{INC} tel que $(s_1, c, s_2) \in T_{i \to INC}$ ssi $proj(c, I_+) \in \mathcal{C}_{ACT}(i_+)$

  $T_{INC \to i} \subseteq (S_{INC} \times C(I_i \cup I_+) \times S_i)$

- *$T_{i \to i} = (S_i \times \mathcal{C}(I_i \cup I_+) \times S_i)$: l'ensemble des nouvelles transitions reliant des états de S_i tel que $(s_1, c, s_2) \in T_{i \to i}$ ssi $proj(c, I_+) \in \mathcal{C}_{ACT}(i_+)$.*

Toutes les transitions sortantes d'un état appartenant aux anciens comportements qui n'existaient pas dans M_i sont étiquetés par la valeur active de l'événement. Pour toute transition $(s_1, c', s_2) \in T_{INC}$ telle que $s_1 \in S_i$ et $s_2 \in S_{INC} \cup S_i$, on a $proj(c', I_+) \in C_{ACT}(I_+)$. Par la suite nous noterons $c' = c \wedge e_act$, avec $c \in \mathcal{C}(I_i)$ and $e_act \in C_{ACT}(I_+)$, toutes les transitions sortantes d'un état de S_i étiquetées par e_act. C'est grâce à cette étiquette que nous sommes capables de repérer dans l'automate de Moore les transitions qui amènent dans les nouveaux comportements. Toutes ces transitions dessinent une frontière entre les anciens et les nouveaux comportements (fig.2.5). A partir des nouveaux comportements il peut exister des transitions qui retournent aux anciens comportements.

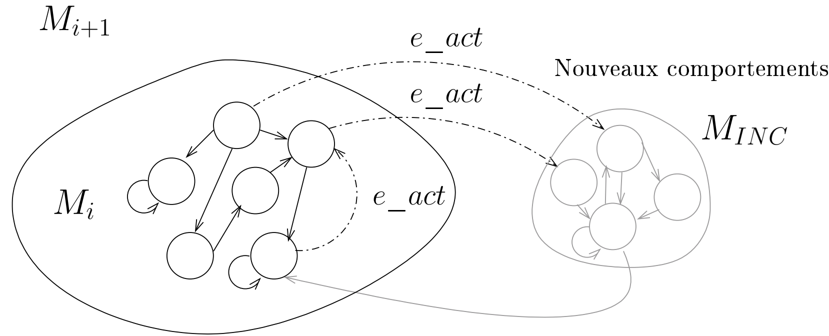


FIG. 2.5 – Frontière e_act

Dans cette section nous avons défini un incrément comme l'ensemble des nouveaux signaux, états et transitions. Cela correspond à une définition très générale d'un incrément, qui ne prend ni en compte la structure du modèle auquel il s'applique, ni même sa propre structure. Dans la suite de ce chapitre, nous allons caractériser l'incrément pour tirer partie de sa structure.

2.3 Spécification CTL de l'ajout

Dans la section précédente, nous avons présenté le cadre général de la conception incrémentale. Ce cadre permet au concepteur d'architecture matérielle de structurer les étapes de conception d'un composant, en garantissant la non-régression des comportements définis étape par étape. Un incrément caractérise les nouveaux comportements utiles à la gestion d'un nouvel événement qui représentent des cas particuliers de fonctionnement d'un composant. Dans de nombreux cas, il est plus simple de raisonner sur l'incrément en tant que module indépendant ne s'appuyant sur aucun des comportements préexistants. Par exemple, lorsque l'on ajoute un mécanisme de traitement des exceptions dans un microprocesseur pipeline, il est facile de concevoir les différentes phases (vider les instructions entrées à tort, positionner les registres d'état, brancher les instructions du gestionnaire d'exceptions ...) sans se soucier du fonctionnement général. Le module peut alors être ajouté au composant et comme nous l'avons expliqué précédemment, il est toujours possible de repérer les nouveaux comportements par le franchissement d'une transition étiquetée par une configuration active de l'événement.

De la même manière, nous voudrions déterminer une valeur particulière pour représenter le *retour* après que la nouvelle fonctionnalité ait achevé son traitement. Nous avons identifié trois types de retours possibles :

1. Sans retour possible sur des états de la structure initiale (figure 2.6(a)) ;
2. Avec une valeur de retour particulière qui marque la fin du traitement de l'événement (figure 2.6(b)) ;
3. Sans condition particulière de retour sur des états de la structure initiale (figure 2.6(c)).

Contrairement à la valeur active d'un événement, la valeur de retour n'est pas uniquement déterminée par une configuration des signaux d'entrée. En effet, la fin d'un traitement peut être défini par une configuration finale des signaux de sortie, le retour à une configuration muette des signaux d'entrée, ou même par une configuration de signaux internes. La valeur de *retour* est définie sur l'ensemble des signaux du composant. Soient E l'ensemble des signaux d'un composant, $\mathcal{C}_{RTN}(E) \subset \mathcal{C}(E)$, et nous notons e_rtn une configuration qui appartient à $\mathcal{C}_{RTN}(E)$ et $\overline{e_rtn}$ une configuration qui ne fait pas partie de $\mathcal{C}_{RTN}(E)$.

Nous avons mis en évidence trois types d'incréments particuliers.

Définition 2.5 Incrément particulier
Soient M_i un automate de Moore, un nouvel événement $e = \langle I_+, \mathcal{C}_{ACT}(I_+), \mathcal{C}_{QT}(I_+)\rangle$, et un incrément $INC = \langle M_{INC}, T_{i\to INC}, T_{INC\to i}, T_{i\to i}\rangle$, tel que :

- *$T_{i\to i} = \emptyset$ pour tout (pas de transitions de l'incrément qui ne fait pas intervenir de nouveaux états) ;*
- *$T_{i\to INC} \subseteq (S_i \times C(I_i \cup I_+) \times S_{0_{INC}})$ (L'ensemble des transitions de M_i à M_{INC} accèdent aux états initiaux de M_{INC}).*

INC est de type :

NORTN *: (sans retour) ssi $T_{INC\to i} = \emptyset$*

VALR *: (avec retour étiqueté) ssi $\forall (s, c, s') \in T_{INC\to i}$, $c \in \mathcal{C}_{RTN}$.*

NOVALR *: (avec retour, sans valeur particulière) ssi $\mathcal{C}_{RTN} = \emptyset$*

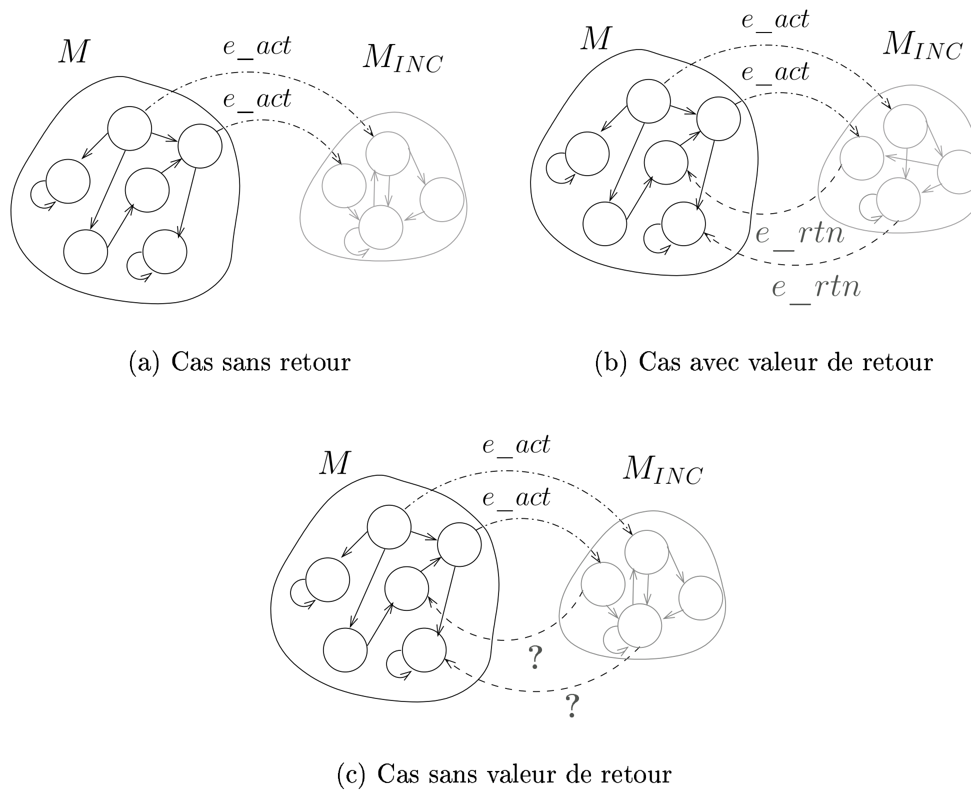


Fig. 2.6 – Les trois types d'incréments concernés par l'extension

Nous verrons dans le chapitre suivant, comment obtenir une partie de la spécification d'un composant plus complexe à partir de la spécification du composant plus simple. A partir des incréments particuliers que nous venons de définir il est maintenant possible de spécifier l'incrément. Le concepteur pourra maintenant, à partir de la spécification du composant simple et de l'incrément obtenir automatiquement la spécification complète du composant complexe. L'écriture de cette spécification est simple car le concepteur se concentre sur un ensemble de traitements pour un unique événement (ou petits nombres d'événements). Cette spécification de l'incrément caractérise seulement ce que l'on ajoute. En effet, lorsque le concepteur veut ajouter la gestion du nouvel événement, il souhaite spécifier uniquement ce nouveau comportement sans avoir à imaginer comment celui-ci va s'insérer dans le modèle initial. L'incrément est vu comme une structure à part qui viendra ensuite enrichir le composant.

Nous pouvons maintenant ajouter une spécification $SPEC_{INC}$ en logique CTL aux automates de l'incrément. $SPEC_{INC}$ est l'ensemble des propriétés CTL vérifiées par l'ensemble des états initiaux de M_{INC}.

Définition 2.6 *La* spécification de l'incrément $SPEC_{INC}$ *est telle que :* $SPEC_{INC} = \{\varphi \mid \forall s_0 \in S_{0_{INC}}, M_{INC}, s_0 \models \varphi\}$

2.4 Peut-on tout modéliser sous forme d'incrément ?

Un incrément est la modélisation de la gestion de nouveaux événements ajoutés sur l'interface d'un composant. Les incréments que nous avons définis enrichissent les comportements et imposent de conserver l'ensemble des comportements préexistants. La conception incrémentale assure la non-régression d'un composant : un problème de conception est résolu une fois pour toutes, nos incréments conservent l'existant. Par conséquent, tous les ajouts de services qui ne permettent pas d'accéder aux anciens comportements ou qui détruisent des comportements ne sont pas des incréments valides dans le cadre de notre conception. Notre méthode de conception exclut, par exemple, les cas d'ajout d'une phase d'initialisation d'un composant, si celle-ci n'a pas été prévue au départ.

L'objectif de cette méthode est d'avancer pas à pas, d'une architecture simple jusqu'à une implémentation contenant tous les comportements possibles de l'architecture. Pour une utilisation efficace de notre méthode, il est donc nécessaire de découper les différents événements à traiter pour un composant. La décomposition des événements est pour certaines architectures assez naturelle. Par exemple pour les traducteurs de protocoles de communication les transactions requêtes/réponses facilitent la décomposition. Les incréments peuvent alors se concevoir de manière indépendante. Une bonne décomposition permet de structurer la conception. Si un ajout ne peut pas être incorporé alors soit un incrément précédent est mal défini soit la décomposition n'est pas correcte.

Le découpage en événements distincts, n'est pas toujours un travail évident. Il faut pouvoir délimiter le champ d'action d'un événement et de l'incrément qu'il induit. De plus, pour respecter notre méthodologie, l'ajout d'incréments ne doit pas détruire les ajouts antérieurs, les incréments doivent être compatibles entre eux. Le traitement *d'incréments incompatibles* est tout de même possible. Il suffit de construire un incrément qui est la composition d'incréments incompatibles afin d'obtenir un incrément indépendant.

Une autre caractéristique de la conception incrémentale est la préemption des ajouts. En effet, l'ordre des ajouts définit aussi un ordre de priorité sur les incréments. Soit un composant obtenu par addition successive d'incréments : $C_1 \xrightarrow{Inc_1} C_2 \xrightarrow{Inc_2} C_3$. Si à un instant donné les deux événements sont actifs en même temps alors par construction l'événement 2 masque l'événement 1. Cette caractéristique est restrictive mais elle garantit *par construction* la non-régression d'un composant tout au long de sa conception.

Un des inconvénients est que les incréments ne peuvent pas être définis une fois pour toute pour n'importe quel composant à implémenter. En effet, la description des incréments dépend souvent du modèle sur lequel il s'applique mais aussi de l'étape de conception du composant. L'avantage est que les incréments ne dépendent que de ce qui a déjà été intégré et non pas de futurs ajouts. Ce qui reste cohérent dans le cadre de l'évolution de composant.

En résumé, la démarche incrémentale permet de concevoir un système petit à petit, en se focalisant sur une difficulté différente à chaque étape. Cette méthode garantit la non-régression des comportements tout le long de la phase de conception. De plus, les incréments sont une modélisation simple des différents ajouts que l'on peut réaliser. Ils peuvent être représentés par des ensembles d'états/transitions ou même directement des machines d'états dans certain cas. La définition d'un incrément et son intégration sont des tâches simples et surtout assez intuitives pour un concepteur. Cette démarche permet l'évolution simple et efficace de systèmes. Nous venons de montrer l'apport de notre méthode dans le travail de modélisation d'un composant, dans la suite nous allons montrer l'apport dans l'écriture des spécifications d'un composant.

Chapitre 3

Couplage conception incrémentale et vérification incrémentale

Ce chapitre montre l'utilité de la conception incrémentale pour la vérification d'un composant. La réutilisation de composant est devenue incontournable dans la conception des systèmes embarqués sur puce. Le concepteur ne construit plus une architecture en créant tous les composants mais réutilise des composants déjà existants. Il s'en sert comme des boîtes noires, il connaît leurs interfaces et leurs fonctionnalités grâce à leur spécification. Le problème d'énoncer une bonne spécification est donc devenu crucial. De plus, l'évolution d'un composant lorsque l'on ajoute des fonctionnalités rend encore plus difficile l'écriture de sa spécification. La méthode incrémentale permet de faire évoluer une spécification de façon simple et automatique.

Le processus de vérification dans lequel nous nous plaçons est le model checking. Un composant est décrit avec une spécification composée d'un ensemble de propriétés logico-temporelles. Cette technique a été utilisée avec succès pour de nombreux systèmes de taille moyenne, des expériences réussies sont décrites dans [GL00, HLQR99, PTK03]. Les spécifications que nous considérons sont décrites par un ensemble de formules CTL (définition 1.7).

La sémantique de la logique CTL est définie sur l'arbre d'exécution infini des structures de Kripke (def.1.4), nous allons donc montrer les structures de Kripke obtenues lorsque l'on suit les règles de la conception incrémentale. Nous nous sommes intéressées au lien qui existe entre l'évolution d'un composant (sa structure de Kripke) et l'évolution de sa spécification. Le problème peut être énoncé de la façon suivant : "Peut-on transformer une formule CTL qui est vraie dans un modèle i en une formule vraie dans le modèle incrémenté $i+1$?". Si une telle transformation est possible, l'ensemble des propriétés déjà vérifiées sur un modèle pourront être directement transformées et n'auront pas à être re-vérifiées sur le modèle plus complexe. La transformation des formules permettra d'alléger la phase de vérification par model checking d'un composant incrémenté. Notre but est de pouvoir construire un ensemble de propriétés CTL qui compose la spécification d'un modèle incrémenté en réutilisant la spécification d'un composant plus simple et dans certain cas la spécification de l'incrément. Evidemment pour la vérification complète du modèle incrémenté, de nouvelles propriétés devront être énoncées. Réciproquement, pour certaines propriétés du modèle complexe, nous voulons avoir la possibilité de les dériver

en propriétés vérifiables sur le modèle plus simple.

Etant donné un composant initial munis d'une spécification, un incrément, et un composant obtenu par cet incrément, nous montrons que la transformation de la spécification est possible. La transformation des propriétés CTL vraies sur le modèle initial, capture la façon dont le modèle plus complexe a été incrémenté. Cela garantit que si le modèle incrémenté est obtenu à partir des règles de la conception incrémentale alors les résultats de la vérification des propriétés CTL transformées sur le modèle incrémenté et la vérification des propriétés CTL initiales appliquées sur le modèle initial sont identiques.

Grumberg et Long ([GL91]), Loiseaux et al. ([LGS+95]) ont défini la préservation de propriétés CTL entre deux structures de Kripke ordonnées par une relation de simulation ($K(M_i) \preceq K(M_{i+1})$. Dans [LGS+95] C. Loiseaux et al. ont énoncé la préservation de l'ensemble des formules ECTL de $K(M_i)$ à $K(M_{i+1})$. Les formules ECTL expriment des propriétés vraies pour au moins un chemin. Lorsqu'il existe une relation de simulation entre deux modèles $K(M_1) \preceq K(M_2)$ alors pour tous les comportements de $K(M_1)$ il existe un comportement équivalent dans $K(M_2)$. Une propriété ECTL vraie dans $K(M_1)$ sera vraie dans $K(M_2)$ puisque celui-ci contient les comportements de $K(M_1)$. En revanche ce n'est pas le cas pour les propriétés ACTL, $K(M_2)$ contient plus de comportements que $K(M_1)$ et par conséquent peut avoir des chemins qui ne vérifient pas la propriété ACTL. Dans [GL91] est énoncée la préservation des propriétés ACTL de $K(M_{i+1})$ à $K(M_i)$. Ce résultat est très utile dans le cadre de vérification par abstraction ([CGL94]). Toutes les propriétés ACTL du modèle le plus abstrait sont conservés dans le modèle plus concret, le modèle le plus concret n'ajoute pas de nouveaux comportements.
Ces travaux considèrent seulement des fragments de CTL et ne transforment pas les formules. De plus la préservation est unidirectionnelle : une formule ACTL vraie sur $K(M_{i+1})$ sera vraie sur $K(M_i)$ alors qu'une formule ACTL vraie sur $K(M_i)$ peut ne pas être vraie sur le modèle incrémenté. Les résultats que nous présentons sur la transformation des formules CTL ne sont pas basés sur la préservation de propriétés CTL d'un composant à un autre qui l'englobe. Elle porte plutôt sur la transformation des formules CTL et permet d'obtenir la bi-implication entre les formules initiales et les formules transformées. Mais comme [GL91] et [LGS+95], pour pouvoir trouver cette transformation il va falloir caractériser le lien qui existe entre un modèle $K(M_i)$ et le modèle incrémenté $K(M_{i+1})$. La figure 3.1 résume les différentes relations que nous venons d'énoncer.

Notre transformation peut aussi être utilisée dans l'analyse de la non-régression des comportements d'un composant lors de sa conception. De façon générale, lorsqu'un concepteur modifie un composant, il doit s'assurer que les nouveaux comportements n'entraînent pas de régression : les corrections ne perturbent pas les autres fonctionnalités déjà correctes. En appliquant la démarche de conception incrémentale, la transformation de l'ensemble des propriétés CTL à partir d'un incrément peut se faire de façon automatique et ainsi produire une spécification juste du nouveau composant. Cette spécification correspond exactement aux anciens comportements, la non-régression est alors garantie *par construction.*

Dans le premier chapitre nous avons vu quels comportements pouvaient être ajoutés à un composant afin d'obtenir une implémentation complète de celui-ci. Dans ce chapitre, nous allons d'abord définir le composant obtenu par la méthode de conception incrémentale. Ensuite, nous étudierons l'incidence de ces transformations sur les structures de

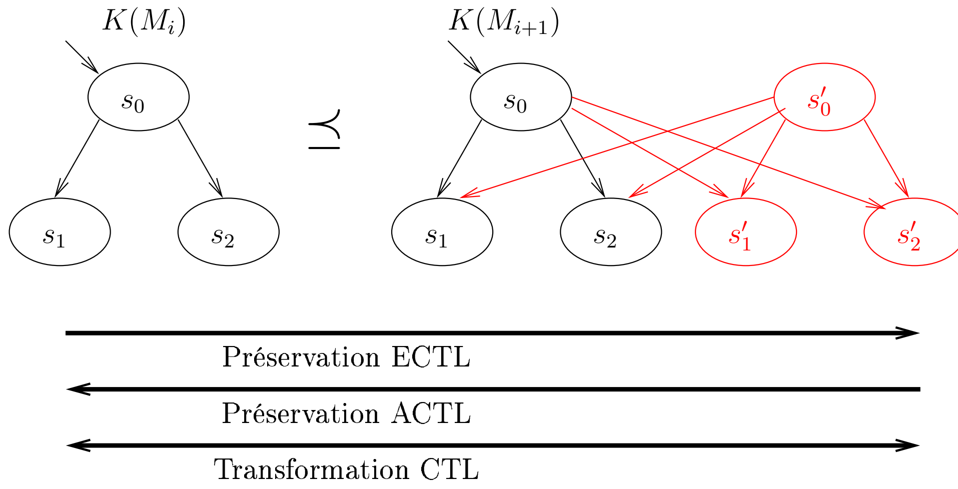


Fig. 3.1 – Relation entre les spécifications des modèles

Kripke des composants à différentes étapes de la conception et quelles sont les propriétés du modèle incrémenté. En outre nous montrerons ce qu'apporte la méthode incrémentale dans la vérification par model checking de composant. Nous montrerons que notre façon d'ajouter des comportements permet de faire évoluer la spécification d'un composant pendant sa conception. Notre méthode de conception allège la phase de vérification car, une partie de la spécification de l'étape i est directement obtenue à partir la spécification du composant à l'étape $i-1$. L'étape de model checking n'est donc pas à refaire sur cette partie de la spécification.

3.1 Propriétés d'un composant incrémenté

3.1.1 Automate de Moore M_{i+1}

Un incrément est un nouvel événement à l'interface d'un composant, nous avons vu (définition 2.1) qu'une valeur particulière permettait de différencier les nouveaux comportements des anciens. De plus, tous les anciens comportements sont préservés dans le nouveau composant, et doivent toujours pouvoir être atteints. Cela implique que l'incrément ne modifie pas les états initiaux du système et que tant que le nouvel événement ne s'exprime pas, le nouveau composant se comporte exactement comme l'ancien. Cette idée s'exprime dans le nouvel automate de Moore par le fait que la valeur muette de l'événement étiquette toutes les transitions qui étaient présentes dans l'ancien automate.

Nous allons définir plus formellement la nouvelle machine M_{i+1} obtenue à partir de $M_i = \langle S_i, I_i, O_i, T_i, \mathcal{G}_i, s_{0_i} \rangle$ et d'un incrément $INC = \langle M_{INC}, T_{i \to INC}, T_{INC \to i}, T_{i \to i} \rangle$. Pour cela nous avons besoin d'étendre les signatures des configurations d'entrées qui étiquettent toutes les transitions qui appartenaient à T_i. La fonction *Extend* complète l'ancienne configuration avec une configuration des nouveaux signaux d'entrée.

Définition 3.1 Fonction *Extend*
Soient M_i un composant et INC un incrément, soit $t = (s_1, c, s_2) \in T_i$, $Extend(t, I_+, c_{val}) =$

t' tel que $t' = (s_1, c', s_2)$ et $c' = c \wedge c_{val}$ (avec $c_{val} \in \mathcal{C}(I_+)$) et $proj(c', I_i) = c$.

Un composant M_{i+1} obtenu à partir d'un composant M_i et d'un incrément INC, préserve tous les comportements de M_i en supposant que dans M_{i+1}, le nouvel événement reste muet. L'accès aux nouveaux comportements ne peut se faire qu'en franchissant une transition où les signaux de l'événement sont dans une configuration active (figure 3.2 en haut).

Définition 3.2 *Soient un composant $M_i = \langle S_i, I_i, O_i, T_i, \mathcal{G}_i, s_{0_i}\rangle$ et un incrément $INC = \langle M_{INC}, T_{i\rightarrow INC}, T_{INC\rightarrow i}, T_{i\rightarrow i}\rangle$, le composant* incrémenté $M_{i+1} = \langle S_{i+1}, I_{i+1}, O_{i+1}, T_{i+1}, \mathcal{G}_{i+1}, s_{0_{i+1}}\rangle$ *est tel que*

$S_{i+1} = S_i \cup S_{INC}$;

$I_{i+1} = I_i \cup I_+$;

$O_{i+1} = O_i \cup O_+$;

$T_{i+1} = \{t' \mid t' = Extend(t, I_{INC}, e_qt), \forall t \in T_i\} \cup T_{i\rightarrow INC} \cup T_{INC\rightarrow i} \cup T_{i\rightarrow i}$;

$G_{i+1} = \mathcal{G}'_i \bullet \mathcal{G}_{INC}$: *$\bullet$ est l'opération de concaténation de vecteurs et $\mathcal{G}'_i$ est telle que pour tous le signaux $o_j \in O_+$ et pour tout $s \in S_i$, $g_j(d) \in C_{QT}(\{o_j\})$;*

$s_{i+1_0} = s_{i_0}$.

M_{i+1} est aussi une machine de Moore complète et déterministe. Le lien entre les machines M_i et M_{i+1} est capturé par une relation de simulation.

Proposition 1 *M_{i+1} simule M_i.*

Preuve On construit une relation binaire ρ_W entre des états de deux composants consécutif M_i and M_{i+1}. Cette relation est telle que $\rho_M \subseteq S_i \times S_{i+1} : \forall (s, c, p) \in T_i$ et $(s', c', p') \in T_{i+1}$, on a $(s, s') \in \rho_M$ ssi $s = s'$ and $c' = c \wedge e_qt$. Par construction ρ_M est une relation de simulation.

■

3.1.2 Structure de Kripke $K(M_{i+1})$

La structure de Kripke incrémentée $K(M_{i+1})$ peut facilement être déduite du composant M_{i+1} en appliquant la transformation définie chapitre 1. Nous obtenons alors une structure de Kripke $K(M_i) = \langle AP_i, \Sigma_i, \Sigma_{0_i}, \mathcal{L}_i, R_i\rangle$, dans la suite du document nous nommerons indifféremment $K(M_i)$ par K_i.

Nous voulons maintenant montrer que la structure $K(M_{i+1})$ englobe les comportements de $K(M_i)$. Dans le cas des automates de Moore nous avons vu que toutes les transitions qui appartenaient à M_i étaient étiquetées avec la valeur muette du nouvel événement. Dans les structures de Kripke, cette information se retrouve dans les états. Dans la figure 3.2, nous avons en bas la transformation incrémentale d'une structure de Kripke $K(M_i)$. Les états σ_1 et σ_2 de la structure $K(M_i)$ ont des états correspondants dans $K(M_{i+1})$. L'ensemble des signaux ont été étendus, par conséquent à l' état σ_1 correspond maintenant deux états σ'_1 et σ''_1, l'un avec une configuration muette du nouveau signal (e_qt), l'autre avec une configuration active (e_act). L'ensemble des états qui existaient

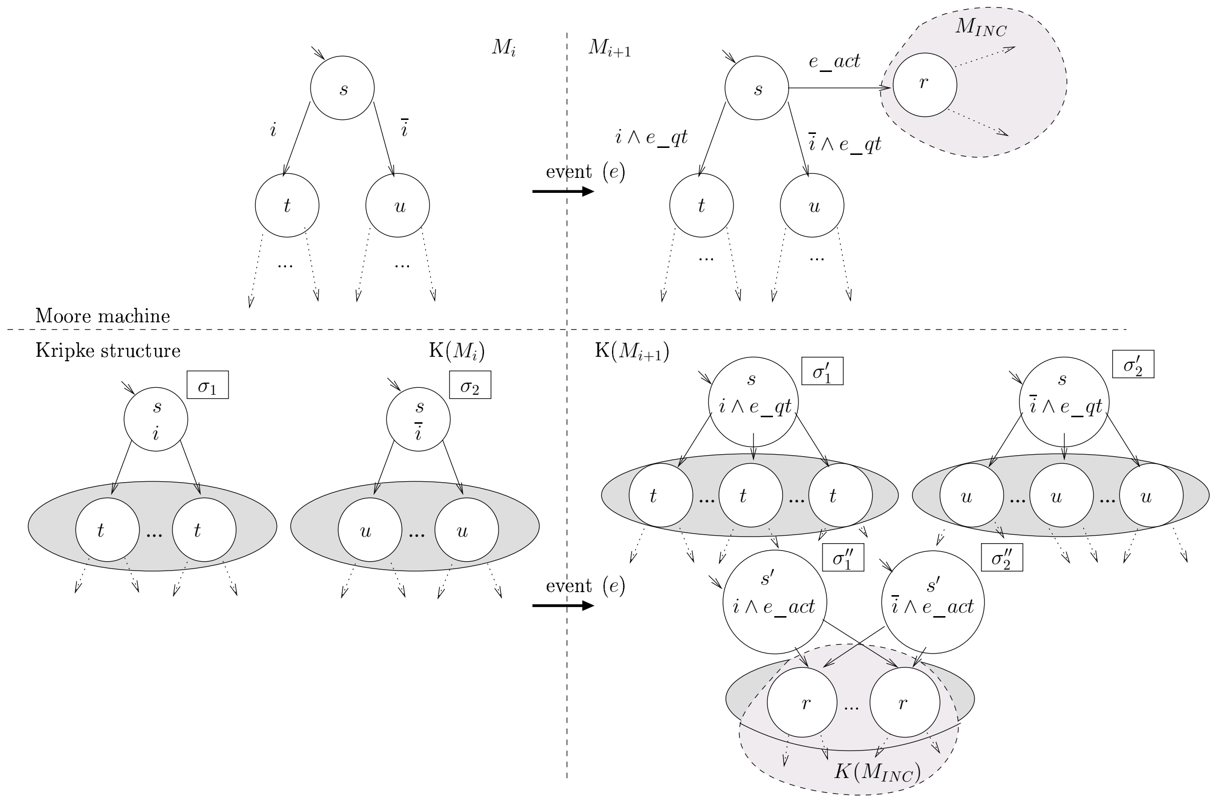


Fig. 3.2 – Règles incrémentales : transformations des automates de Moore et des structures de Kripke

dans $K(M_i)$ ont leur fonction d'étiquetage étendue par une configuration des nouveaux signaux et tel que pour chaque état de $K(M_i)$, il existe toujours un état correspondant dans $K(M_{i+1})$ où la configuration des nouveaux signaux est à une valeur muette. Dans la machine de Moore, l'accès aux nouveaux comportements se faisait en franchissant une transition étiquetée par e_act. Dans une structure de Kripke, l'accès au nouveaux comportements se fait en accédant à un état étiqueté par la valeur active de l'événement (e_act). La conception incrémentale permet d'avoir une *frontière très nette* entre ce qui existait déjà et ce que l'on ajoute. Tous les états préexistants sont accessible à partir d'un état initial étiqueté par la valeur e_qt et sont eux-mêmes étiquetés par une valeur e_qt. Tous ce qui est nouveau n'est accessible qu'à partir d'un état étiqueté par une valeur e_act.

L'ensemble des comportements de $K(M_i)$ se retrouve donc dans $K(M_{i+1})$. Cette relation de préservation est capturée par une relation de simulation entre les états des structures de Kripke. Plus précisément, la relation d'*enrichissement* explicite le fait que le nouveau composant englobe les comportements du composant précédent, et ces comportements sont marqués par une configuration muette du nouvel événement.

Définition 3.3 Relation d'enrichissement
Pour tous les états $\sigma = (s, c) \in \Sigma_i$*, il existe* $\sigma' = (s', c')$ *et* $\sigma'' = (s'', c'') \in \Sigma_{i+1}$ *tel que :*
$s' = s$, $c' = c \wedge e_qt$

$s'' = s$, $c'' = c \wedge e_act$.
On dit que σ' *et* σ'' enrichissent σ *(avec* e_qt *dans le premier cas).*

Comme pour les machines de Moore, les structures de Kripke $K(M_i)$ et $K(M_{i+1})$ sont liées par une relation de simulation. Pour une meilleur lisibilité, on note l'état initial de $K(M_i)$ $s_{K(M_i),0}$ par σ_0 et l'état initial de $K(M_{i+1})$, $s_{K(M_{i+1}),0}$ par σ'_0.

Proposition 2 *$(K(M_{i+1}), \sigma'_0)$ simule $(K(M_i), \sigma_0)$ avec σ'_0 enrichit σ_0 par e_qt*

Preuve On définit $\rho_{K_M} \subseteq \Sigma_{K(M_i)} \times \Sigma_{K(M_{i+1})}$, tel que $\forall \sigma = (s, c) \in \Sigma_{K(M_i)}$, $s \in M_i$ et $c \in C(I_i)$, $\exists \sigma' = (s', c') \in \Sigma_{K(M_{i+1})}$, avec $s' \in M_{i+1}$ et $c' \in C(I_{i+1})$. Pour σ et σ' on a $(\sigma, \sigma') \in \rho_{K_M}$ ssi $s' = s$ and $c' = c \wedge e_qt$. Par construction, ρ_{K_M} est une relation de simulation. ■

Remarque 3 De ce qui précède on a :
- σ' enrichit σ avec $e_qt \Rightarrow \sigma'$ simule σ
- σ' enrichit σ avec $e_act \not\Leftrightarrow \sigma'$ simule σ
- σ' simule $\sigma \not\Leftrightarrow \sigma'$ enrichit σ.

Les trois corollaires suivants caractérisent la frontière qui sépare les nouveaux des anciens comportements dans $K(M_{i+1})$. Rappelons que notre incrément est maintenant composés d'une structure de Kripke dérivée de la machine de Moore M_{INC}, et des connexions entre $K(M_i)$ et K_{INC}, tel que $INC = \langle K_{INC}, R_{i \to INC}, R_{INC \to i}, R_{i \to i} \rangle$ et :
- $R_{i \to INC} \subseteq \Sigma_i \times \Sigma_{0_{INC}}$ tel que $((s_1, c_1), (s_2, c_2)) \in R_{i \to INC}$ ssi $(s_1, c'_1) \in \Sigma_i$, $c_1 = c'_1 \wedge e_act$;
- $R_{INC \to i} \subseteq \Sigma_{INC} \times \Sigma_i$;
- $R_{i \to i} \subseteq \Sigma_i \times \Sigma_i$ tel que $((s_1, c_1), (s_2, c_2)) \in R_{i \to i}$ ssi $(s_1, c'_1) \in \Sigma_i$, $c_1 = c'_1 \wedge e_act$;

La figure 3.3 représente un exemple d'une structure de Kripke incrémentée et illustre les corollaires suivants.

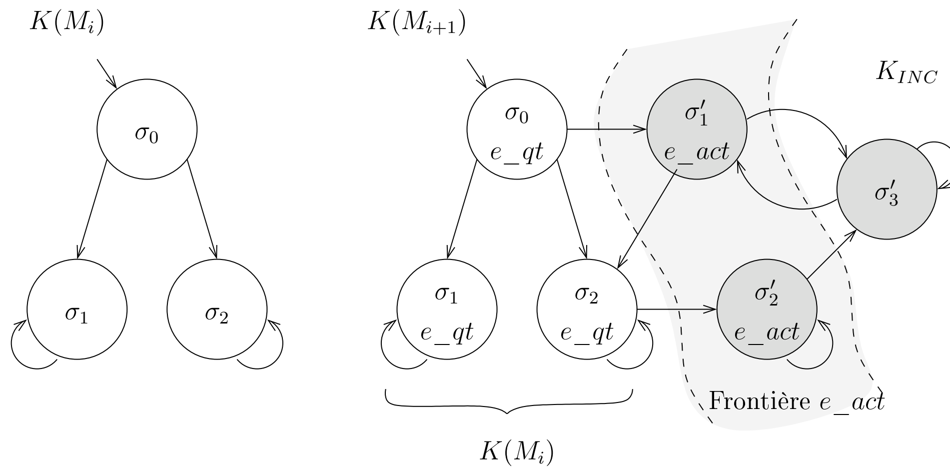


FIG. 3.3 – Illustration des corollaires

Corollaire 1 *S'il existe un chemin infini dans $K(M_i)$, alors il existe un chemin infini dans $K(M_{i+1})$ le long duquel l'événement e a toujours une configuration muette.*
Soit $\pi = \sigma_0 \xrightarrow{*} \sigma_n \xrightarrow{*} \ldots$ dans $K(M_i)$, il existe $\pi' = \sigma'_0 \xrightarrow{*} \sigma'_n \xrightarrow{*} \ldots$ dans $K(M_{i+1})$ tel que tout $i \in \mathbb{N}$, σ'_i enrichit σ_i avec e_qt.

Corollaire 2 *$K(M_i)$ est le sous-graphe maximal dans $K(M_{i+1})$, accessible à partir σ'_0, (qui enrichit σ_0 par e_qt) lorsque e reste dans une configuration muette.*

Corollaire 3

1. *L'ensemble des états de $K(M_{i+1})$ obtenus à partir d'un état de Σ_{INC}, ne sont accessibles à partir d'un état initial σ'_0 (qui enrichit σ_0 par e_qt) par un chemin le long duquel au moins un état est étiqueté par e_act.*
2. *Le long de ce chemin le premier état étiqueté par e_act enrichit par e_act un état de $K(M_i)$.*

Preuve

corollaire 1 Par induction sur la longueur de π.

corollaire 2 Par construction de $K(M_{i+1})$, on a que σ'_0 enrichit σ_0. D'après le corollaire 1, tous les chemins de $K(M_i)$ existent aussi dans $K(M_{i+1})$ étiquetés par la configuration muette de l'événement e. De plus, s'il existe un état $\sigma' \in K(M_{i+1})$ étiqueté par e_act, accessible par le chemin $\pi = \sigma'_0 \xrightarrow{*} \sigma'$ tel que tous les états σ_i sont étiquetés par e_qt, alors le successeur de σ' n'appartient pas à $\Sigma(K(M_i))$.

corollaire 3 Directement du corollaire 2.

■

Un quatrième corollaire découle des relations entre un modèle à l'étape i et à l'étape $i+1$. Contrairement au trois corollaires précédents, il énonce une propriété qui découle du composant le plus complexe sur le composant plus simple. Il établit qu'à partir de $K(Mi+1)$, on peut retrouver $K(M_i)$.

Corollaire 4 *Soit σ'_1 un état de $K(M_{i+1})$ qui enrichit σ_1, un état de $K(M_i)$, avec e_qt, alors pour tout $\sigma'_2 \in K(M_{i+1})$ tel que $\sigma'_1 \rightarrow \sigma'_2$, il existe $\sigma_1 \in K(M_i)$ tel que σ'_2 est obtenu à partir de σ_2, et $\sigma_1 \rightarrow \sigma_2$.*

Preuve Soit $\sigma'_1 \in K(M_{i+1})$ qui enrichit $\sigma_1 \in K(M_i)$ avec e_qt, et $\sigma'_1 \rightarrow \sigma'_2$. Supposons que σ'_2 *n'est pas* obtenu à partir d'un état σ_2 de $K(M_i)$. Par construction σ'_2 est produit par un état s de M_{i+1} (qui n'existe pas dans M_i) accessible par un transition étiquetée par e_act (Corollaire 3). On a donc que σ'_2 ne peut pas être le successeur de σ'_1 (voir Figure 3.2) (Contradiction). ■

3.2 Conséquence sur les spécifications

3.2.1 Transformation de la spécification de $K(M_i)$

Dans cette section nous allons énoncer les règles de transformation d'une propriété CTL ψ vraie sur un modèle initial en une propriété ψ' vraie sur le modèle incrémenté. La question à laquelle nous répondons peut être énoncée de la manière suivante : "Soit $\psi \in CTL$ telle que $K(M_i), \sigma_0 \models \psi$, quelle est la propriété $\psi' \in CTL$ telle que $K(M_{i+1}), \sigma'_0 \models \psi' \Leftrightarrow K(M_i), \sigma_0 \models \psi$?". Le principe de la transformation est basé sur la réduction du dépliage de l'arbre de la structure de Kripke. Nous avons vu dans la section précédente que les nouveaux comportements n'étaient accessibles qu'à partir d'états étiquetés par la valeur active du nouvel événement. L'ensemble de ces états représente la frontière entre les comportements de la structure $K(M_i)$ et $K(M_{i+1})$. De plus, cette frontière n'est atteinte que par des chemins étiquetés uniquement par la valeur muette (e_qt) et représentant les comportements de $K(M_i)$ dans la structure incrémentée $K(M_{i+1})$. L'idée de la transformation est de restreindre la propriété au sous-graphe se trouvant avant cette frontière, c'est à dire l'ensemble des états accessibles à partir de l'état initial étiqueté par e_qt.

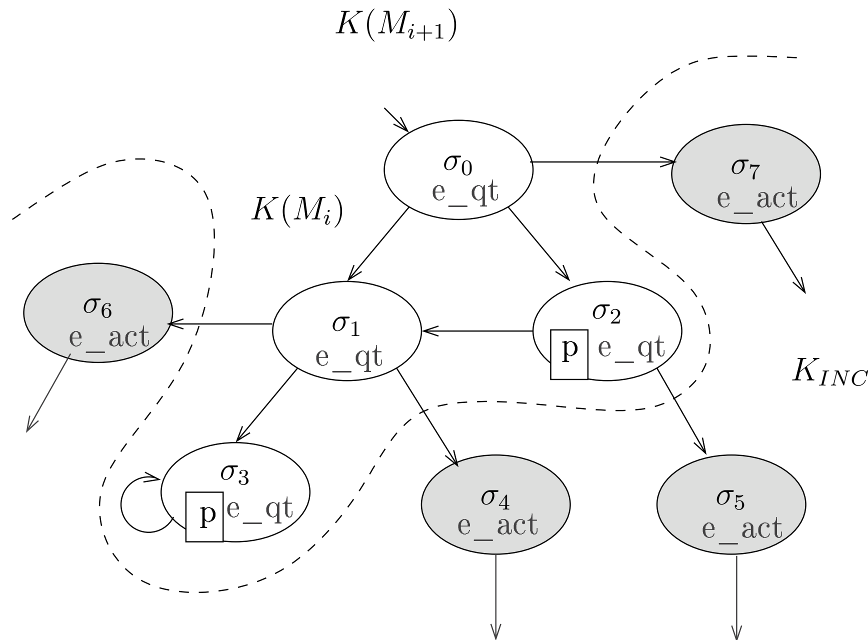


Fig. 3.4 – Frontière des valeurs actives dans $K(M_{i+1})$

Exemple Soit un nouvel événement $e = \langle\{\alpha\}, e_qt, e_act\rangle$, une structure de Kripke $K(M_i)$ telle que $K(M_i), s_0 \models \mathbf{AF}p$ et une structure de Kripke $K(M_{i+1})$ obtenue par l'incrément induit par e, $\langle K_{INC}, R_{i\rightarrow INC}, R_{INC\rightarrow i}\rangle$ (figure 3.4). Nous savons que si nous restreignons le parcours du graphe, à partir de l'état initial, au sous-graphe représentant $K(M_i)$ alors celui-ci vérifie $\mathbf{AF}p$. Dans $K(M_{i+1})$, tous les chemins où l'événement est muet vérifient donc $\mathbf{AF}p$. Mais il existe de nouveaux chemins amenant à de nouveaux

comportements. Par l'application des règles de la conception incrémentale, ces chemins ne sont accessibles qu'à partir d'états étiquetés par la valeur active de l'événement (ces états vérifient la proposition atomique e_act). Par conséquent, tous les chemins de $K(M_{i+1})$, à partir de l'état initial, restent dans les comportements préexistants (qui vérifient $\mathbf{AF}p$), ou rencontrent un état qui vérifie e_act. Cela s'exprime par la formule CTL suivante : $\mathbf{AF}(e_act \vee p)$.

Le théorème suivant énonce les règles de transformation des propriétés CTL pour un incrément général (définition 2.4). Nous ne donnons ici qu'une idée de la preuve. La preuve détaillée se trouve en annexe.

Théorème 1 *Soient $\sigma \in \Sigma_{K(M_i)}$ et $\sigma' \in \Sigma_{K(M_{i+1})}$ telle que σ' enrichit σ par e_qt. Pour toutes propositions atomiques $p \in AP_{K(M_i)}$ et pour toute formule CTL Φ, χ and Ψ (avec toutes leurs propositions atomiques dans $AP_{K(M_i)}$),*

$$K(M_i), \sigma \models \Phi \Leftrightarrow K(M_{i+1}), \sigma' \models \Phi'$$

où Φ' est la formule obtenue par l'application récursive des règles de transformation suivantes :

1	$\Phi = p$	$\Leftrightarrow \Phi' = p.$
2	$\Phi = \Psi \vee \chi$	$\Leftrightarrow \Phi' = \Psi' \vee \chi'.$
3	$\Phi = \neg\Psi$	$\Leftrightarrow \Phi' = \neg\Psi'.$
4	$\Phi = \mathbf{EX}\Psi$	$\Leftrightarrow \Phi' = e_qt \wedge \mathbf{EX}\Psi'.$
5	$\Phi = \mathbf{EF}\Psi$	$\Leftrightarrow \Phi' = \mathbf{E}[e_qt\mathbf{U}\Psi'].$
6	$\Phi = \mathbf{EG}\Psi$	$\Leftrightarrow \Phi' = \mathbf{EG}(e_qt \wedge \Psi').$
7	$\Phi = \mathbf{E}[\Psi\mathbf{U}\chi]$	$\Leftrightarrow \Phi' = \mathbf{E}[(e_qt \wedge \Psi')\mathbf{U}\chi'].$
8	$\Phi = \mathbf{E}[\Psi\mathbf{W}\chi]$	$\Leftrightarrow \Phi' = \mathbf{E}[(e_qt \wedge \Psi')\mathbf{W}\chi'].$
9	$\Phi = \mathbf{AX}\Psi$	$\Leftrightarrow \Phi' = e_qt \Rightarrow \mathbf{AX}\Psi'.$
10	$\Phi = \mathbf{AF}\Psi$	$\Leftrightarrow \Phi' = \mathbf{AF}(e_act \vee \Psi').$
11	$\Phi = \mathbf{AG}\Psi$	$\Leftrightarrow \Phi' = \mathbf{A}[\Psi'\mathbf{W}(e_act \wedge \Psi')].$
12	$\Phi = \mathbf{A}[\Psi\mathbf{U}\chi]$	$\Leftrightarrow \Phi' = \mathbf{A}[\Psi'\mathbf{U}((e_act \wedge \Psi') \vee \chi')].$
13	$\Phi = \mathbf{A}[\Psi\mathbf{W}\chi]$	$\Leftrightarrow \Phi' = \mathbf{A}[\Psi'\mathbf{W}((e_act \wedge \Psi') \vee \chi')].$

Preuve (Idée) : La transformation est basée sur la réduction de l'arbre d'exécution exploré au sous-arbre le long duquel la configuration active du nouvel événement n'est pas considérée. D'après le corollaire 2, ce sous-graphe représente $K(M_i)$. La transformation est alors prouvée pour chaque opérateur CTL auquel on inclut la contrainte e_qt ou e_act dans sa définition. La preuve procède alors par induction sur la formule Φ. ■

Pour une meilleur compréhension du théorème 1, nous donnons ici la signification intuitive de ces transformations :

- Ligne 1 : l'ensemble des propositions atomiques composant l'étiquette d'un état $\sigma \in \Sigma_{K(M_i)}$, étiquette aussi un état $\sigma' \in \Sigma_{K(M_{i+1})}$ qui enrichit σ.
- Ligne 4 : Soit $\sigma \in \Sigma_{K(M_i)}$, $\sigma \models \mathbf{EX}\Psi$, il existe $\sigma' \in \Sigma_{K(M_{i+1})}$ qui enrichit σ avec e_qt et tel que $\sigma' \rightarrow \sigma''$ et $\sigma'' \models \Psi'$

- Ligne 11 : Soit $\sigma \in \Sigma_{K(M_i)}$, $\sigma \models \mathbf{AG}\Psi$, l'opérateur **AG** se transforme en un opérateur **AW**. Dans la structure de Kripke incrémentée les chemins infinis correspondants aux comportements de $K(M_i)$ sont aussi présents. Il existe donc des chemins infinis où tous les états vérifient Ψ et où on accède jamais à un état étiqueté par e_act. La deuxième partie de la formule contenant l'opérateur weak until devient vrai dès que l'on accède aux nouveaux comportements, c'est à dire dès que l'on accède à un état étiqueté par e_act.

Les transformations des formules CTL ne modifient pas la structure temporelle de la formule initiale, dans le sens où l'imbrication des opérateurs reste identique. La taille des formules CTL obtenues après transformation, mesurée en nombre d'opérateur imbriqué, est donc inchangée. Cela vaut aussi pour la transformation des opérateurs **EF** et **AG** transformés en **EU** et **AW**. En effet, la formule obtenue ne change pas la complexité de la vérification de façon significative, étant donnée que ses opérateurs sont basés sur le même calcul de point fixe.

3.2.2 Incorporation de la spécification de l'incrément

Dans ce chapitre nous allons considérer les incréments $\langle K_{INC}, T_{i\to INC}, T_{INC\to i}, T_{i\to i}\rangle$ tels que $T_{i\to i} = \emptyset$. Dans le chapitre précédent nous avons défini trois types d'incrément particulier :

NORTN : sans retour ;

VALR : avec retour étiqueté par une valeur appartenant à $\mathcal{C}_{RTN}$;

NOVALR : avec retour, sans valeur particulière.

Ces trois types d'incrément possèdent une spécification CTL $SPEC_{INC}$. Est-il possible d'ajouter cette spécification à la spécification de K_{i+1} ? Si nous pouvons réaliser cela alors, à partir de la spécification du composant K_i et à partir de la spécification de l'incrément, nous obtenons la spécification complète de K_{i+1}. Pour les incréments particuliers de type **NORTN** et **VALR** une transformation des propriétés est possible afin de pouvoir les intégrer à la spécification de K_{i+1}. En revanche, le manque d'information sur les états de retour de l'incrément **NOVALR** nous empêche d'intégrer sa spécification.

Dans le cas d'incréments de type **NORTN**, le sous-graphe représentant les comportements de l'incrément vérifie les propriétés de $SPEC_{INC}$ à partir des états initiaux de K_{INC}. Par conséquent lorsque l'on ajoute un incrément sans retour, la spécification CTL de l'incrément sera une spécification de K_{i+1} dès que l'événement sera actif.

Théorème 2 *Soient K_{i+1} une structure de Kripke obtenue à partir de K_i par l'incrément INC de type* **NORTN** *et K_{INC} la structure de Kripke de l'incrément :*

$$K_{INC} \models \varphi \Rightarrow K_{i+1}, \sigma_0 \models \mathbf{A}[e_qt\mathbf{W}(e_act \wedge \mathbf{AX}\varphi)], \forall \sigma_0 \in \Sigma_{0_{i+1}}$$

Preuve Soit K_i une structure de Kripke, INC un incrément, tel que $K_{INC} \models \varphi$ et K_{i+1} la structure de Kripke incrémentée. D'après les corollaires 1 et 2, dans K_{i+1}, tous les comportements correspondant à K_i ont leur états étiquetés par e_qt et tous les chemins infinis de K_i se retrouvent dans K_{i+1}. De plus, d'après le corollaire 3 tous les nouveaux comportements ajoutés par l'incrément ne sont seulement accessibles en passant par un

état étiqueté par la valeur active de l'événement (e_act). De plus, d'après la définition 2.5, les transitions reliant K_i à K_{INC} accèdent aux états initiaux de K_{INC} ($R_{i \to INC} \subseteq (\Sigma_i \times C(I_i \cup I_+) \times \Sigma_{0_{INC}})$). Par hypothèse, les états initiaux de $\Sigma_{0_{INC}}$ vérifient φ. Par conséquent, tous les chemins infinis de K_{i+1} sont tels que e_qt est toujours vraie et dès que l'on accède à K_{INC} la propriété φ est vraie. Donc $K_{i+1} \models \mathbf{A}[e_qt\mathbf{W}(e_act \wedge \mathbf{AX}\varphi)]$. ■

Nous avons vu dans la section 3.2.1, que la valeur active d'un événement dessine une frontière entre les comportements du composant à l'étape i de conception et les comportements ajoutés. De la même manière, la valeur de retour trace aussi une frontière et celle-ci détermine la séparation entre les nouveaux comportements et les comportements initiaux. Nous pouvons appliquer le même raisonnement que celui utilisé dans la section 3.2.1, et établir que les états initiaux de K_{INC}, ne satisfont plus $SPEC_{INC}$ mais satisfont maintenant le nouvel ensemble de propriétés $SPEC'_{INC}$ directement dérivé de $SPEC_{INC}$. Pour cela on utilise la transformation récursive du théorème suivant qui est une extension du théorème 1. Cela montre aussi que pour toute structure dans laquelle il existe une frontière entre différents comportements, la transformation des formules CTL peut s'appliquer.

Théorème 3 *Soient K_{i+1} une structure de Kripke obtenue à partir de K_i par l'incrément INC de type* **VALR**, *K_{INC} la structure de Kripke de l'incrément et soient $\sigma \in \Sigma_{INC}$ et $\sigma' \in \Sigma_{i+1}$ tels que $\sigma' = \sigma$. Pour toutes propositions atomiques $p \in AP_{INC}$ et pour toute formule CTL Φ, χ and Ψ (avec l'ensemble des propositions atomiques appartenant à AP_{INC}), nous avons :*

$$K_{INC}, \sigma \models \Phi \Leftrightarrow K_{i+1}, \sigma' \models \Phi'$$

où Φ' est la formule obtenue en appliquant de façon récursive les transformations suivantes.

1 $\Phi = p \quad \Leftrightarrow \Phi' = p.$
2 $\Phi = \Psi \vee \chi \quad \Leftrightarrow \Phi' = \Psi' \vee \chi'.$
3 $\Phi = \neg\Psi \quad \Leftrightarrow \Phi' = \neg\Psi'.$
4 $\Phi = \mathbf{EX}\Psi \quad \Leftrightarrow \Phi' = \overline{e_rtn} \wedge \mathbf{EX}\Psi'.$
5 $\Phi = \mathbf{EF}\Psi \quad \Leftrightarrow \Phi' = \mathbf{E}(\overline{e_rtn}\mathbf{U}\Psi').$
6 $\Phi = \mathbf{EG}\Psi \quad \Leftrightarrow \Phi' = \mathbf{EG}(\overline{e_rtn} \wedge \Psi').$
7 $\Phi = \mathbf{E}[\Psi\mathbf{U}\chi] \quad \Leftrightarrow \Phi' = \mathbf{E}[(\overline{e_rtn} \wedge \Psi')\mathbf{U}\chi'].$
8 $\Phi = \mathbf{E}[\Psi\mathbf{W}\chi] \quad \Leftrightarrow \Phi' = \mathbf{E}[(\overline{e_rtn} \wedge \Psi')\mathbf{W}\chi'].$
9 $\Phi = \mathbf{AX}\Psi \quad \Leftrightarrow \Phi' = \overline{e_rtn} \Rightarrow \mathbf{AX}\Psi'.$
10 $\Phi = \mathbf{AF}\Psi \quad \Leftrightarrow \Phi' = \mathbf{AF}(e_rtn \vee \Psi').$
11 $\Phi = \mathbf{AG}\Psi \quad \Leftrightarrow \Phi' = \mathbf{A}[\Psi'\mathbf{W}(e_rtn \wedge \Psi')].$
12 $\Phi = \mathbf{A}[\Psi\mathbf{U}\chi] \quad \Leftrightarrow \Phi' = \mathbf{A}[\Psi'\mathbf{U}((e_rtn \wedge \Psi') \vee \chi')].$
13 $\Phi = \mathbf{A}[\Psi\mathbf{W}\chi] \quad \Leftrightarrow \Phi' = \mathbf{A}[\Psi'\mathbf{W}((e_rtn \wedge \Psi') \vee \chi')].$

La preuve est la même que pour le théorème 1.

A partir de ces transformations nous pouvons en déduire les règles de transformation des spécifications de l'incrément dans le cas où il existe une valeur spéciale de retour.

Théorème 4 *Soient K_{i+1} une structure de Kripke obtenue à partir de K_i par l'incrément INC de type* **VALR** *et K_{INC} la structure de Kripke de l'incrément. Nous avons :*

$$K_{INC} \models \varphi \Rightarrow K_{i+1} \models \mathbf{A}[e_qt\mathbf{W}(e_act \wedge \mathbf{AX}\varphi')]$$

φ' est obtenue en appliquant les règles du théorème 3.

Preuve C'est la conséquence directe des théorèmes 2 and 3. ■

La conception incrémentale est d'une part un outil pédagogique et méthodologique pour concevoir des composants. D'autre part, elle allège le travail de spécification. Elle rend possible la génération d'une partie de la spécification tout le long de la conception d'un composant. La décomposition des étapes de la conception permet de se focaliser sur des problèmes plus petits à chaque étape, la spécification de chacune des parties du composant est alors simplifiée. Nous avons montré que pour certain type d'incrément, cette spécification pouvait elle aussi être intégrée à la spécification du composant incrémenté. Nous pensons que de définir les frontières entre ce que l'on ajoute et ce qui existe déjà est une manière naturelle de procéder. Il est plus simple de raisonner sur de sous-fonctionnalités et de bien les spécifier afin de les intégrer facilement à l'architecture complexe.

La conception incrémentale permet d'obtenir tout ou une partie de la spécification de K_{i+1} directement à partir de K_i et de l'incrément. L'avantage majeur est que la spécification obtenue est garantie *par construction*. Par conséquent, la phase de model checking pour cette partie de la spécification n'a pas besoin d'être effectuée, cela permet de gagner un temps considérable pendant la phase de vérification.

Chapitre 4

Cas particulier du contrôle de flux dans les architectures pipelines

Dans ce chapitre nous particularisons la démarche de conception incrémentale au contrôle de flux des architectures pipelines. L'objectif est d'obtenir des résultats de transformations et de préservations plus forts. Les contrôles de flux sont la partie d'une application ou d'une architecture qui cadencent les différentes opérations que se déroulent les unes après les autres. Nous avons choisi ce type de composant d'abord parce que leur architecture est régulière, de plus, les contrôles de flux se retrouvent dans de nombreuses applications (pipeline de processeur, convertisseur de protocole, flexible manufacturing system (FMS) [Upt92]...). Enfin, notre définition des incréments simplifie la caractérisation de l'ensemble des événements à ajouter.

Un composant peut être vu comme un ensemble de traitements à réaliser et la partie contrôle comme l'automate qui autorise le passage de données aux différents étages de traitements. C'est cet automate qui contrôle le flux. La brique de base de la conception incrémentale est le composant tel que tous les traitements se font de manière continue : c'est le cas optimal. Les ajouts que nous allons considérer sont tous les événements qui perturbent la suite de traitements. Ils peuvent insérer des délais entre les étages, des bégaiements ou même modéliser des destructions volontaires de données. Ces événements sont produits par le gestionnaire d'événements. Celui-ci récupère tous les événements, internes au composant ou engendrés par l'environnement, puis les transmet à la partie de contrôle du flux. Du point de vue de la partie contrôle, il n'y a pas de différence entre des événements internes ou externes.

La figure 4.7 montre un composant qui prend des données en entrée, puis réalise trois traitements successifs T_0, T_1 et T_2. A chaque instant une donnée $Dout_i$ est produite par un étage i et est écrite dans un tampon qui sera ensuite la nouvelle donnée Din_{i+1} en entrée du traitement T_{i+1}. Dans ce chapitre nous considérons uniquement les tampons d'une seule place. Un tampon peut-être une FIFO à une place, ou un stock d'une capacité unitaire pour un flexible manufacturing system. La partie contrôle cadence le composant par l'intermédiaire des commandes x_0, x_1, x_2 et x_3. Elle autorise les données en sortie de chaque étage à passer à l'étage suivant. Dans cet exemple, il existe un tampon à l'entrée et à la sortie du composant. Dans la suite de ce chapitre, nous supposons que ces tampons sont présents mais notre démarche s'applique aussi aux composants sans tampon aux

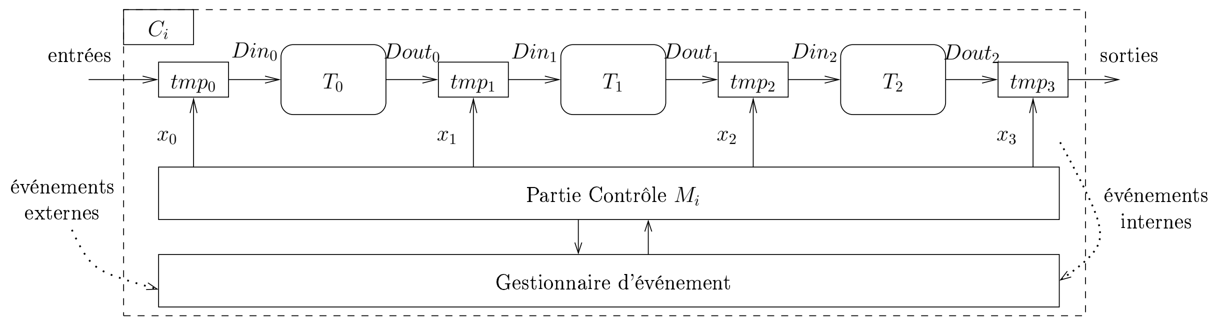


FIG. 4.1 – Suite de traitements

interfaces.

4.1 Formalisation du contrôle de flux

Nous représentons la partie contrôle par un automate de Moore $\langle M_i = S_i, I_i, O_i, T_i, G_i, s_{0_i}\rangle$. Ses états sont représentés par les signaux de sortie de l'automate. Les états peuvent être vus comme un vecteur de commandes $(x_0, \ldots, x_n)$. Chaque x_i représente l'autorisation de transmettre les données entre les traitements de l'étage $i-1$ à i. Les états de l'automate représentent à chaque instant les transferts de données possibles et les transitions représentent le flux de données dans le composant.

Chaque x_i peut donc prendre une des trois valeurs suivantes :
- $x_i = 0$ représente l'absence de donnée à traiter par T_i.
- $x_i = 1$ représente l'insertion du résultat de T_{i-1} à T_i.
- $x_i = \sharp$ représente le blocage du résultat de T_i, il ne peut passer dans T_{i+1}.

L'ensemble des vecteurs $V_l^r = x_l, x_{l+1}, ..., x_r$ est défini tel que $\forall i \in [l, r]$, $x_i \in \{0, 1, \sharp\}$. Ces vecteurs représentent l'état du flux de l'étage l à l'étage $r-1$.

Définition 4.1 *Un* mot d'état *pour un automate de Moore réalisant le contrôle de flux de n traitements est un vecteur $V_0^n = (x_0, x_1, ..., x_n)$, $\forall i \in [0, n]\, x_i \in \{0, 1, \sharp\}$.*

Les mots d'état représentent les état de l'automate de Moore.

A partir du mot d'état, nous pouvons décrire le contenu de l'ensemble des tampons du composant. En effet, si nous considérons un flux de k données en entrée pour $k \in \mathbb{N}$, nous avons :
- $x_i = 0 \implies tmp_i = \oslash$ et $Din_i = \oslash$,
- $x_i = 1 \implies tmp_i = Dout_{i-1}^{k+1}$ et $Din_i = Dout_{i-1}^k$,
- $x_i = \sharp \implies tmp_i = Dout_{i-1}^k$ et $Din_i = Dout_{i-1}^k$,

$Dout_i^k$ représente le résultat du traitement i pour la k^e donnée entrante. Le contenu des tampons peut être vu comme un vecteur composé de données $Dout_i^k$ ou de $\oslash$ quand le tampon est vide.

Définition 4.2 *Un* mot de donnée *pour un composant réalisant le contrôle de flux de n traitements est un vecteur $D_0^n = (tmp_0, tmp_1, ..., tmp_n)$ de $\forall i \in [0, n]\, tmp_i \in \{Dout_i^k, \oslash\}$.*

A partir des définitions du mot d'état et du mot de données nous pouvons décomposer les deux mots en préfixe et un suffixe.

Définition 4.3 Fonctions préfixe et suffixe *pour la machine de Moore M.*
Pour un étage donné $l \in [0, n-1]$, la fonction `pref` *: $\mathbb{N} \times S \rightarrow V_0^l$ associe à chaque état s* le préfixe *de s de 0 à l.*
Pour un étage donné $l \in [0, n-1]$, la fonction `suff` *: $\mathbb{N} \times S \rightarrow V_{l+1}^n$ associe à chaque état s* le suffixe *de s de $l+1$ à n.*
Pour un étage donné $l \in [0, n-1]$, la fonction $\texttt{pref}_d$ *: $\mathbb{N} \times S \rightarrow D_0^l$ associe à chaque état s* le préfixe des données *présentes dans les tampons de 0 à l.*
Pour un étage donné $l \in [0, n-1]$, la fonction $\texttt{suff}_d$ *: $\mathbb{N} \times S \rightarrow D_{l+1}^n$ associe à chaque état s* le suffixe des données *présentes les tampons de $l+1$ à n.*

La notion de progression des données inhérente à un flux est définie par la fonction de progression :

Définition 4.4 Fonction progress.
La fonction $\texttt{progress}_{l,r}$ *: $\{0,1\} \times V_l^r \rightarrow V_l^r$ est le décalage vers la droite de une place de chaque élément du vecteur V_l^r avec une injection soit d'un 0 ou d'un 1.*

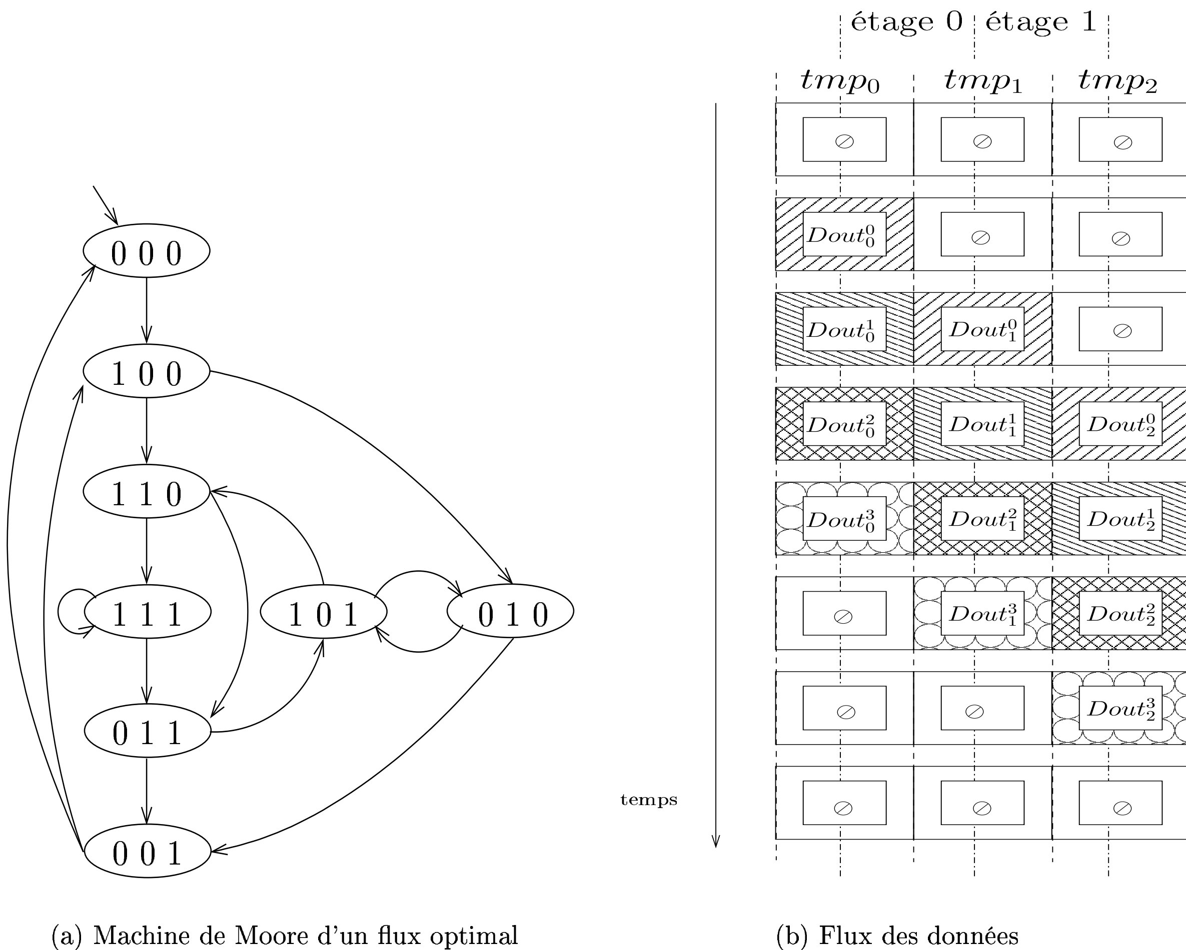
Les indices l et r de la fonction *progress* pourront être omis lorsqu'il n'y pas d'ambiguïté sur ce qu'ils représentent. A l'aide des différentes fonctions nous définissons le modèle de contrôle de flux optimal.

La figure 4.2(a) représente l'automate de contrôle du flux optimal, c'est à dire le premier composant de la conception incrémentale. Cet automate représente un flux pipeline de trois traitements, c'est à dire de trois étages. Pour réaliser de tels composants , la conception incrémentale, propose de commencer par décrire le cas où tous les traitements se font de manière successive et sans interruption possible du flux. Le concepteur commence par définir les différents traitements possibles, l'ordre de ces traitements, et les différents signaux qui cadencent le composant. Puis, il peut ajouter les différents événements qui perturbent le flux un à un. La figure 4.2(b) montre un exemple de donnée en transit dans le composant dans le cas d'un flux optimal.

La progression des commandes lors d'un flux optimal est représentée par un automate de Moore. Cet automate est tel qu'une donnée en entrée produira toujours un résultat (aucune donnée ne peut être perdue). De plus, le résultat sera disponible exactement à la fin du temps nécessaire pour effectuer tous les traitements. Aucune pause n'est possible dans la réalisation du flux (pas de cycle d'attente).

Définition 4.5 *Soit un contrôle de flux de n étages, $M_{opt} = \langle S_{opt}, S_{0_{opt}}, I_{opt}, O_{opt}, T_{opt}, \mathcal{G}_{opt} \rangle$ est un* automate de Moore optimal *si et seulement si*

- $S_{opt} \subseteq \{s \mid \forall x_i \in V_0^n,\ x_i = 0 \vee x_i = 1\}$ *; S_{opt} contient au maximum 2^n états.*
- $T_{opt} = \{(s, c, s') \mid s, s' \in S_{opt} \wedge s' = progress(*, s) \forall * \in \{0,1\}\}$



(a) Machine de Moore d'un flux optimal

(b) Flux des données

Fig. 4.2 – Le flux optimal pour un composant à 2 étages

4.2 Ajout de bégaiement

4.2.1 L'événement *stall*

Le premier événement que nous allons considérer est l'ajout d'un comportement qui ressemble aux anciens comportements mais dans lequel les différents traitements sont ralentis. Cet ajout représente des événements qui imposent d'attendre une certaine configuration de l'état du système ou de son environnement. Ce type d'événement peut par exemple caractériser l'attente par un processeur d'une donnée absente du cache. Le processeur est gelé jusqu'à ce que la donnée soit disponible.

Nous appelons ce type d'événement, un événement de gel (ou *stall*). La condition de gel à un étage r donné est modélisée par un événement $evt_stall_r = \langle stall_r, stall_r_act, stall_r_qt\rangle$. Lorsque une valeur active du signal $stall_r$ survient, alors l'ensemble des comportements engendrés sont les suivants :

- Tous les étages en amont $l \leq r$ sont gelés.
- L'étage r qui reçoit le gel ne transmet aucun résultat à l'étage suivant, l'étage $r+1$ reçoit une opération vide.
- Les étages en aval $l > r$ ne sont pas affectés : ils continuent leur progression normalement.

Lorsque que l'événement evt_stall_r redevient inactif (i.e. l'événement est dans une configuration muette), le pipeline reprend une progression optimale. La gestion de cet événement ajoute des états qui ne modifient pas les traitements en eux-mêmes mais ralentissent la cadence du flux et désynchronisent la progression des données à travers les différents étages.

Il est possible d'avoir plusieurs événements *stall* impliquant des étages différents. Les événements *stall* peuvent être composés. Dans ce cas, l'événement qui arrive à l'étage le plus en aval (le numéro d'étage le plus élevé) a plus d'impact sur le flux que des événements qui concernent des traitements plus en amont (numéro d'étage plus petit).

Définition 4.6 Ensemble de Stalls.
Soit $F = \{l \mid l \in [0, n-1]\}$ *l'ensemble des étages où un gel peut se produire.*

Remarque 4 Lorsque l'on introduit un nouvel incrément evt_stall_l ayant un impact sur l'étage $l < max(F)$, la configuration active est maintenant $\forall r \in F\ r > l,\ stall_r_qt \wedge stall_l_act$. Cela vient du fait que si un événement evt_stall_r est actif et gèle un étage plus en aval alors peu importe si l'événement evt_stall_l est aussi actif, evt_stall_r gèle $\texttt{pref}(r, s)$, qui comprend $\texttt{pref}(l, s)$.

Il existe une sorte de priorité entre les incréments *stall*. Lorsque deux incréments evt_stall_k et evt_stall_l tel que $l > k$ sont actifs en même temps, l'incrément qui s'exprime est evt_stall_l, puisque celui-ci englobe le précédent. Dans le cas où, lors de la conception, l'intégration du evt_stall_l se fait avant l'intégration du evt_stall_k, il faut faire attention à laisser la priorité à evt_stall_l en exprimant la valeur active de evt_stall_k comme l'ensemble des valeurs muettes des *stall* ayant un impact sur des étages supérieurs et la valeur active du nouvel événement.

4.2.2 L'incrément de bégaiement

Nous allons maintenant caractériser l'incrément induit par un événement *stall*. Cet incrément est appelé un *incrément bégayant*. En effet, les données en transit dans le composant peuvent simplement être retardées par l'intervention d'un gel. L'ensemble des données et commandes en amont de l'événement *stall* seront répétées jusqu'à ce que l'événement redevienne inactif. La suite de traitements effectués pour un même ensemble de données sera identique dans le modèle M_i et le modèle M_{i+1}. En revanche, la suite de données en sortie n'arriveront pas forcément au même moment dans les deux composants. La traversée d'une donnée dans M_{i+1}, peut être retardée par un gel. A l'interface du composant, l'ordre des traitements reste inchangé mais certaines données mettront plus de temps à sortir.

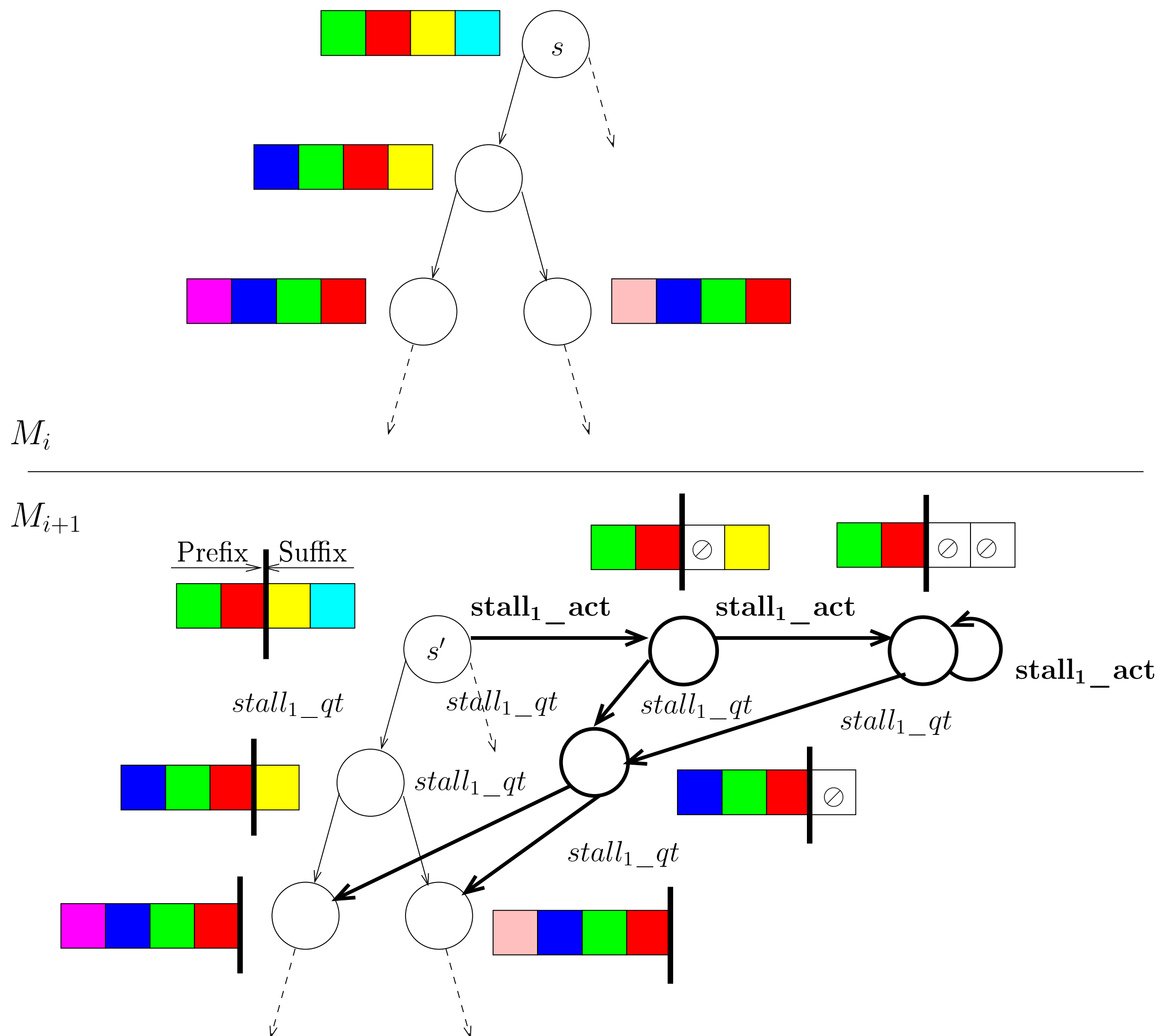


FIG. 4.3 – M_i incrémenté par $stall_2$

La figure 4.3 représente un arbre d'exécution possible des nouveaux comportements

engendrés par un événement *stall* à l'étage 1 d'un pipeline à 3 étages. Le composant M_i à un comportement régulier optimal. Le composant M_{i+1} a le même flux de données en entrée mais peut lui être gelé par l'activation du nouvel événement. Lorsque l'événement est actif, le suffixe continue sa progression alors que le préfixe de l'état est gelé tant que l'événement reste actif. L'ensemble de ces comportement définit l'incrément de bégaiement.

La définition suivante caractérise l'incrément de bégaiement dans le cas général. C'est à dire sans aucune restriction sur l'ensemble des états sur lesquels l'événement a un impact. Le concepteur peut restreindre les états concernés par le nouvel événement mais celui-ci doit être contenu dans l'ensemble que nous avons défini. L'incrément bégayant ajoute des nouveaux états par conséquent $T_{i\rightarrow i} = \emptyset$.

Définition 4.7 Incrément bégayant
Soient M_i un automate de contrôle de flux, un événement stall $evt_stall_l = \langle stall_l, stall_l_act, stall_l_qt \rangle$, l'incrément $INC = \langle M_{INC}, T_{i\rightarrow INC}, T_{INC\rightarrow i}, T_{i\rightarrow i}\rangle$ (avec $T_{i\rightarrow i} = \emptyset$) est défini par les règles suivantes :

R1 $\forall s \in M_i$, $\exists t = (s, stall_l_act, s')$ *tel que* $t \in T_{i\rightarrow INC}$ *et* $s' \in S_{0_{INC}}$

- $\forall x'_j \in \texttt{pref}(l, s')$: $x'_j = \begin{cases} \sharp & if\ x_j = 1 \\ 0 & if\ x_j = 0 \end{cases}$
- $\texttt{suff}(l, s'') = progress(0, \texttt{suff}(l, s))$

R2 $\forall s \in S_{INC}$, $\exists t = (s, c, s')$ *tel que*

- *Si* $c = stall_l_act$ *alors* $t \in T_{INC}$ *et* $s' \in S_{INC}$ *est obtenu par R1*
- *Si* $c = stall_l_qt$ *alors* $s' = progress(*, s)$ *et si* $s' \in S_i$ $t \in T_{INC\rightarrow i}$ *sinon* $t \in T_{INC}$.

4.2.3 Propriétés de l'incrément *stall*

Dans un premier temps nous ne considérons que les ajouts d'un unique incrément de bégaiement. La comparaison du flux des données à l'intérieur du pipeline entre M_i et M_{i+1}, montre que le préfixe du pipeline de M_{i+1} a toujours un équivalent au préfixe de M_i. Sur la Figure 4.4, on remarque que tant que l'événement *stall* est actif, le préfixe ne progresse pas. Puis, lorsque l'événement redevient inactif, la progression du préfixe est identique à la progression optimale. La relation suivante exprime cette progression.

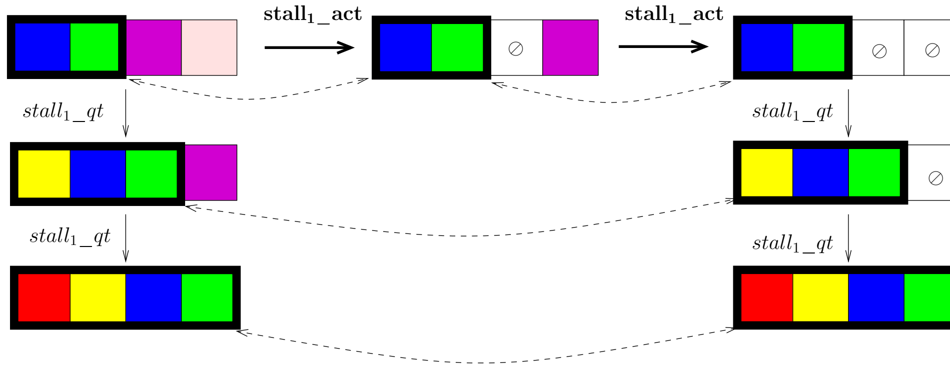


FIG. 4.4 – Progression du préfixe

Définition 4.8 Progression du préfixe.
Soit un événement $evt_stall_l = \langle stall_l, stall_l_act, stall_l_qt \rangle$ *intervenant à un étage* l*, qui amène la machine* M_i *à la machine* M_{i+1} *sur un même flux de données. Soit* B_l *une relation binaire dans* $S_i \times S_{i+1}$ *tel que :*
$(s_i, s_{i+1}) \in B_l$ *ssi* $\texttt{pref}_d(l, s_i) = \texttt{pref}_d(l, s_{i+1})$.

Propriété 4.1 B_l *est une relation d'équivalence bégayante.*

Preuve Par définition de l'incrément bégayant 4.7 (règle *R1*), $\exists s_1 \in S_i$ et $\exists s_2 \in S_{i+1}$ tel que $(s_1, s_2) \in B_l$. Soit $s'_1 \in S_i$, tel que $s'_1 = progress(x, s_1)$, alors

- Par construction de M_{i+1}, $\exists t \in T_{i+1}$ tel que $t = (s_2, stall_l_qts'_2)$ et $s'_2 = progress(x, s_2)$. Par conséquent, $\texttt{pref}_d(l{+}1, s'_1) = \texttt{pref}_d(l{+}1, s'_2)$ qui inclut $\texttt{pref}_d(l, s'_1) = \texttt{pref}_d(l, s'_2)$
- $\exists t \in T_{i+1}$ tel que $t = (s_2, stall_l_act, s'_2)$ alors $\forall x'_j \in \texttt{pref}(l, s'_2) : x'_j = \begin{cases} \sharp & \text{if } x_j = 1 \\ 0 & \text{if } x_j = 0 \end{cases}$
  On a alors $\texttt{pref}_d(l, s_1) = \texttt{pref}_d(l, s'_2)$. Cela est vrai le long du chemin où l'événement est actif. Dès que l'événement redevient inactif, par construction de M_{INC} (règle *R2*) $\exists s''_2 = progress(x, s_2)$. Par conséquent, $\exists \pi = s_2 \xrightarrow{*} s''_2$ et $\texttt{pref}_d(l+1, s'_1) = \texttt{pref}_d(l+1, s'_2)$ qui inclut $\texttt{pref}_d(l, s'_1) = \texttt{pref}_d(l, s'_2)$

Par conséquent B_l est une relation d'équivalence bégayante. ■

La même démarche appliquée au suffixe ne fonctionne pas : ce n'est pas une relation de bisimulation. La définition suivante exprime la progression du suffixe lors de l'ajout d'un incrément *stall*.

Définition 4.9 Progression du suffixe.
Soit un événement $evt_stall_l = \langle stall_l, stall_l_act, stall_l_qt \rangle$ *intervenant à un étage* l*, qui amène la machine* M_i *à la machine* M_{i+1} *pour un même flux de données. Soit* R_l *une relation binaire dans* $S_i \times S_{i+1}$ *tel que :* $(s_i, s_{i+1}) \in R_l$ *ssi*

1. $\texttt{suff}_d(l, s_i) = \texttt{suff}_d(l, s_{i+1})$ *et*
2. $\forall s'_i \in S_i$ *tel que* $s_i \rightarrow s'_i$, $\exists s'_{i+1} \in S_{i+1}$ *tel que* $s_{i+1} \rightarrow s'_{i+1}$ *et* $(s'_i, s'_{i+1}) \in R_{l+1}$.

Malheureusement, la relation R_l n'est pas une bisimulation car la relation R_{l+1} n'est pas incluse dans R_l. Cette propriété est seulement locale au gel et n'exprime que la progression du suffixe, que le flux en aval soit gelé ou non.

Maintenant nous nous concentrons sur l'évolution du flux de données (figure 4.5). Dans le cas où le flux est optimal, le préfixe évolue le long des différents traitements. L'instruction en vert est initialement au premier étage, puis elle traverse le second etc... Dans le cas d'un gel au deuxième étage, les instructions de l'étage 0 et 1 vont rester dans le même étage puis, une fois que le gel redevient inactif, continuent leur progression. Si on regarde uniquement ces deux instructions, leur progression est bégayante par rapport à une progression optimale. Cette caractéristique est formalisée par la propriété suivante.

Propriété 4.2 Progression bégayante.
Dans M_i *: on a* $\pi = s_0 s_1 \xrightarrow{*} s_n$ *tel que* $s_n : V_n^{n+l} = progress^n(V_0^l)$.
Dans M_{i+1} *: Soit* $\texttt{stall}_l$ *l'action de gel à l'étage* l*. Alors* $\exists \pi' = (s_0)^* \rightarrow s_1 \xrightarrow{*} s'_n$ *tel que* $s'_n : V_n^{n+l} = progress^n(V_0^l)$.

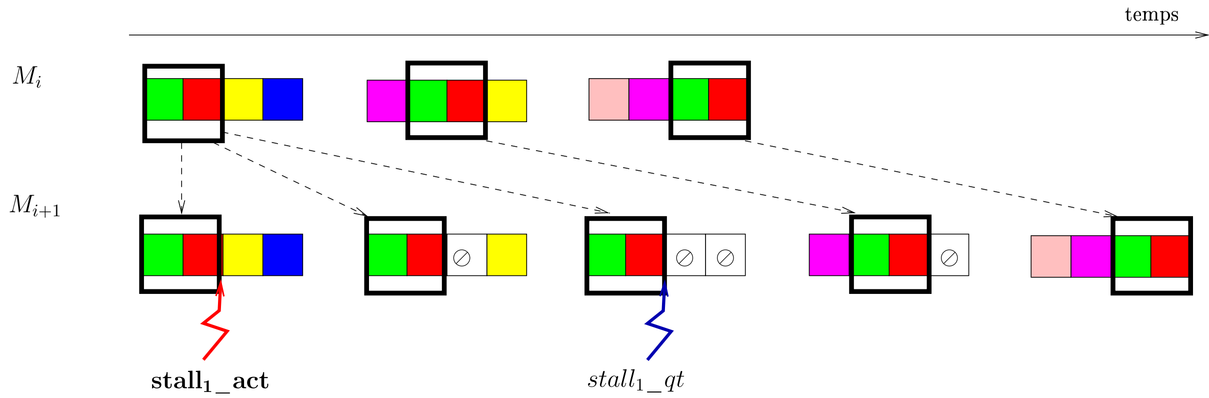


FIG. 4.5 – Progression bégayante du préfixe

Preuve C'est la conséquence directe de la règle de construction *R2* (En ajoutant l'hypothèse que le gel aura toujours une fin). ■

Dans le cas où l'on ajoute plusieurs événements *stall*, la propriété de bisimulation sur les préfixes ne s'applique plus directement. En effet, il est possible d'avoir une partie des données en transit dans le composant qui reprennent leur progression alors que d'autres données en amont sont toujours gelées par un événement affectant des étages inférieurs.

Propriété 4.3 *(Extension de la propriété 4.8 pour un ensemble d'événement stall F)* *Soit un automate M_{i+1} obtenu après l'application de plusieurs incréments bégayants à partir de l'automate M_i. L'ensemble des événements stall est F'. Soient $l \leq min(F')$ et $R_l \subseteq S_i \times S_{i+1}$ la relation définie telle que*
$s_i \; R_l \; s_{i+1}$ ssi $\texttt{pref}_d(l, s_i) = \texttt{pref}_d(l, s_{i+1})$.

1. *R_l est une relation d'équivalence bégayante.*
2. *$\forall \; j > l$, R_j n'est pas une bisimulation.*

Preuve La preuve du premier point procède comme pour la propriété 4.8.
Pour le deuxième point : Dans le cas d'un seul incrément bégayant à l'étage l, les traitements allant de 0 à $l-1$ ont la même progression : soit ils sont gelés ($stall_l_act$), soit ils progressent à la même vitesse (lorsque l'événement *stall* redevient inactif). Cela est capturé par les propriétés 4.8 et 4.2 Maintenant si $l > min(F_s)$, alors il existe un gel $r < l$ qui coupe l'intervalle des traitements $[0; l[$ en deux intervalles $[0; r]$ et $]r; l[$. Les étages dans l'intervalle $[0; r]$ sont gelés jusqu'à ce que l'événement $stall_r$ redevienne inactif. Alors que les traitements $]r; l[$ progressent. Par conséquent, l'équivalence des traitements de l'intervalle $]r; l[$ n'est plus capturée par R_l mais par R_r et la propriété de progression bégayante. ■

La propriété de progression bégayante 4.2 demeure : elle est toujours valable lorsque M_i est soumise à plusieurs événements *stall* sur différents étages.

Dans le cas où les tampons situés entre les différents traitements contiennent plus d'une place, alors les propriétés de l'un incrément bégayant doivent être adaptées. La suite de donnée et l'ordre des traitements restent toujours inchangés. Maintenant, il est

possible que l'amont continue à progresser tant que les FIFOs ne sont pas pleine, alors que l'aval continue sa progression avec l'insertion d'un zéro. La condition d'un gel à un étage dépend aussi du remplissage des FIFOS en sorties. Les propriétés 4.8 4.2 et 4.3 sont toujours valables.

4.2.4 Ajout de destruction

Le deuxième type d'événement qui modifie le contrôle de flux est l'événement *kill*. L'action d'un événement *kill* détruit le traitement à un étage donné, mais ne perturbe pas le flux de contrôle. En revanche, le flux de données va lui être modifié, puisqu'une donnée entrante pourra ne jamais sortir. Le *kill* est l'opération de base utilisée lors d'une ré-initialisation d'un composant ou lors d'un changement de contexte sur un processeur. Dans notre modélisation, cette action consiste à remplacer l'autorisation de transfert par une instruction vide et le résultat d'un étage l par une donnée vide. Le résultat du traitement à cet étage n'est pas pris en compte à l'étage suivant. L'action à l'étage l est représentée par l'événement $evt_kill_l = \langle kill_l, kill_l_act, kill_l_qt\rangle$. Les changements de comportement qu'il induit sont les suivants :

- le résultat du traitement à l'étage l est annulé : la barrière de registre à l'étage $l+1$ prend une opération vide.
- Les autres étages progressent normalement.

Cela entraîne que la suite des données présentes dans le composant n'est plus continue.

La figure 4.6 représente un arbre d'exécution de l'automate de Moore de la partie contrôle optimal plus l'incrément induit par un événement evt_kill_1 à l'étage 1. Comme on l'a vu précédemment, toutes les transitions qui étaient déjà présentes sont maintenant étiquetées avec une valeur muette de l'événement. Lorsque l'événement s'exprime (valeur active), la progression du flux reste constant, en revanche il y a une insertion d'une opération vide à l'étage 2. L'instruction rose ne termine pas sa progression dans le pipeline. Tant que l'événement est actif, toutes les instructions passant par l'étage 1 sont détruites, dès que l'événement redevient muet, le pipeline se comporte exactement comme dans le cas d'un flux optimal.

Dans notre représentation, une action *kill* consiste à modifier la valeur de la commande x_i par une opération vide qui détruit le résultat du traitement. Un incrément $kill_l$ est défini de la manière suivante

Définition 4.10 Incrément de destruction
Soit un événement $evt_kill_l = \langle kill_l, kill_l_act, kill_l_qt\rangle$ *impactant l'étage* l *de l'automate* M_i*,* INC *peut être défini de la manière suivante :*

- T_{INC} *: L'ensemble des nouvelles transitions est défini tel que*

  R3 $\forall s \in S_i, \exists t = (s, kill_l_act, s')$ *tel que* $t \in T_{i\rightarrow INC}$ *et* $s' \in S_{INC}$ *est tel que* $x'_0 = 0$ *ou* 1*,* $x_l = 0$ *et* $\forall i \neq l$*,* $x'_i = x_{i-1}$*.*

  R4 $\forall s \in S_{INC}$*, obtenu par R3,* $\exists t \in T_{INC}$ *telle que* $t = (s, kill_l_act, s')$ $t \in T_{INC}$ *et* $s' \in S_{INC}$ *est aussi obtenu par R3.*

  R5 $\forall s \in S_{INC}$*,* $\exists t = (s, kill_l_qt, s')$*,* $s' = progress(*, s)$ *et si* $s \in S_i$ *alors* $t \in T_{INC\rightarrow i}$ *sinon* $t \in T_{INC}$*.*

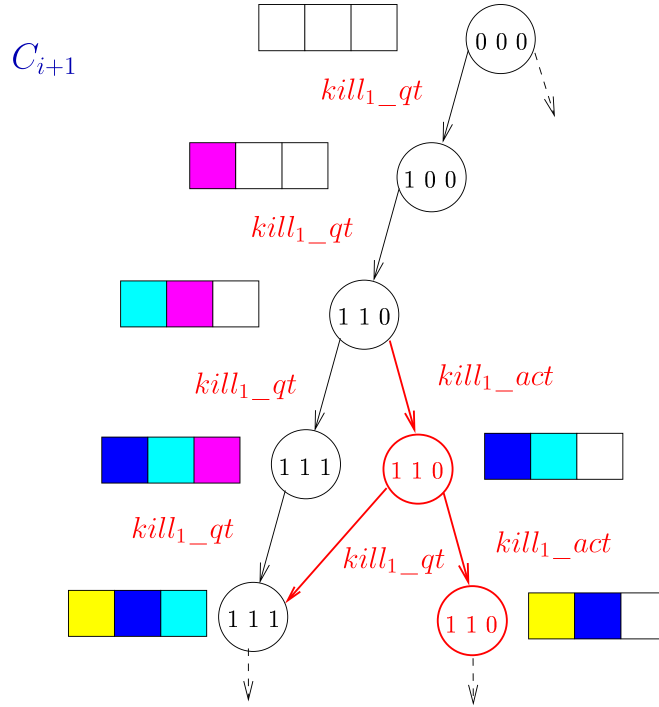


FIG. 4.6 – Graphe d'exécution de la machine de Moore avec un kill à l'étage 1

De cette définition se dégagent aussi des propriétés sur les préfixes et les suffixes des états. Mais dans ce cas une donnée peut être détruite, par conséquent le flux de données n'est plus le même. Le suffixe continue sa progression normalement, alors que le préfixe a toujours une progression continue mais la donnée de l'étage l a été détruite.

Définition 4.11 Progression des suffixes
Soit un événement $evt_kill_l = \langle kill_l, kill_l_act, kill_l_qt\rangle$ *intervenant à un étage* l*, qui amène la machine* M_i *à la machine* M_{i+1} *pour un même flux de données. Soit* R_l *une relation binaire dans* $S_i \times S_{i+1}$ *tel que :* $(s_i, s_{i+1}) \in R_l$ *ssi*

1. $\mathtt{suff}_d(l, s_i) = \mathtt{suff}_d(l, s_{i+1})$ *et*
2. $\forall s_i' \in S_i$ *tel que* $s_i \rightarrow s_i'$, $\exists s_{i+1}' \in S_{i+1}$ *tel que* $s_{i+1} \rightarrow s_{i+1}'$ *et* $(s_i', s_{i+1}') \in R_{l+1}$.

Comme nous l'avons vu pour un incrément de bégaiement, R_l n'est pas une relation de bisimulation.

Propriété 4.4 Progression des préfixes
Soit un événement $evt_kill_l = \langle kill_l, kill_l_act, kill_l_qt\rangle$ *intervenant à un étage* l*, qui amène la machine* M_i *à la machine* M_{i+1} *pour un même flux de données. Soit* B_l *une relation binaire dans* $S_i \times S_{i+1}$ *tel que :* $(s_i, s_{i+1}) \in B_l$ *ssi* $\mathtt{pref}_d(l-1, s_i) = \mathtt{pref}_d(l-1, s_{i+1})$. B_l *est une relation de bisimulation.*

Preuve Par construction de M_{INC} (règle *R3*), il existe $s_2 \in S_{i+1}$ et $s_1 \in S_i$ tel que $\mathtt{pref}_d(l-1, s_i) = \mathtt{pref}_d(l-1, s_{i+1})$. Soit $s_1' \in S_i telque s_1 \rightarrow s_1'$ alors

- $\exists t = (s_2, kill_l_qt, s'_2)$ et $s'_2 = progress(*, s_2)$, par conséquent $\texttt{pref}_d(l, s'_1) = \texttt{pref}_d(l, s'_2)$ qui inclus $\texttt{pref}_d(l-1, s'_1) = \texttt{pref}_d(l-1, s'_2)$.
- $\exists t = (s_2, kill_l_act, s'_2)$ alors le flux progresse normalement à l'exception de $x_l = 0$. On a alors $\texttt{pref}_d(l-1, s'_1) = \texttt{pref}_d(l-1, s'_2)$.

B_l est une relation de bisimulation. ■

Les règles de progressions des préfixes et des suffixes mettent en évidence que la progression totale du flux n'est pas modifiée par un incrément *kill*. Un événement *kill* ne modifie pas la progression des données et des commandes dans le flux. En revanche il modifie la suite des données par la destruction de certaine donnée.

4.3 Vérification du contrôle de flux

La caractéristique principale de la démarche incrémentale est la garantie par construction de la non-régression d'une implémentation tout au long de sa conception. Dans le cas des architectures décrites dans la section précédente section 4.2, la régularité du flux nous permet d'être plus précis sur les transformations de la spécification lors d'ajouts d'un nouvel événement *stall* ou *kill*. En effet, certaines classes de formules peuvent directement faire partie de la spécification d'un modèle plus complexe sans avoir à être transformées. De plus, la caractérisation des ajouts des incréments *stall* et *kill* nous permet d'obtenir des transformations de formules plus fines.

Dans cette section, nous donnons les résultats de préservation ou de transformation des formules CTL entre une machine initiale et une autre obtenue par composition des différents incréments *stall* et *kill*. La progression bégayante du flux engendré par l'ajout d'incrément *stall*, nous impose de considérer les formules CTL privées de l'opérateur "Next" CTL\X .

4.3.1 Les classes de propriétés CPI et CPE

Les formules de CTL\X ont été divisées en 2 classes : les propriétés internes et les propriétés externes. La première spécifie les comportements à l'intérieur du pipeline, elles portent sur l'enchaînement des actions dans le composant. La seconde classe spécifie les comportements aux deux extrémités du pipeline. Elles concernent les actions et données qui entrent dans le pipeline et les résultats produits.

Les propositions atomiques des spécifications d'un composant se rapportent à un traitement (T_l) en particulier. On note ϕ_l la proposition atomique (ou *sa négation*) qui spécifie une partie de l'étage l. Cette propriété porte sur la commande x_l et sur les données présentes à l'intérieur des tampons tmp_l (fig.4.7).

Définition 4.12 Classe des propriétés internes (**CPI**)
Soient f et g deux formules positives de CTL\X , $l, r \in [0, n-1]$, les formules f et g sont construites suivant les règles :

- $p = \phi_l \mid \phi_l \wedge \phi_r \mid \phi_l \vee \phi_r \mid \mathbf{true} \mid \mathbf{false}$
- $f = p \mid \mathbf{A}[f\ \mathbf{U} g] \mid \mathbf{E}[f\ \mathbf{U} g] \mid \mathbf{AG} f \mid \mathbf{EG} f \mid \mathbf{AF} f \mid \mathbf{EF} f \mid f \vee g \mid f \wedge g$

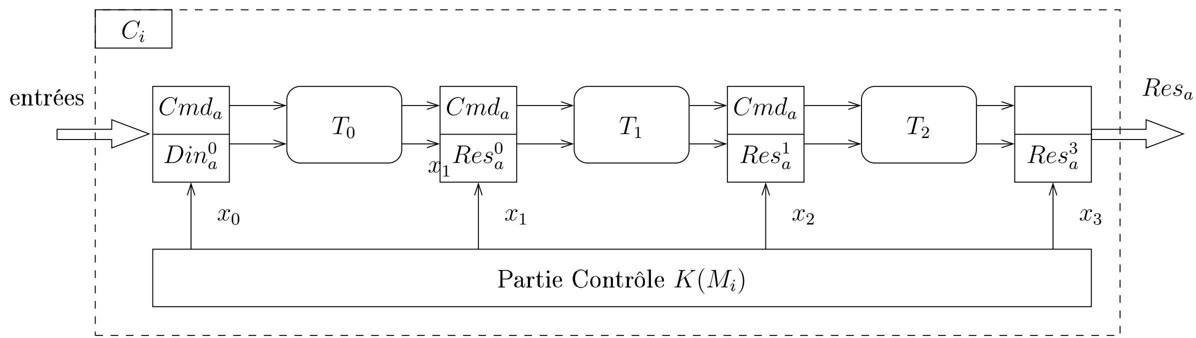


FIG. 4.7 – Découplage commande résultat

La classe de propriétés externes concerne les propriétés globales du composant, c'est à dire les propriétés liant les commandes qui entrent et les résultats produits en sortie du pipeline. Le composant est considéré comme une boîte noire, le fonctionnement peut être vu comme une commande qui entre et qui plus tard (selon le nombre de traitements et les actions de gel) produira un résultat. L'environnement du composant peut être alors considéré comme un ensemble de paires commande/résultat $E = \{(Cmd_a, Res_a)\}$, où le couple (Cmd_a, Res_a) représente la commande a et le résultat associé (figure 4.7). Les liens de causalités entre une commande et son résultat peuvent être modélisées par des formules CTL\X [DAC99]. Une commande entrante peut être exprimée de la manière suivante : $\phi_{0,a} = (x_0 = 1 \wedge tmp_0 = Cmd_a)$ (à l'étage 0 la commande a est écrite dans le tampon). La fin du calcul induite par Cmd_a est exprimé par : $\phi_{n-1,a} = (x_{n-1} = 1 \wedge tmp_{n-1} = Cmd_a \wedge Res_a)$ (le résultat est produit par le dernier étage). On note $\phi_{l,a}$ la proposition atomique (ou *sa négation*) qui spécifie une partie de l'étage l pour une commande a donnée.

Définition 4.13 Classe de propriétés externes (**CPE**)
L'ensemble des propositions atomiques $\phi_{i,a}$ est tel que i peut prendre la valeur 0 ou n-1. Soient f et g des formules de **CPE**, *et p, $\phi_{i,a}$ des propositions atomiques. La formule f est construite selon les règles suivantes :*
- $p = \phi_{i,a}$ | **true** | **false**
- $f_p = p \mid \phi_{0,a} \Rightarrow \mathbf{AF}\phi_{n-1,a}$
- $f = f_p \mid \mathbf{AF}f_p \mid \mathbf{EF}f_p \mid \mathbf{AG}f_p \mid \mathbf{EG}f_p \mid f \vee g$

Les propriétés de la classe **CPE** ne peuvent pas être des conjonctions. En effet, cette classe de propriétés est définie pour un calcul a donné.

4.3.2 Transformations et préservations de la spécification

Les théorèmes suivants énoncent la préservation ou les différentes transformations des formules CTL lors de l'évolution d'une spécification d'un modèle initial M_i en une spécification du modèle M_{i+1}. M_{i+1} est obtenue par la composition d'un ensemble d'incréments *stall* F'_s. On appelle F_s, l'ensemble des *stall* défini pour la machine M_i ($F'_s = F_s \cup \{new_stall_j\}$).

L'équivalence bégayante des préfixes permet de préserver l'ensemble des formules basées sur des propositions atomiques du préfixe, déterminé par le minimum des étages

affectés par les événements *stall*, et par conséquent toutes les formules CTL\X dont l'ensemble de propositions atomiques sont dans le préfixe. Mais cette préservation ne concerne pas les formules dont les propositions atomiques concernent des étages non compris dans le préfixe. Un incrément *stall_j* gèle une partie des étages, tandis que la seconde partie progresse normalement et un certains nombres de bulles (opérations vides) sont insérées dans le pipeline à partir de l'étage j. Les différentes actions peuvent maintenant être désynchronisées par rapport au flux précédent (sans *stall_j*), chaque proposition atomique peut être retardée par un gel. De plus, les étages qui contiennent une bulle ne vérifient pas le même ensemble de propriété que précédemment. La transformation des formules de la classe **CPI** est telle que chaque proposition atomique peut maintenant être retardée.

Théorème 5 Préservation du préfixe
Pour toute formule φ de la classe **CPI**, *soit j l'étage le plus petit tel que $stall_j \in F'_s$.*

$$M_i \models \varphi \Leftrightarrow M_{i+1} \models \varphi'$$

avec φ' telle que pour toute proposition atomique ϕ_l de f
- *$\forall\varphi$ portant sur des ϕ_l pour $l \geq j$ alors ϕ_l est remplacée par* $\mathbf{AF}\phi_\mathbf{l}$
- *sinon ϕ_l reste inchangée.*

Preuve Le second point est obtenu directement par la propriété 4.8 de bisimulation faible des préfixes. Le premier point vient de la progression bégayante du préfixe (propriété 4.2) : aucune donnée ou commande ne sera détruite et elles finirons toujours leur traversée du pipeline. La proposition atomique sera simplement retardée.

■

Pour la classe de formules **CPE** concernant les comportements globaux du système il y a une préservation des propriétés lorsque le composant est construit uniquement à partir d'incréments *stall*.

Théorème 6 *Toute formule de la classe* **CPE** *est préservée dans un modèle obtenu par composition d'incréments de bégaiement.*

Preuve C'est la conséquence directe de la propriété 4.2. ■

Théorème 7 *Soit un automate M_{i+1} obtenu après l'application d'incréments de destruction à partir de l'automate M_i. Soit une formule f de la classe interne* **CPI** *portant sur un ou un ensemble d'étage [k,l]. Si il n'existe pas d'incrément $kill_i$ tel que $k \leq l$ alors*
$M_i \models f \Rightarrow M_{i+1} \models f$

Preuve Conséquence directe de la propriété de bisimulation 4.4 : lorsque l'événement evt_kill_i est actif, l'étage i est détruit mais le flux continue sa progression. ■

Les propriétés de la classe **CPI** ne sont pas préservés dans le cas d'incrément *kill*. Pour la classe de formule **CPE**, un incrément *kill* à un étage i, détruit la commande et par conséquent aucun résultat ne sera produit. En revanche, l'ensemble des données qui n'ont pas été touchées par le *kill* produiront un résultat.

Théorème 8 *Soient φ une formule de la classe* **CPE** *et $KILL$ l'ensemble des étages pour lesquels un événement kill est défini. Les formules sont transformées selon les règles suivantes :*

(p) $\phi_{i,a} \Leftrightarrow kill_qt_i \wedge \phi_{i,a}$

(f_p) $\phi_{0,a} \Rightarrow \mathbf{AF}\phi_{n-1,a} \Leftrightarrow$
$(kill_qt_0 \wedge \phi_{0,a}) \Rightarrow \mathbf{A}[\bigwedge_{i \in KILL} kill_qt_i \mathbf{U} \bigvee_{i \in KILL}(kill_act_i \wedge \phi_{i,a}) \vee \phi_{n-1,a}]$

Preuve Ce théorème est la conséquence des propriétés 4.4 et 4.11. ■

L'architecture particulière du contrôle de flux permet d'obtenir des résultats plus fins sur la transformation et la préservation des propriétés CTL. De plus, le découpage en incréments de la méthode incrémentale s'applique naturellement à la conception de telles architectures. Cette méthode est utilisée pour l'enseignement de l'architecture du MIPS R3000 pipeline [PH04] en première année du master d'informatique de l'université Pierre et Marie Curie [Uni] : on commence par expliquer ce qu'est un flux pipeline dans le cas optimal, puis petit à petit, on ajoute les événements qui peuvent perturber le flux. Ce cours s'appuie sur un modèle du microprocesseur décrit en VHDL comportemental et structurel de la chaîne de CAO ALLIANCE [all]. Il se trouve que la conception de ce MIPS et son enseignement reposent sur la méthode incrémentale.

Chapitre 5

Application pour la conception et la vérification incrémentales de convertisseurs de protocoles

Dans ce chapitre, nous appliquons la méthode incrémentale à la conception de convertisseurs de protocoles aussi appelés wrappers. L'apport de notre méthode est double, d'une part elle est une aide à la conception du composant en procédant étape par étape. D'autre part, elle facilite l'écriture de sa spécification et la vérification par model checking. La conception de SoC se fait le plus souvent par réutilisation d'IP déjà existants. Ces IPs sont conçus indépendamment des autres composants du système et ils possèdent tous une interface permettant de communiquer avec les autres composants. Aujourd'hui, les interfaces des composants sont pour la plupart des protocoles standardisés. Malheureusement, les composants n'implémentent pas tous le même standard. La connexion entre différents protocoles est une tâche difficile et souvent elle comporte un risque d'erreur important. Les IPs et les médias de communications sont pour la plupart pré-vérifiés, la vérification de l'ensemble du système consiste alors à vérifier les interactions entre les différents composants. Les convertisseurs de protocoles qui réalisent le lien entre les différents acteurs du système doivent être garantis sans erreur pour ne pas bruiter la tâche de vérification des interactions du système complet.

On trouve dans la littérature, de nombreux travaux sur la synthèse automatique des convertisseurs de protocoles. Ces méthodes garantissent l'adaptation correcte entre deux protocoles de communication. R. Passerone et al. [PRSV98] proposent un algorithme qui construit une machine d'états réalisant l'interface entre deux protocoles de communication différents. Les deux protocoles de communication sont définis par des expressions régulières, puis transformés en automate d'états finis. L'algorithme construit alors le produit des deux automates représentant l'intersection des langages des deux automates. La consistance des chemins est assurée par des règles de cohérence sur les transferts de données. Ces travaux ont été étendus dans [PdAHSV02], en plus de l'automate produit, il est possible d'ajouter une spécification sous forme d'automate. Ce dernier caractérise le convertisseur en spécifiant la taille des buffer, l'ordre des événements, etc ... C'est cette spécification qui garantit l'absence de faux chemin dans l'automate du convertisseur. Les machines d'états finis ainsi obtenues ont une taille exponentielle sur le nombre de symboles

dans l'expression régulière. Selon le niveau de détails des deux protocoles les automates d'interface [dAH01] peuvent ainsi exploser en nombre d'états. Cette démarche a été étendue aux systèmes ouverts par de Alfaro et al. [dAdSF+05], et automatisée par l'outil TICC [AdAdS+06].

Notre approche de conception est différente de celle proposée par Passerone et al., nous essayons de rester proches des méthodes utilisées par les concepteurs d'architectures matérielles. En pratique, la spécification des différents composants est définie sur un document qui décrit les signaux d'interface, les chronogrammes des transactions et des transferts de données. Le plus souvent, les différentes transactions sont présentées sans tenir compte des éventuelles subtilités dûes au traitement des cas particuliers. Les cas où, une transaction se passe mal ou la description du comportement lorsque le signal "reset" est activé, sont décrits en langage naturel. Le concepteur commence donc par écrire un convertisseur de protocole en se calquant sur les comportements de base, puis ajoute petit à petit les différents traitements de ces cas "exceptionnels".

Pour la vérification formelle de composant matériel, le problème de départ est la traduction du document de spécification en formules de logique temporelle. Cette traduction en propriétés CTL est une élaboration longue et complexe. En pratique, une spécification est décrite par des centaines de propriétés CTL. La réécriture du cahier des charges en spécification formelle est au coeur de la méthode B [Lan96]. Les concepts de B sont d'excellents outils pour mener à bien ce travail de réécriture.

Un autre problème lié à la traduction d'une spécification, vient du fait que ce travail est réalisé par un être humain qui peut faire des erreurs. Ces erreurs ne peuvent être détectées qu'à la fin du processus de vérification. En effet, lorsque le système ne vérifie pas une propriété, le concepteur doit pouvoir déterminer si l'erreur vient du composant ou bien de la propriété CTL. De plus, souvent le travail de vérification et par conséquent l'élaboration d'une spécification, n'est pas réalisée par le concepteur. Cela peut être une difficulté supplémentaire et une autre source d'erreur dans l'écriture de la spécification.

La méthode de conception incrémentale permet d'alléger le travail de spécification d'un composant. Nous avons vu dans le chapitre précédent qu'une partie de la spécification d'un composant M_{i+1} peut être automatiquement obtenue à partir de la spécification d'un composant M_i et dans certains cas de la spécification de l'incrément. Nous avons réalisé un outil [Bra03] qui automatise la transformation des formules CTL à partir des règles du chapitre 3.

La méthode incrémentale a été utilisée pour la conception de deux types de wrappers. Tous deux relient un composant respectant la norme VCI (Virtual Component Interface [On-00]) soit avec le bus PI (Peripheral interconnect bus [Ope94]), soit avec le bus AMBA (Advanced Microcontroler Bus Architecture [ARM99]). Ces derniers ont été réalisés par Joël Ossima Kheba [Oss03] en système C, afin d'enrichir la bibliothèque de composant SoCLib [soc]. Ils ont été vérifiés par simulation et n'ont pas été spécifiés par des formules CTL. Ce chapitre expose la conception des wrappers VCI/PI [Oss02] et l'évolution de la spécification au cours de sa conception. Des modèles de ces wrappers ont été réalisés en Verilog et des propriétés CTL ont été vérifier par model checking avec l'outil VIS [gro96].

5.1 La Plate-forme d'expérimentation

5.1.1 Les protocoles VCI et PI

La norme VCI a été réalisée par le consortium VSI (Virtual Socket Interface Alliance [vsi]). Elle a pour but de faciliter la réutilisation de composant virtuel en définissant un standard pour l'interface des composants et trois modes de transaction pour la communication entre les composants. Cette norme définit trois familles de communication point à point : peripheral (PVCI), basic (BVCI) et advanced (AVCI). Dans nos travaux nous avons utilisé des interfaces BVCI. La communication se joue entre deux acteurs : un *initiateur* de communication et une *cible* qui répond si elle le peut. Le protocole de communication est dit "split" : l'émission des requêtes (de l'initiateur vers la cible) est indépendante de la réception des réponses (de la cible vers l'initiateur) (fig. 5.1). La seule contrainte est que l'ordre de réception des réponses doit être le même que l'ordre d'émission des requêtes. Une transaction VCI est composée de paquets requêtes et de paquets réponses. Les paquets peuvent être composés d'une chaîne de cellules élémentaires, un indicateur marquant la fin d'une chaîne (signal *eop* : *end of paquet*) est positionné lors de l'envoi de la dernière cellule. La norme VCI est une interface de communication standard qui ne définit pas de support physique de transfert.

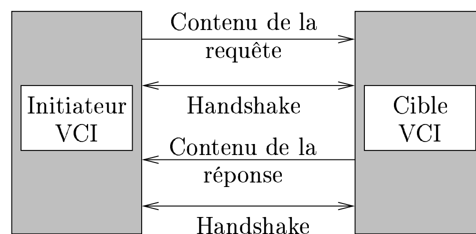


Fig. 5.1 – Protocole BVCI

Le protocole du Pi bus définit une connexion point à point entre deux agents : un *maître* et un *esclave*. Il a été standardisé dans le cadre du projet européen OMI en 1995 (Open Microprocessor Initiative [Pev95]). Le PI-bus est un bus intégré sur puce à grande vitesse, il permet des échanges rapides entre, par exemple, un microprocesseur et la mémoire. Un transfert sur le bus est un échange de une ou plusieurs données entre le maître et l'esclave sélectionné. Les échanges peuvent être atomiques ou en rafale. Dans le cas de rafale, le protocole d'échange est pipeliné : au premier cycle l'adresse de la donnée est envoyé, au second cycle le maître reçoit la donnée réponse (cas lecture) ou envoie la donnée (cas écriture) et envoie l'adresse du transfert suivant (fig. 5.2). Le PI-bus contient un arbitre de bus dans le cas de plate-forme multi-maîtres, mais aucun détail sur l'arbitrage n'est donné dans la spécification. C'est aux concepteurs de choisir la politique d'arbitrage. Le protocole du PI-bus fait partie des benchmarks Texas-97 [tex] et fait partie des exemples d'architectures matérielles vérifiées à l'aide du model checker VIS [gro96].

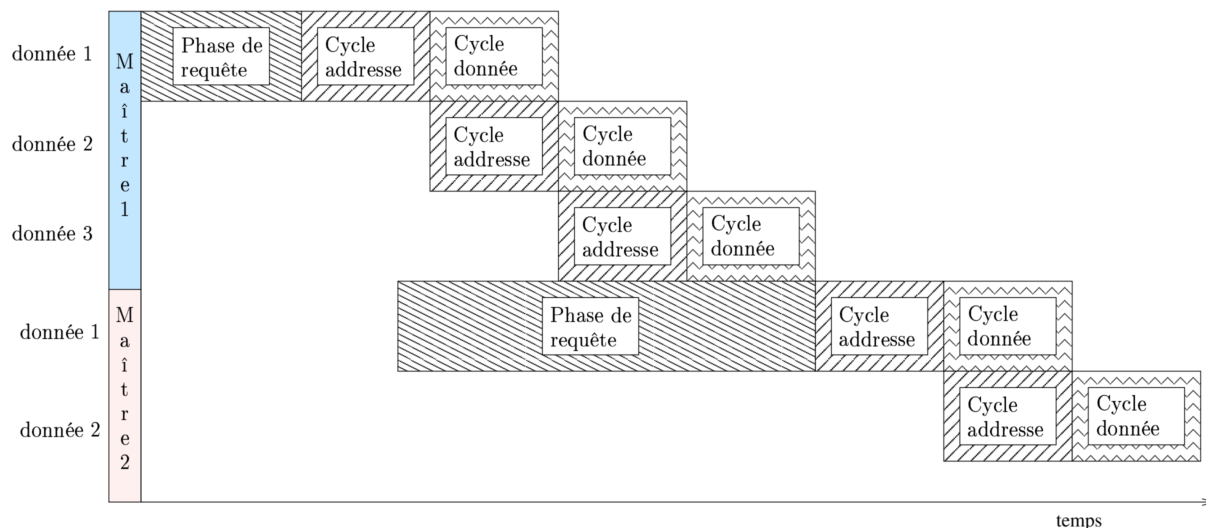
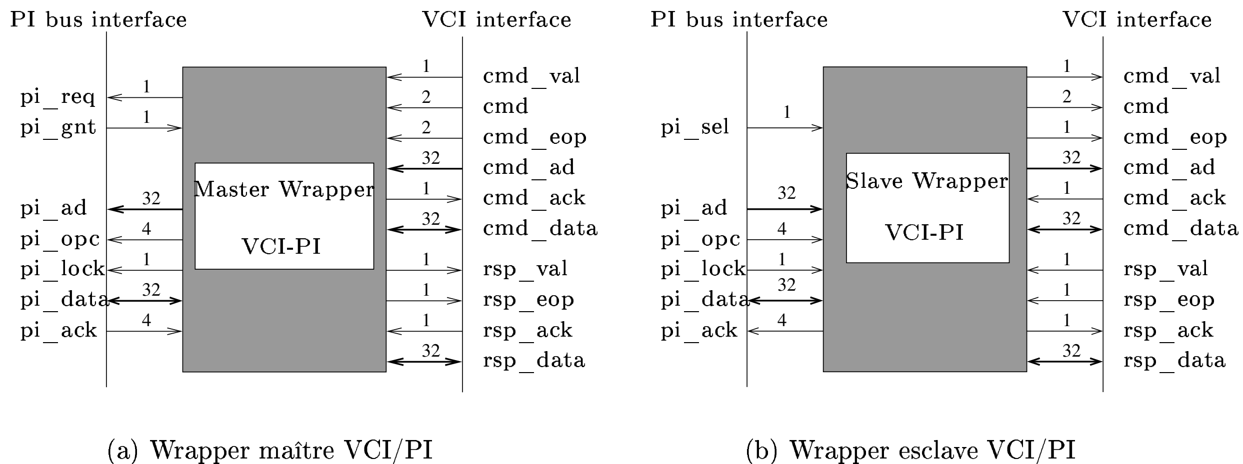


Fig. 5.2 – Protocole du PI-bus

5.1.2 La plate-forme

Notre travail s'est concentré sur le développement des wrappers VCI/PI qui permettent la communication entre les différents protocoles. Chacun des protocoles précédents distingue les initiateurs de transfert et les composants qui répondent à une requête. Nous avons défini deux wrappers différents. Un wrapper "maître" qui joue le rôle du maître PI et fait le lien entre l'initiateur VCI et le bus ; un wrapper "esclave" qui joue le rôle de l'esclave VCI et relie le PI-bus et la cible VCI. La figure 5.3 présente les signaux d'interface des deux wrappers.



(a) Wrapper maître VCI/PI

(b) Wrapper esclave VCI/PI

Fig. 5.3 – Interface des wrappers VCI/PI

Les wrappers sont transparents : du point de vue de l'initiateur VCI le wrapper maître joue le rôle de la cible et réciproquement, du point de vue de la cible VCI, le wrapper esclave joue le rôle de l'initiateur. La figure 5.4 décrit notre plate-forme d'expérimentation

avec un unique maître VCI et une unique cible VCI. Le transfert de traduction d'un paquet VCI est aussi illustré sur cette figure et procède de la manière suivante :

(1) L'initiateur VCI envoie une requête au wrapper maître.

(2) Le wrapper maître demande l'accès au bus via l'arbitre de bus (BCU).

(3) Le BCU donne l'accès au wrapper maître (3a) et sélectionne le wrapper cible (3b).

(4,5) Le wrapper maître envoie chaque cellule VCI au wrapper cible via le bus PI (en deux phases : adresse/données).

(6) Le wrapper esclave traduit la cellule PI en une requête VCI et la transmet à la cible VCI.

(7) La cible VCI répond au wrapper cible.

(8,9) Le wrapper cible transmet la réponse du wrapper maître via le bus PI.

(10) La réponse PI est traduite en réponse VCI et est ensuite transmise à l'initiateur VCI.

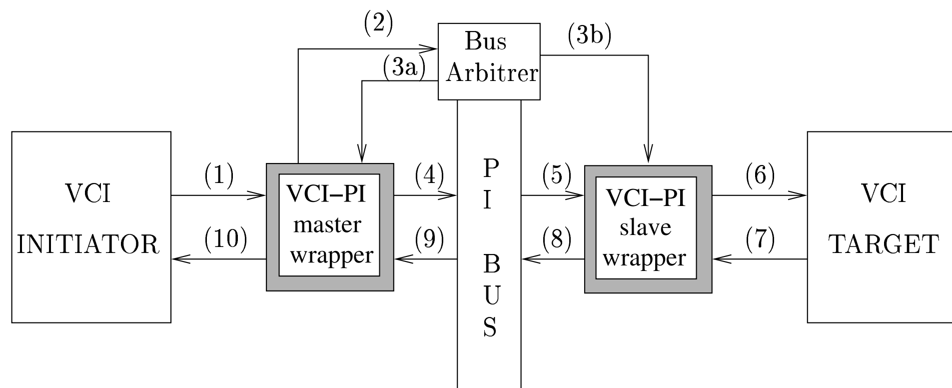


Fig. 5.4 – Illustration d'un transfert sur la plate-forme de traduction VCI-PI-VCI

Dans certains cas, comme le cas de requêtes d'écriture pour la cible VCI, le wrapper cible anticipe les réponses, autorisant l'envoi des cellules subséquentes.

5.2 Hierarchie des wrappers VCI/PI

5.2.1 Le modèle initial

Deux types de wrappers VCI/PI ont été élaborés : les *wrappers maîtres* et les *wrappers esclaves*. Dans ce chapitre, nous nous restreindrons aux wrappers maîtres, la conception des wrappers esclaves est décrite dans [Oss02].

En suivant notre démarche de conception incrémentale, nous avons défini une hiérarchie de neuf wrappers maîtres de **A** à **C"**. Le wrapper maître **A** est le cas le plus simple, c'est la brique de base de notre conception. Dans ce cas, nous faisons l'hypothèse que l'environnement se comporte de façon idéale : toutes les requêtes d'écriture ou de lecture au PI bus et toutes les requêtes réponses à l'initiateur VCI seront satisfaites en un cycle. Le fonctionnement du wrapper se décompose en deux parties (fig 5.5). La prémisse, qui attend une demande de l'interface VCI, demande et attend l'accès au bus. La seconde

partie correspond à l'échange de donnée proprement dite. Dans le but d'utiliser au maximum la fonctionnalité pipeline du PI bus, l'échange de donnée a un fonctionnement en trois phases réalisées par un pipeline, décrit de la manière suivante :

étage 0 [AD] Envoie l'adresse de la requête k sur l'interface PI et reçoit la requête (k+1) sur l'interface VCI ;

étage 1 [DT] Envoie ou reçoit la donnée PI correspondant à la (k-1)^{e} requête VCI ;

étage 2 [RSP] Envoie la réponse sur l'interface VCI correspondant à la (k-2)^{e} requête VCI.

Le protocole VCI envoie dans une cellule l'adresse et la donnée en même temps, alors que le protocole du PI bus décompose ce transfert en deux cycles. C'est pourquoi le premier étage du pipeline du wrapper reçoit l'adresse et la donnée $k+1$ en même temps qu'il envoie l'adresse $k-1$. La fin d'une transmission est marquée par l'activation du signal cmd_eop lors de l'envoi de la dernière cellule du paquet requête.

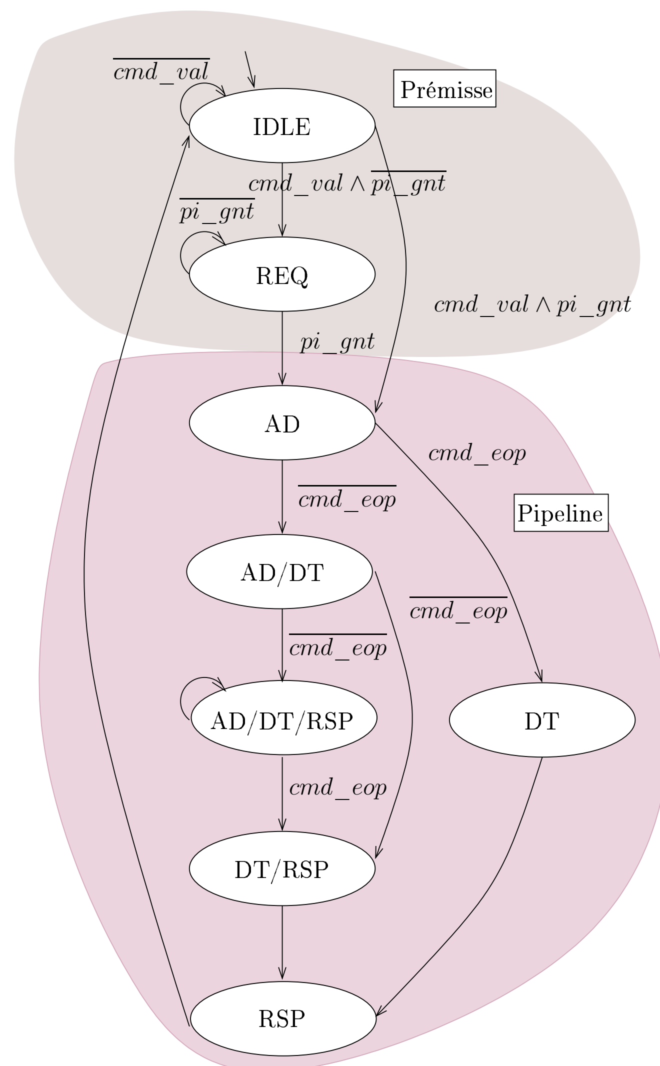


Fig. 5.5 – Automate du wrapper maître (flot optimal)

Le chemin de données est constitué d'une dizaines de registres et d'un décodeur qui permettent de décomposer les requêtes VCI dans un format adapté au protocole du PI bus (fig. 5.6). On peut remarquer le découpage des différents étages, chacun séparé par une barrière de registre.

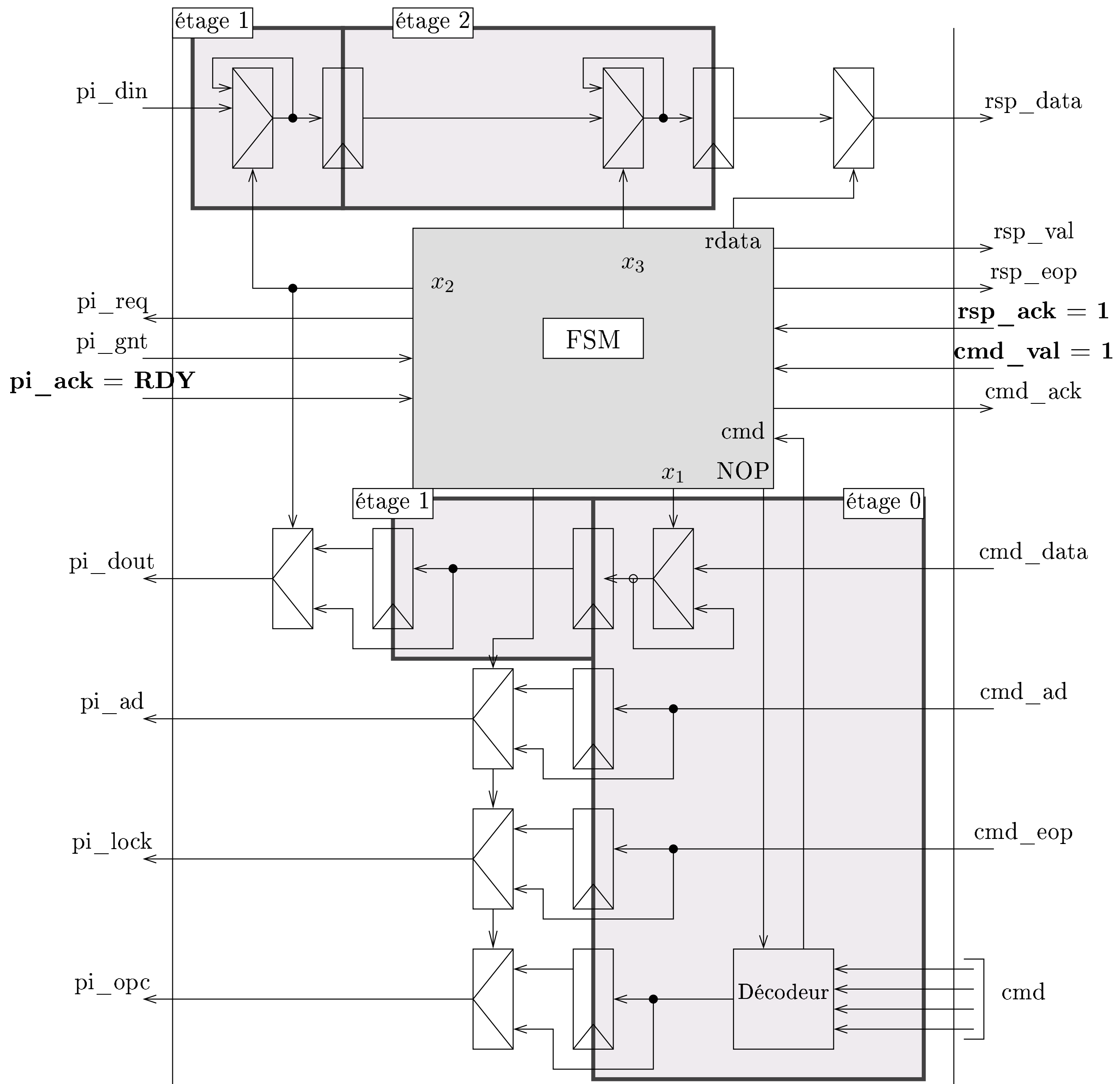


FIG. 5.6 – Chemin de donnée du wrapper maître

5.2.2 Les différents événements compatibles

Le comportement optimal n'admet aucun événement qui peut ralentir la progression du pipeline. Les différents événements que nous allons considérer sont ceux qui perturbent le flux du pipeline. Ils s'expriment par l'intermédiaire des signaux suivants :

- pi_ack : acquittement du wrapper cible ($\mathcal{D}om(pi_ack = \{RDY\}$ dans le cas optimal).
- cmd_val et rsp_ack : transaction et acquittement de l'initiateur VCI ($\mathcal{D}om(cmd_val) = \{0\}$ et $\mathcal{D}om(rsp_ack) = \{0\}$ dans le cas optimal.
- $reset$: l'initateur VCI ré-initialise le wrapper VCI/PI (n'existe pas dans le cas optimal).

Ils peuvent se décomposer en deux classes d'événement. Les événements sur l'interface VCI et sur l'interface PI.

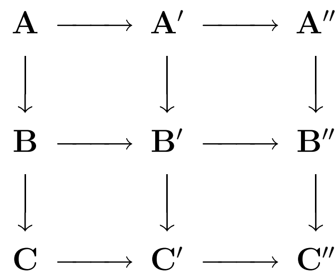
- Interface initiateur VCI :
  1. L'initiateur peut insérer des cycles d'attentes lors de l'émission des chaînes de cellules requête et/ou dans l'envoi des accusées réception des cellules de réponse ($\mathcal{D}om(cmd_val) = \{0, 1\}$ et $\mathcal{D}om(rsp_ack) = \{0, 1\}$).
  2. L'initiateur peut demander la ré-initialisation du wrapper($\mathcal{D}om(reset) = \{0, 1\}$).
- Interface bus PI
  3. L'esclave peut répondre qu'il n'est pas prêt à traiter la nouvelle requête ($\mathcal{D}om(pi_ack) = \{RDY, WAIT\}$).
  4. L'esclave peut interrompre la transmission en cours et imposer la retransmission de la dernière requête ($\mathcal{D}om(pi_ack) = \{RDY, WAIT, RET\}$).

A partir de ces événements un ensemble de 9 wrappers a été développé, du plus simple (wrapper **A**) au plus complexe (wrapper **C"**). La hiérarchie est présentée dans le tableau 5.1. Chaque wrapper de cette hiérarchie est tel que chaque wrapper plus complexe inclut les comportements des wrappers précédents (tab. 5.2). C'est une conséquence directe de notre méthode de conception. Cela ne veut pas dire que le même incrément s'applique du wrapper **B** au wrapper **B'** et du wrapper **C** au wrapper **C'**. En effet, c'est le même événement qui est considéré, mais il ne s'applique pas sur le même automate. Les différents incréments à ajouter dépendent du wrapper sur lequel ils s'appliquent.

Événement considéré	Optimal	1	2
Optimal	**A**	**A'**	**A"**
3	**B**	**B'**	**B"**
4	**C**	**C'**	**C"**

Tab. 5.1 – Hiérarchie des wrappers maîtres VCI/PI

les wrappers de **A** à **C'** ont été réalisés et vérifiés par simulation [Oss02] avec CASS [Pét03]. Nous avons implémenté la plate-forme 5.4 contenant les wrappers de **A** à **B'** en *Verilog synchrone*. Les modèles **A"**, **B"** et **C"** sont des extensions au signal reset et n'ont pas été implémentés.



Tab. 5.2 – Inclusion des wrappers maîtres VCI/PI

5.3 Conception incrémentale des wrappers VCI/PI

En suivant le bon sens du concepteur, nous allons suivre les étapes suivantes : **A** → **B** → **B'** → **C'** → **C"**. L'ensemble des automates incrémentés se trouve en annexe B.

5.3.1 Incrément 1 (de A à B)

Cet incrément réalise la gestion de l'événement 3. Il correspond à l'extension du domaine de définition du signal *pi_ack* sur l'interface du PI-bus avec la valeur $WAIT$. Ce signal permet à l'esclave d'exprimer le fait qu'il n'est pas prêt à répondre à une requête du maître. Le chronogramme 5.7 montre l'évolution des signaux sur l'interface PI du wrapper lors de la gestion de ce nouvel événement dans le cas d'une écriture, comme indiqué dans la spécification du protocole du PI-bus.

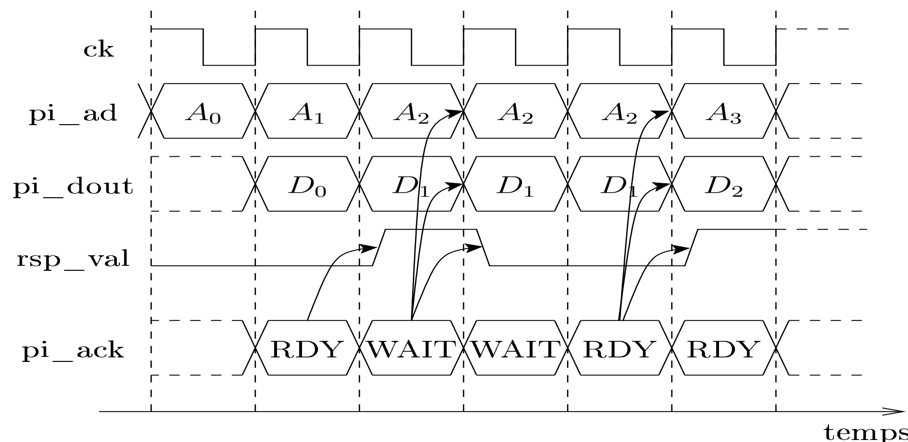


Fig. 5.7 – Chronogramme de la gestion du signal WAIT

Lorsque le signal *pi_ack* est positionné à la valeur $WAIT$ le transfert de donnée ne progresse plus : le wrapper maître maintient l'adresse A_2 et la donnée D_1 sur le bus. Tant que le signal *pi_ack* est à $WAIT$ le wrapper n'accepte plus de nouvelles données de l'initiateur VCI et ne positionne pas le signal de réponse *rsp_val*. Les étages 0 et 1 du pipeline ne progressent plus. La gestion du nouvel événement ajoute donc un bégaiement du comportement du maître sur le préfixe composé des étages 0 et 1. L'événement

correspond donc à un incrément bégayant qui gèle le pipeline à partir de l'étage 1. Le wrapper **B** peut maintenant gérer les cas où la cible n'est pas prête à envoyer ou recevoir une donnée.

5.3.2 Incrément 2 (de B à B')

Cet incrément est implémentation de l'événement 1, il est réalisé par la composition de deux événements. Les nouveaux événements considérés par cette composition se trouve sur l'interface VCI du wrapper maître. Les nouveaux événements s'expriment par le biais de deux signaux de l'interface VCI, le signal *cmd_val* et le signal *rsp_ack*. Leur domaine de définition est étendu, ils peuvent tous deux prendre la nouvelle valeur 0, qui exprime le fait que l'initiateur n'est pas prêt à émettre ou à recevoir de nouvelles données (il n'acquitte pas la réponse). Les comportements induits par ces événements correspondent aux gels du pipeline à l'étage 0 et à l'étage 2. L'incrément considéré est un incrément de bégaiement $STALL_{0,2}$, les nouveaux états et les nouvelles transitions correspondant au traitement des gels sont réalisés comme nous l'avons décrit dans la section 4.2. Les changements sur l'interface VCI impliquent des changements sur les signaux de sorties de l'interface PI. En effet, lorsque l'initiateur n'est pas prêt à émettre et qu'il bloque l'étage 1 du pipeline, au coup suivant l'étage 2 doit effectuer une opération vide. Cette opération correspond au fait qu'aucune adresse n'a été envoyée donc on ne peut ni lire ni écrire une donnée correspondant à cette adresse (bulle insérée à l'étage 2). Cela s'exprime par l'extension du domaine de définition du signal correspondant au code de l'opération de la requête envoyée sur le PI-bus qui peut maintenant prendre la valeur NOP. Cette nouvelle valeur permet de ne demander ni écriture ni lecture sur le PI-BUS.

5.3.3 Incrément 3 (de B' à C')

Cet incrément est aussi l'extension du domaine de définition du signal d'acquittement des commandes sur le PI-bus : *pi_ack*. Le signal peut maintenant prendre la valeur $RETRACT$. Cette valeur indique que la cible demande la réémission de la transaction en cours. Lorsque ce nouvel événement survient, le wrapper maître doit se comporter de la manière suivante :

- Arrêter la requête en cours ;
- Relâcher le bus ;
- Demander à nouveau l'accès au bus ;
- Réémettre la requête de façon atomique (sans pipeline).

La définition de cet incrément s'est fait par la description sous forme d'automate de la gestion du retract. Le retour aux anciens comportements est clairement défini par la fin d'un retract et la réception de l'acquittement du paquet réponse réémis. Sur l'interface VCI l'initiateur voit seulement le non-acquittement de ces paquets requêtes. La gestion d'une réémission de paquet est complètement réalisée par le wrapper et transparente pour l'initiateur VCI.

5.3.4 Incrément 4 (de C' à C")

Cet incrément est une composition de trois incréments *kill*, un pour chaque étage du pipeline. Le nouvel événement est un nouveau signal *cmd_reset* , c'est à dire que lorsque cet événement est actif tout ce qui a été commencé doit être arrêté immédiatement.

- Les commandes dans le pipeline du wrapper sont détruites ;
- Le wrapper envoie un signal de réinitialisation au PI-bus pour stopper la transaction en cours.

Pour chaque état existant dans le modèle initial, toutes les transitions sont maintenant étiquetées avec la valeur muette du *reset* et de tous les états il existe une transition étiquetée par la valeur active du reset dans lequel l'événement est traité. A noter qu'il y a deux traitements différents, un pour la prémisse du pipeline et un pour le pipeline en lui-même. Le nouvel événement induit un nouveau signal sur l'interface du PI-bus, le signal *pi_resetn*.

5.4 Évolution de la spécification des wrappers VCI/PI

Dans cette section nous donnons un exemple de propriétés automatiquement transformées grâce aux résultats de transformations présents aux chapitres 3 et 4. Nous présentons trois propriétés appartenant à la spécifications des wrappers **B** et nous montrons leur transformation lorsque l'on incrémente les wrappers afin d'obtenir des wrappers **B'**.

Le système a été vérifié à l'aide du model checker VIS [gro96]. Nous avons écrit 80 propriétés CTL pour les wrappers maître et esclave **B** et le système entier. Voici des exemples de propriétés CTL (non-transformées) qui ont été vérifiées sur **B**.

La première propriété spécifie le protocole de communication entre le wrapper maître et le contrôleur de bus. L'implémentation du wrapper cible est composée de deux automates, la propriété 2 spécifie que les deux automates sont bien synchronisés. Les deux automates commencent et terminent leur traitement en même temps. La troisième propriété décrit que l'acquittement du wrapper maître vers l'initiateur VCI est suspendu tant que la cible n'est pas prête.

```
#-----------------------#
# -> : implies operator #
# *  : and operator     #
# +  : or operator      #
#-----------------------#

# Check the interface between
# the PI bus arbiter and
# the master wrapper.
# property 1: #
AG( (wrap0.state = R_REQ) ->
   (A( (m_pi_req0 = 1) U (m_pi_gnt0 = 1))));
```

```
# Check the behavior of the slave wrapper
# (its two automatas are well synchronized).
# property 2: #
!EF((wrap_cible.cmd_cible.state = CMD_IDLE) *
    !(wrap_cible.rsp_cible.state = RSP_IDLE));

# Check the waiting state of wrapper B
    AG((m_cmd_val0 = 1 * m_cmd_eop0 = 0 ) ->
           A(m_cmd_ack0 = 1 U
                   (A(     m_pi_ack0[2:0] = 0 U
                           m_cmd_ack0 = 0 * m_pi_ack0[2:0] = 3))));
```

Le wrapper **B** a été incrémenté afin d'obtenir un wrapper **B'** et les autres composants ont été modifiés pour pouvoir générer et gérer les nouveaux événements possibles. Le nouvel événement étend le domaine de définition des signaux *cmd_ val* et *rsp_ ack*, ils peuvent maintenant prendre la valeur 0 qui impose d'attendre que l'initiateur soit prêt pour continuer le transfert. Le nouvel événement est défini tel que $e = \langle (inc_{b'}, \{0,1\}), \{0\}, \{1\} \rangle$; l'ensemble des configurations muettes : $C_{QT}(inc_{b'}) = \{0\}$, cela correspond à la configuration suivante $(cmd_val = 1) \wedge (rsp_ack = 1)$; l'ensemble des configurations actives : $C_{ACT}(inc_{b'}) = \{1\}$, elle se produit lorsque $(cmd_val = 0) \vee (rsp_ack = 0)$.

La première propriété porte sur la prémisse du pipeline. Cette partie n'est pas concernée par l'incrément. D'après le théorème 6, cette propriété fait directement partie de la spécification du wrapper B'. La seconde propriété concerne le wrapper cible B, cette propriété fait aussi directement partie de la spécification du wrapper cible B'. La troisième propriété est transformée par le théorème 1 de la façon suivante :

```
  A( (m_cmd_val0 = 1 * m_cmd_eop0 = 0 ) ->
A(m_cmd_ack0 = 1 U (incb'_act * m_cmd_ack0 = 1) +
 (A (m_pi_ack0[2:0] = 0 U (incb'_act * m_pi_ack0[2:0] = 0) +
m_cmd_ack0 = 0 * m_pi_ack0[2:0] = 3))) W
(incb'_act * (m_cmd_val0 = 1 * m_cmd_eop0 = 0 ) ->
A(m_cmd_ack0 = 1 U (incb'_act * m_cmd_ack0 = 1) +
 (A (m_pi_ack0[2:0] = 0 U (incb'_act * m_pi_ack0[2:0] = 0) +
m_cmd_ack0 = 0 * m_pi_ack0[2:0] = 3)))));
```

L'outil CTLINC, implémentant les algorithmes de transformations décrits aux chapitres 3 et 4 a été appliqué au 80 propriétés CTL. Les propriétés transformées ont été aussi vérifiées sur la plate-forme contenant les wrapper **B'**. Une partie des propriétés de B est fausse sur la plate-forme contenant le wrapper **B'**. Nous avons pu vérifier avec succès les propriété transformées, cela illustre bien que les modifications de **B** à **B'** n'introduisent pas de régression. Dans la pratique, cette phase de vérification des propriétés transformées n'est pas à faire. D'autant plus que les temps de vérification que nous obtenons rien que sur la plate-forme **B'** deviennent déjà très grands. Notre méthode de conception permet de sauter cette étape de vérification puisque nous avons montré dans les chapitres précédents que les propriétés sont vraies par construction.

Le tableau 5.4 présente les informations quantitatives du processus de vérification pour les trois propriétés décrites précédemment. Nous avons utilisé le model checker VIS. Une première étape calcule l'ensemble des états accessibles du système, puis une seconde étape vérifie la propriété. Les expériences ont été faites sur un Pentium IV à 3.20GHz avec 2Go de mémoire, et le calcul de l'ensemble des états accessibles a été effectué en utilisant l'algorithme de réordonnancement dynamique "sift".

Les différentes plates-formes sont composées des éléments suivants :

I. Un maître unique et un esclave de type **B** avec un initiateur VCI, une cible VCI, un PI bus (1M1SB).

II. Un maître unique et un esclave de type **B'** avec un initiateur VCI, une cible VCI, un PI bus (1M1SB').

III. Deux maîtres et un esclave de type **B** avec deux initiateurs VCI, une cible VCI, un PI bus (2M1SB).

IV. Deux maîtres et un esclave de type **B'** avec deux initiateurs VCI, une cible VCI, un PI bus (2M1SB').

Pour les plates-formes de petite taille (plates-formes I et II), le temps de vérification sur l'ensemble des propriétés augmente pour le modèle plus complexe. La différence de temps est occupée par le calcul de l'espace d'états accessibles réduits (pruned Reachable states). Cet ensemble est calculé à partir de l'espace d'états complet, il est ensuite réduit selon la propriété à vérifier. Ces résultats sont confirmés pour nos systèmes de taille moyenne (III et IV), où l'écart entre le temps de vérification des plates-formes **B** et **B'** est dû à l'augmentation de la complexité des deux systèmes.
La propriété 1 est plus longue à vérifier. Cela vient du fait que cette propriété est préservée entre les modèles B et B'. De plus, elle concerne les composants qui ont la plus forte augmentation de taille entre les deux modèles. En revanche la propriété 2 qui met en oeuvre les wrappers cibles qui n'ont pas été beaucoup modifiés par l'incrément a un temps de vérification court. Enfin, la propriété 3 a été transformée mais l'augmentationde temps de vérification est dûe à l'augmentation de la complexité des modèles plus qu'à la complexification de la formule. Cela vient du fait que lors de la vérification des propriétés sur **B'**, c'est la même partie de l'espace d'états (que **B**) qui est analysée, mais les BDD représentant la relation de transition et l'espace d'états de la plate-forme **B'** est beaucoup plus grand que ceux de la plate-forme **B**.

La conception incrémentale permet de ne pas relancer la phase de model checking pour l'ensemble des propriétés déjà vérifiées sur le modèle B, le gain en temps est considérable. Mais les temps de model checking obtenus pour l'ensemble de ces plates-formes restent encore trop grands.

Platform	FSM depth	Number of variables	BDD size (# of nodes)	Reachable states	Analysis time	Prop (#)	Pruned Reach. states	Checking time
I	475	289	34578	8.56e+06	50.61s	1	6.26s	1s
						2	5.9s	0.91s
						3	5.73s	1.53s
II	806	297	31404	1,11e+07	64.71s	1	35.31s	99.95s
						2	37.14s	4.27s
						3'	42.46s	4s
III	604	436	161846	3.10e+10	43min	1	222.01s	42min
						2	490.89s	11min
						3	363.65s	42min
IV	1086	447	225039	4.53e+11	1h00	1	261.88s	55min
						2	637.29s	12min
						3'	637.27s	44min

TAB. 5.3 – Temps de Vérification et mémoire requise

Deuxième partie

Formules CTL comme Abstraction de Composants

Chapitre 6

Démarche CEGAR pour la vérification de SoC

Dans la partie précédente, nous avons présenté une méthode de conception incrémentale qui permet d'une part de concevoir des composants matériels et d'autre part de faire évoluer la spécification tout au long des différentes phases de conception. Cette dernière technique prend toute son importance dans la conception de systèmes sur puce où la réutilisation des composants est essentielle. Pour permettre une réutilisation correcte de chaque composant, l'écriture de la spécification est cruciale.

Néanmoins, d'après les conclusions que nous avons pu tirer de nos expériences sur la plate-forme VCI/PI, la vérification de propriétés *globales* souffre toujours du problème d'explosion combinatoire en nombre d'états. Une propriété *globale* met en oeuvre l'ensemble des composants du système. Notre proposition pour améliorer la vérification de ce type de propriété est de prendre en compte la spécification de chacun des composants pour évaluer une propriété globale. Nous proposons d'abstraire chaque composant par un sous-ensemble de sa spécification.

La première idée qui avait été mise en oeuvre par Cédric Roux consistait à construire le plus petit système qui a les mêmes comportements que le système étudié *si on considère uniquement les signaux qui apparaissent dans la propriété*. La réduction de chaque composant préservait les signaux d'interface et les propositions atomiques de la propriété globale à vérifier. Cette réduction permet de vérifier toute propriété globale CTL. L'algorithme de Lin et Newton [LN91] a été utilisé pour cette réduction. Malheureusement, les composants ne peuvent pas être réduit indépendamment les uns des autres, et la réduction de chaque composant est faible. Les composants sont tous connectés et par conséquent la plupart des signaux d'interface sont conservés et chaque composant entre en jeu. La réduction ne permettait pas d'obtenir une abstraction assez petite.

Nous avons donc décidé de construire une abstraction plus grossière et plus simple à construire pour chacun des composants. La contrepartie est que l'interface des composants n'est pas préservée. Nous perdons alors la préservation de l'ensemble des propriétés CTL : les propriétés ACTL vraies sur le modèle abstrait seront préservées sur le modèle concret. En revanche, si une propriété n'est pas validée sur le modèle abstrait, nous ne pourrons pas en déduire que celle-ci est fausse sur le modèle concret. C'est pourquoi, nous avons intégré notre méthode d'abstraction dans le cadre d'une boucle de raffinement d'abstraction par

analyse de contre-exemple (CEGAR).

6.1 Abstraction de composants

L'abstraction est probablement la technique la plus importante pour réduire le problème d'explosion combinatoire des états. L'abstraction permet d'omettre ou de simplifier des détails de l'implémentation qui ne sont pas utiles pour la vérification d'une propriété donnée. La vérification du modèle abstrait est alors plus simple à réaliser. De nombreuses techniques de réduction du nombre d'états des systèmes ont été proposées pour réduire la taille des modèles. Le but est donc de remplacer le modèle concret par un modèle plus petit qui préserve les propriétés à vérifier. Le modèle abstrait peut contenir plus de transitions et d'états que le modèle concret mais sa représentation symbolique est plus petite. Une des difficultés est de trouver une abstraction plus petite mais qui garantit que les propriétés vraies sur le modèle abstrait sont aussi valables pour le modèle concret. Se pose alors le problème de précision d'une abstraction : faut-il construire un modèle abstrait qui est sur ou sous-approximation du système ? Il existe plusieurs méthodes qui définissent une abstraction telle que toutes les questions résolues sur l'abstraction sont résolues sur la concrétisation.

Clarke, Grumberg et Long [CGL94, Lon93] proposent une abstraction existentielle qui associe à chaque état abstrait, un ensemble d'états concrets. Le critère d'abstraction repose sur les domaines des variables du modèle concret (data abstraction). Les abstractions par prédicats [GS97], formalisent de manière plus générale les *data abstractions*. Ces deux techniques construisent une approximation du modèle directement à partir du code du système mais en supposant que le code est donné par des formules de logique du premier ordre (pour plus de détail sur ces techniques d'abstraction voir [Gru05]). Le modèle abstrait obtenu est une sur-approximation et par conséquent, les propriétés préservées font partie du fragment ACTL. L'*interprétation abstraite* [CC77], permet par analyse statique de la description du système d'extraire une abstraction à partir d'une *fonction d'abstraction* et d'une *fonction de concrétisation* [DGG97] données.

L'ensemble des techniques d'abstraction décrites précédemment sont conservatives pour la véracité d'une formule. C'est à dire que les formules vraies sur le modèle abstrait seront aussi valables pour le modèle concret. Mais à partir d'une formule non vérifiée sur le modèle abstrait, nous ne pouvons pas directement en déduire la validité de celle-ci sur le modèle concret. En effet, la sur-approximation de l'abstraction implique que le modèle abstrait a plus de comportements que le modèle concret. Il est possible d'aller plus loin si l'évaluation des formules est interprétée par une sémantique à 3 valeurs (logique de Kleene) [BG99]. Cette sémantique évalue une formule à vraie (**true**), fausse (**false**), ou indéterminée **unkwn**. La valeur indéterminée correspond à "on ne sait pas si la propriété est vraie ou fausse". Les modèles abstraits obtenus sont alors conservatifs pour **true** et **false**. En revanche, si la formule est indéfinie sur le modèle abstrait (**unkwn**) cela correspond à un manque d'information sur le modèle abstrait pour pouvoir statuer sur la validité de la propriété. Gurfinkel et Chechik [GWC06] utilisent une structure multi-valuée, où l'ensemble des états et l'ensemble des transitions sont sur 4-valeurs (logique de Belnap, [Fit91]). Cette représentation est plus générale que la représentation par 3-valeurs. Ils

ont réalisé un model checker χCheck [ECD+03] qui gère les quatre valeurs et permet la vérification de propriété CTL sur des programmes mono-thread.

6.2 Notre démarche d'abstraction à partir des formules CTL

Nous avons vu qu'il était difficile de vérifier des propriétés globales sur un système composé de plusieurs composants, alors que la vérification d'un unique composant (avec des hypothèses sur les comportements de l'environnement) semble une tâche beaucoup plus simple. De plus comme nous l'avons déjà énoncé, les SOC's sont la plupart du temps une composition d'IP's déjà vérifiés. L'idée ici est donc de construire une abstraction directement à partir des composants déjà vérifiés. Nous proposons un algorithme qui construit automatiquement une abstraction à partir d'un sous-ensemble de la spécification d'un composant. L'abstraction ainsi obtenue est la représentation du sous-ensemble de la spécification choisie.

Nous nous sommes inspirées des travaux de Xie et Browne présentés en 2003 dans [XB03]. Ils proposent une méthode de vérification compositionnelle basée sur cette idée dans le contexte de la conception logicielle. Dans leurs travaux, les propriétés temporelles (LTL) de chaque composant sont spécifiées, vérifiées et empaquetées avec le composant. Le système complet est alors vérifié en utilisant l'abstraction de chacun de ses composants, chaque abstraction est construite à partir des propriétés déjà vérifiées plus quelques hypothèses sur les comportements de l'environnement (défini dans la même logique temporelle). Le raffinement est alors fait par une boucle classique d'abstraction-raffinement guidée par contre-exemple. Le projet européen Prosyd [BJP05] regroupant des entreprises et des universitaires a pour objectif d'établir un standard intégrant le concept *property-based* pour la conception de système matériel. Ce concept tend à intégrer et unifier toutes les phases de développement, de la définition des besoins jusqu'à l'implémentation et la vérification, en un unique flot de conception cohérent. Leur flot de conception est construit sur le langage de spécification PSL/Sugar. Le groupe de Hans Eveking du TUD[SNBE06] adopte une approche similaire basée sur des propriétés PSL pour la génération d'abstractions conservatives. Ces méthodes sont utilisées pour la génération des moniteurs à partir de propriétés [MAB06]. Büttner [Büt05] de OneSpin Solutions [one] propose une génération d'abstraction dans le contexte de composition synchrone de modules. Ses modèles abstraits sont bien adaptés pour fournir une abstraction au cycle près, intéressante dans la vérification de micro-architecture. L'abstraction obtenue est alors synthétisée et embarquée dans un environnement de simulation matérielle.

Pour notre part, nous définissons aussi une abstraction de composant basée sur *la spécification du composant*. Les composants sont représentés par des structures de Kripke et leur spécification autant que les hypothèses sur les comportements de l'environnement sont exprimées par des propriétés CTL. Nous proposons un algorithme qui construit une abstraction du composant directement à partir d'un sous-ensemble de sa spécification. Notre abstraction est une traduction directe de formules CTL en structure de Kripke. Notre démarche de vérification suit les étapes suivantes illustrées par la figure 6.1 :

– Les composants sont déjà spécifiés et vérifiés.

- Une propriété globale est choisie pour être vérifiée sur le système entier.
- Pour chaque composant, un sous-ensemble de sa spécification est choisi selon la propriété globale à vérifier.
- Une première abstraction est construite pour chacun des composants.
- Toutes les abstractions sont composées afin d'obtenir une abstraction du système complet. Cette abstraction est l'abstraction initiale du processus CEGAR.
- Si l'abstraction doit être raffinée, une nouvelle formule d'un ou de plusieurs composants est sélectionnée et ajoutée à la nouvelle abstraction.

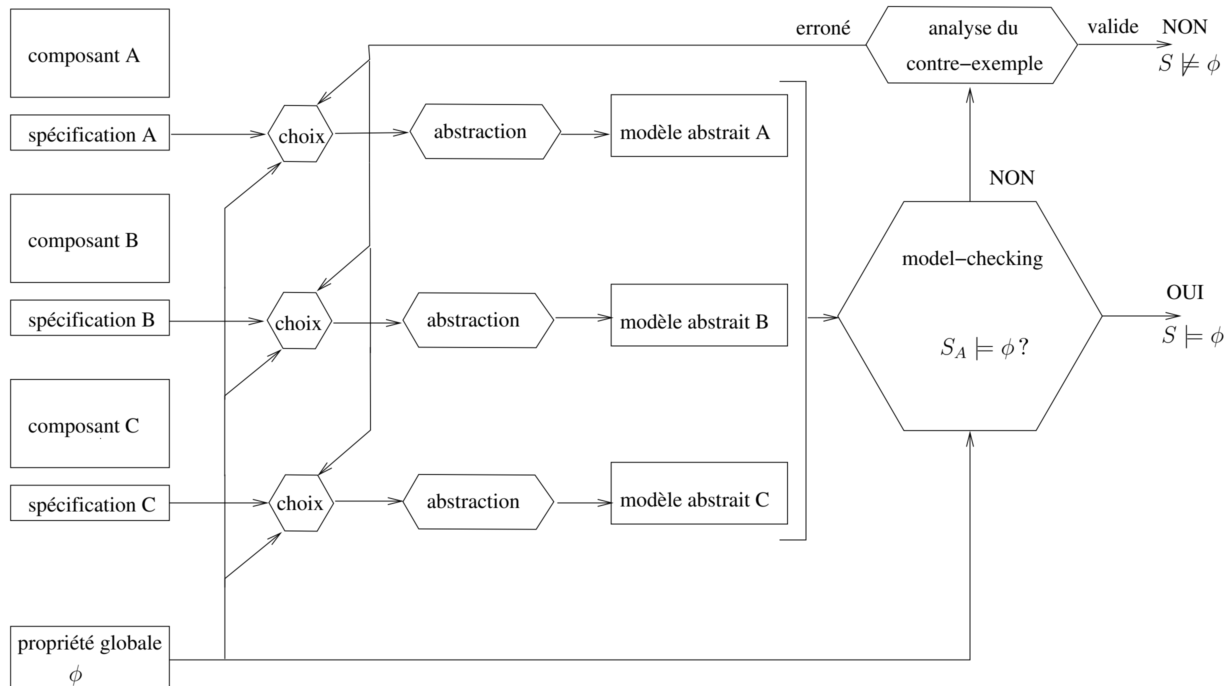


FIG. 6.1 – Notre boucle de CEGAR

La contribution majeure de ce chapitre est la définition d'un algorithme qui construit automatiquement une abstraction à partir d'un sous-ensemble de la spécification d'un composant. Peng et Tahar [PMT02] proposent de synthétiser le tableau de formules ACTL [GL91] en description Verilog. Notre travail va dans la même direction mais notre algorithme est basé sur une analyse syntaxique de la formule plutôt que sur la définition des points fixes pour chacune des sous-formules. De plus notre algorithme ne se restreint pas au fragment universel de CTL : nous pouvons synthétiser des structures pour CTL\X .

Plusieurs travaux étudient la traduction des logiques temporelles en automate. Kupferman et al.[KVW00] ont présenté une traduction linéaire des logiques temporelles arborescentes vers des automates d'arbres alternants. Leur méthode conduit à un algorithme de model checking basé sur les automates d'arbres alternants (*automata-theoretic based*) qui est linéaire en temps pour CTL. Plus récemment Dams et Namjoshi [DN05] ont proposé d'utiliser les automates d'arbre comme des abstractions pour tous les dépliages d'une structure de Kripke. Notre objectif est différent : nous voulons obtenir une abstraction

qui peut être insérée dans la plate-forme à vérifier. Nous allons montrer que le modèle de structure de Kripke abstraite que nous manipulons (proche des $\mathcal{L}$-valued structure de Kripke de [GC06]) est bien adapté pour un tel objectif : nous pouvons combiner de façon homogène les modèles concrets et les modèles abstraits.

Notre structure abstraite est évidemment moins précise que le modèle concret. Nous avons besoin de représenter moins d'information. Nous nous sommes inspirées du model checking multi-valué de [BG99, CDEG03, GC06] pour représenter la perte d'information, puis nous l'intégrons dans un model checker classique (2 valeurs).

6.3 Définition d'un composant abstrait

6.3.1 Structure de Kripke équitable

Dans notre approche, un composant est constitué d'une description comportementale et de deux ensembles de formules CTL : sa spécification et ses hypothèses sur l'environnement. En général une spécification est valide pour un composant lorsqu'un certain nombre d'hypothèses sur l'environnement sont aussi valides.

Définition 6.1 *Un* composant *est un triplet $C = \langle K, P, A \rangle$ où K est une structure de Kripke équitable (fair), P et A sont deux ensembles disjoints de formules CTL, représentant respectivement la spécification du composant et les hypothèses sur l'environnement.*

Dans le chapitre 1, nous avons défini les structures de Kripke, ici nous allons considérer des *structures de Kripke équitables* [CGP99]. En model checking, le plus souvent, nous nous intéressons uniquement à la validité d'une formule sur un chemin équitable. Par exemple, si l'on considère la plate-forme VCI-PI à un maître et un esclave, nous ne voulons pas prendre en compte les chemins où la permission du contrôleur de bus n'arrive jamais. Pour ôter ces chemins, nous ajoutons une *contrainte d'équité* aux comportements du modèle. Dans notre cas, les *contraintes d'équité* sont des conditions d'acception de Büchi généralisée.

Une structure de Kripke équitable est un 6-uplet $K = \langle AP, S, S_0, \mathcal{L}, R, F \rangle$, où $F = \{F_0, \dots F_n\}$ est un ensemble conditions d'acceptation de Büchi généralisées tel que $F_i \subseteq S$. Un chemin de K est équitable si pour tout i, F_i est visité infiniment souvent.

6.3.2 Structure de Kripke Abstraite (AKS)

Pour modéliser l'abstraction d'un composant, nous avons besoin de représenter des informations supplémentaires sur la validité d'une proposition atomique. L'idée est de connaître, pour chaque état , si une proposition atomique est vraie, fausse, *indéterminée* ou *inconsistante*. Une valeur indéterminée exprime l'absence de connaissance sur la validité d'une proposition atomique. Une valeur inconsistante peut être générée par l'abstraction. A la différence de [BG00] pour le model checking 3-valeurs, ou [CDEG03] pour le model checking 4-valeurs, nous n'introduisons pas de nouveaux symboles. Nous avons choisi d'étendre l'ensemble des propositions atomiques AP de la structure de Kripke. L'ensemble des littéraux Lit contient l'ensemble des propositions atomiques et leur *négation* : pour toutes propositions atomiques p appartenant à AP, p et $\overline{p}$ appartiennent à Lit.

Définition 6.2 *Soit AP un ensemble de propositions atomiques,* l'ensemble des littéraux *Lit est tel que :*
$Lit = AP \cup \{\overline{p} \mid p \in AP\}$.

Dans notre structure de Kripke abstraite, la fonction d'étiquetage n'est plus défini sur l'ensemble des parties de AP mais sur l'ensemble des parties de Lit. Cette approche est similaire au *dual-rail encoding* utilisé dans la conception de circuit asynchrone ([Spa01]). Evidemment, ici nous utilisons les quatre valeurs pour représenter l'information dans un état. Le tableau 6.1 récapitule l'information correspondant à chacune des quatre valeurs.

$p \notin \mathcal{L}(s) \wedge \overline{p} \notin \mathcal{L}(s)$	p est indéterminée dans s
$p \notin \mathcal{L}(s) \wedge \overline{p} \in \mathcal{L}(s)$	p est fausse dans s
$p \in \mathcal{L}(s) \wedge \overline{p} \notin \mathcal{L}(s)$	p est vraie dans s
$p \in \mathcal{L}(s) \wedge \overline{p} \in \mathcal{L}(s)$	p est inconsistante dans s

TAB. 6.1 – Information sur une proposition atomique p dans un état s

Définition 6.3 *Une* structure de Kripke abstraite (AKS) *est un 6-uplet*
$A = \langle AP, S, S_0, \mathcal{L}, R, F \rangle$. *$S, S_0, R, F$ sont défini comme dans la définition 1.4 précédente et $\mathcal{L} : S \rightarrow 2^{Lit}$.*

Nous définissons les états inconsistants comme des états où au moins une des propositions atomiques p est vraie et sa négation $\overline{p}$ aussi.

Définition 6.4 *Un* état inconsistant *s est un état où il existe $p \in AP$ tel que $p \in \mathcal{L}(s)$ and $\overline{p} \in \mathcal{L}(s)$.*

Un chemin inconsistant *$\overline{\pi}$ contient au moins un état inconsistant.*

L'abstraction que nous allons construire dans le chapitre suivant est représentée par une structure de Kripke abstraite. La représentation de l'information dans un état, que nous avons choisie, permet de faciliter l'écriture de notre algorithme d'abstraction. De plus, l'avantage de rester sur une logique à deux valeurs prend son sens pour l'intégration de nos modèles dans un model checker classique.

Chapitre 7

Spécification comme abstraction

7.1 Construction d'une AKS à partir de formules CTL

Nous allons présenter la construction d'une AKS K_φ, à partir d'une formule CTL φ. Les formules CTL considérées sont restreintes à la forme normale positive de CTL$\backslash X$ dans laquelle les négations ne s'appliquent qu'aux propositions atomiques.

7.1.1 Définitions Préliminaires

Pour pouvoir construire nos AKS, nous avons besoin d'une fonction qui puisse étendre la fonction d'étiquetage pour une structure. Cela peut arriver lorsque l'on ajoute des nouvelles propositions atomiques à considérer ou lorsque l'on étend l'ensemble des états de la structure de Kripke abstraite. Les deux définitions suivantes décrivent successivement, l'extension de $\mathcal{L}$ lorsque l'ensemble des propositions atomiques augmente et l'ensemble des états reste inchangé, puis lorsque l'ensemble des propositions atomiques est constant et l'ensemble des états augmente.

Définition 7.1 Extension d'états
Soient $\mathcal{L} : S \rightarrow 2^{Lit}$ *et* $\mathcal{L}' : S' \rightarrow 2^{Lit}$ *deux fonctions d'étiquetages avec le même ensemble de propositions atomiques.* $\mathcal{L}'' = \mathcal{L} \cdot \mathcal{L}'$ *est une fonction d'étiquetage liée à Lit telle que pour chaque état* $s \in S \cup S'$ *:*

$\mathcal{L}''(s) = \mathcal{L}(s)$ *ssi* $s \in S$ *et* $s \notin S'$ *;*
$\mathcal{L}''(s) = \mathcal{L}'(s)$ *ssi* $s \in S'$ *et* $s \notin S$ *;*
$\mathcal{L}''(s) = \mathcal{L}(s) \cup \mathcal{L}'(s)$ *ssi* $s \in S \cap S'$*.*

Définition 7.2 Extension des propositions atomiques
Soit $\mathcal{L}$ *une fonction d'étiquetage* $\mathcal{L} : S \rightarrow 2^{Lit}$*, et soit* Lit' *un ensemble de propositions atomiques tel que* $Lit \cap Lit' = \emptyset$*. La fonction d'étiquetage* $\mathcal{L}^{+Lit'} : S \rightarrow Lit' \cup Lit$ *est définie tel que :*
$\forall p \in Lit, \forall s \in S$*,* $\mathcal{L}^{+Lit'}(s) = \mathcal{L}(s)$ *et l'ensemble des nouvelles propositions atomiques ont une valeur indéterminée.*

Pour la construction des ensembles de contraintes d'équités nous ajoutons deux opérateurs : $F \oplus S$ qui fait l'union de chaque ensemble de F avec S ; $F \ominus S$ est l'opération duale à la précédente.

Définition 7.3 *Soit* $F = \{F_0, \dots F_n\}$ *un ensemble de contraintes d'équités et soit* S *un ensemble d'état :*

- $F \oplus S = \{F_i \cup S \mid F_i \in F\}$;
- $F \ominus S = \{F_i \setminus S \mid F_i \in F\}$;

7.1.2 Composition de structures

Dans notre démarche, la composition intervient dans deux cas : lors de la création d'une abstraction pour un seul composant par conjonction de formules CTL ; lors de la composition des abstractions des composants du système. Dans les deux cas, l'abstraction doit garantir l'absence de chemin inconsistant. Dans le premier cas, les chemins inconsistants ne correspondent à aucune exécution possible du composant. En effet, un composant ne peut pas positionner un signal à deux valeurs en même temps. Dans le second cas, un état inconsistant peut seulement provenir de composants différents qui contrôlent le même signal par l'intermédiaire de *tri-state*. En général des politiques d'écriture sont mises en place afin de garantir l'exclusivité d'écriture. De plus, l'emploi de ce type d'élément est de plus en plus évité. Par conséquent, nous supposons là aussi que les chemins inconsistants ne correspondent à aucune exécution possible du système.

Nous reprenons la définition de la composition parallèle de [GL91] pour prendre en compte les contraintes d'équités. Les contraintes d'équités assurent qu'un chemin de $K_1 \parallel K_2$ est équitable si et seulement si sa projection sur chacun des composants correspond à un chemin équitable dans K_1 et dans K_2.

Définition 7.4 *Soient* $K_1 = \langle AP_1, S_1, S_{0_1}, L_1, R_1, F_1 \rangle$ *et* $K_2 = \langle AP_2, S_2, S_{0_2}, L_2, R_2, F_2 \rangle$ *deux AKS. La* composition synchrone *de* K_1 *et* K_2 *notée* $K_1 \parallel K_2$ *est la structure* K *définie de la façon suivante :*

$AP = AP_1 \cup AP_2$;

$S \subseteq (S_1 \times S_2) \setminus \{(s_1, s_2) \mid p \in \mathcal{L}((s_1, s_2)) \text{ et } \overline{p} \in \mathcal{L}((s_1, s_2))\}$;

$S_0 = (S_{0_1} \times S_{0_2})$;

$\mathcal{L}((s_1, s_2)) = \mathcal{L}_1(s_1) \cup \mathcal{L}_2(s_2)$;

$R \subseteq S_1 \times S_2 \times S_1 \times S_2$, *avec* $R((s_1, s_2), (t_1, t_2))$ *ssi* $R_1(s_1, t_1)$ *et* $R_2(s_2, t_2)$;

$F \subseteq \{(P_1 \times S_2) \cap S \mid P_1 \in F_1\} \cup \{(S_1 \times P_2) \cap S \mid P_2 \in F_2\}$.

7.2 Algorithme de construction

L'algorithme construit une structure de Kripke abstraite directement à partir d'une formule CTL\X φ qui est la structure la moins *contrainte* qui vérifie φ (et ne vérifie pas $\overline{\varphi}$). Nous avons décomposé l'algorithme en 3 parties :

1. Traductions des formules ou des sous-formules avec comme premier quantificateur **A**.
2. Traductions des formules ou des sous-formules avec comme premier quantificateur **E**.
3. Règles pour la combinaison des deux traductions précédentes.

7.2.1 Quantificateur universel A

Les opérateurs ACTL sont réécrits de la manière suivante :
- $\mathbf{AG}(f \wedge g)$ en $\mathbf{AG}(f) \wedge \mathbf{AG}(g)$;
- $\mathbf{A}[(f \wedge g)\mathbf{U}h]$ en $\mathbf{A}[f\mathbf{U}h] \wedge \mathbf{A}[g\mathbf{U}h]$;

La première étape de notre algorithme construit la structure de Kripke correspondante à la formule **true**. Cette structure contient un état unique dans lequel toutes les formules peuvent être vraies. Cela correspond au fait qu'aucune proposition atomique n'appartient à la fonction d'étiquetage, ainsi toute proposition atomique p est indéterminée (elle peut être vraie comme elle peut être fausse). Nous définissons un état spécial absorbant s_T avec une unique transition qui boucle sur lui-même. L'arbre d'exécution obtenu par le dépliage des transitons de cet état est une abstraction de n'importe quel arbre d'exécution.

Définition 7.5 *Un état s est* absorbant *si les deux conditions suivantes sont vraies :*

1. *$\mathcal{L}(s) = \emptyset$ et*
2. *il existe une unique transition sortante de s vers lui-même.*

Un état composé (s, s') appartient est absorbant si et seulement si s et s' sont eux-aussi absorbants.

Propriété 7.1 *Une structure de Kripke abstraite réduit à un état $s_T \in S_T$ simule les comportements infinis de toutes structures de Kripke (non vide).*

Preuve Soit K une structure de Kripke. On définit la relation $H = S_K \times \{s_T\}$. Evidemment, H est une relation de simulation :
- $\mathcal{L}_K(s_K) \cap Lit' = \mathcal{L}'_T(s_T)$ $(Lit' = \emptyset)$ et
- Pour toutes paires s_1, s'_1 tel que $H(s_1, s'_1)$ et pour tout état $s_2 \in S_K$ tel que $s_1 \rightarrow s_2$, il existe $s'_2 \in \{s_T\}$ tel que $s'_1 \rightarrow s'_2$, $s'_2 = s_T$ et $H(s_2, s'_2)$.

■

Dans une structure abstraite obtenue à l'aide de notre algorithme, un état s_T est accessible à partir d'un état s qui valide la formule. C'est à dire que la formule est vraie à partir de l'état initial et tous les chemins sortants de s ne modifient plus la validité de la formule. Nous noterons S_T l'ensemble des états absorbants qui ne sont accessibles qu'à partir d'un chemin qui valide la formule.

Définition 7.6 *Soit S_T l'ensemble des états absorbants accessibles à partir d'un chemin qui valide la formule, S_{predT} est* l'ensemble des états prédécesseurs *d'un état appartenant à S_T : $\forall s \in S_{predT}$, $\exists s_T \in S_T$, tel que $s \rightarrow s_T$.*

La figure 7.1 présente l'exemple de la structure de Kripke abstraite obtenue à partir de la formule $\mathbf{AF}p$. La fonction d'étiquetage est : $\mathcal{L}(s_0) = \emptyset$; $\mathcal{L}(s_1) = \{p\}$. La contrainte d'équité est $\{s_T\}$. Dans l'état s_1 la proposition atomique p est vraie donc $\mathbf{AF}p$ est vraie, la suite de l'arbre d'exécution n'est pas utile pour l'évaluation de la formule. Ceci est représenté par la transition de s_1 vers s_T. Dans le cas où la formule contient une promesse, nous devons ajouter une contrainte d'équité afin que cette promesse soit accomplie. Dans

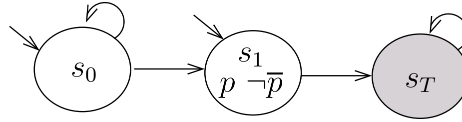


Fig. 7.1 – AKS de la formule $\mathbf{AF}p$

l'exemple, la contrainte d'équité impose d'accéder à l'état s_T et donc de passer par l'état s_1 qui vérifie p. A chaque niveau de profondeur de l'arbre d'exécution obtenu à partir de s_0, l'état s_0 peut prendre une transition vers s_1 et par conséquent s_0 vérifie $\mathbf{AF}p$.

L'algorithme suivant construit une AKS à partir de la formule φ. Les cas de bases sont représentés sur les figures 7.2. Pour un état s, toutes les propositions atomiques p qui n'apparaissent pas dans son étiquette sont telles que $p \notin \mathcal{L}(s)$.

Définition 7.7 *Soit $AKS(\varphi)$ la fonction qui associe une formule φ à une structure de Kripke abstraite, $AKS(\varphi)$ est définie par induction de la manière suivante. AP est un ensemble de propositions atomiques initial, p et q sont des propositions atomiques, φ_1, φ_2 sont des formules CTL et $K_{\varphi_i} = \langle AP_i, S_i, S_{0_i}, \mathcal{L}_i, R_i, F_i\rangle$ est une structure de Kripke abstraite liée à φ_i pour $i = 1, 2$.*

- $\varphi = \mathbf{true}$,

  $K_\varphi = \langle \emptyset, \{s_T\}, \{s_T\}, \emptyset, \{s_T, s_T\},\ \{s_T\}\rangle$

- $\varphi = \mathbf{p}$,

  – $K_T = AKS(\mathbf{true}) = \langle \emptyset, S_T, s_T, \emptyset, R_T, F_T\rangle$
  – Soit s un nouvel état tel que $p \in \mathcal{L}(s)$

  $K_\varphi = \langle AP \cup \{p\}, \{s\} \cup S_T, \{s\}, \mathcal{L}, \{(s, s') \mid s' \in S_{0_T}\} \cup R_T, F_T\rangle$

- $\varphi = \varphi_\mathbf{1} \vee \varphi_\mathbf{2}$,

  – $K_{\varphi_1} = AKS(\varphi_1)\,;\ K_{\varphi_2} = AKS(\varphi_2)$

  $K_\varphi = \langle AP_{\varphi_1} \cup AP_{\varphi_2}, S_{\varphi_1} \cup S_{\varphi_2}, S_{0_{\varphi_1}} \cup S_{0_{\varphi_2}}, L_{\varphi_1}^{+Lit_{\varphi_2}} \cdot L_{\varphi_2}^{+Lit_{\varphi_1}}, R_{\varphi_1} \cup R_{\varphi_2}, F_{\varphi_1} \cup F_{\varphi_2}\rangle$.

- $\varphi = \varphi_\mathbf{1} \wedge \varphi_\mathbf{2}$,

  – $K_{\varphi_1} = AKS(\varphi_1)\,;\ K_{\varphi_2} = AKS(\varphi_2)$

  $K_\varphi = K_{\varphi_1} \parallel K_{\varphi_2}$

- $\varphi = \mathbf{AF}\varphi_\mathbf{1}$,

  – $K_{\varphi_1} = AKS(\varphi_1)$
  – Soit s un état tel que $\forall p \in AP_{\varphi_1}$, p et $\overline{p} \notin \mathcal{L}(s)$.

  $K_\varphi = \langle AP_{\varphi_1}, \{s\} \cup S_{\varphi_1}, \{s\} \cup S_{0_{\varphi_1}}, \mathcal{L} \cdot \mathcal{L}_{\varphi_1}, R_{\varphi_1} \cup \{(s, s)\} \cup \{(s, s_i) \mid \forall s_i \in S_{0_{\varphi_1}}\}, F_{\varphi_1}\rangle$

- $\varphi = \mathbf{AG\varphi_1}$,
  - $K_{\varphi_1} = AKS(\varphi_1)$
  - $R = [S_{\varphi_1} \cap S_{predT}] \times S_{0_{\varphi_1}}$: pour tous les états $s \in S_{\varphi_1} \cap S_{predT}$ et pour tout $s_0 \in S_{0_{\varphi_1}}$, $(s, s_0) \in R$.

  $K_\varphi = \langle AP_{\varphi_1}, S_{\varphi_1} \setminus S_T, S_{0_{\varphi_1}}, L_{\varphi_1}, R \cup (R_{\varphi_1} \setminus \{(s, s_T) \mid s \in S_{predT} \wedge s_T \in S_T\}), (F_{\varphi_1} \ominus S_T) \cup S_{predT_{\varphi_1}} \rangle$

- $\varphi = \mathbf{A}[\mathbf{\varphi_1 U \varphi_2}]$,
  - $K_{\varphi_1} = AKS(\varphi_1)\,;\, K_{\varphi_2} = AKS(\varphi_2)\,;\, K_{\varphi_1 \wedge \varphi_2} = AKS(\varphi_1 \wedge \varphi_2)$
  - $R = [S_{\varphi_1} \cap S_{predT}] \times [S_{0_{\varphi_1}} \cup S_{0_{\varphi_2}}]$
  - Soit R' une relation de transition définie telle que pour tous les états $s \in S_{0_{\varphi_1}} \setminus S_{predT}$ et pour tout $(s_{0_1}, s_{0_2}) \in S_{0_{\varphi_1 \wedge \varphi_2}}$, $(s, s_0) \in R'$ ssi $(s, s_{0_1}) \in R_{0_{\varphi_1}}$.

  $K_\varphi = \langle AP_{\varphi_1} \cup AP_{\varphi_2}, (S_{\varphi_1} \setminus S_T) \cup S_{\varphi_2} \cup S_{\varphi_1 \wedge \varphi_2}, S_{0_{\varphi_1}} \cup S_{0_{\varphi_2}} \cup S_{0_{\varphi_1 \wedge \varphi_2}}, L_{\varphi_1}^{+AP_2} \cdot L_{\varphi_2}^{+AP_1}, R \cup R' \cup (R_{\varphi_1} \setminus \{(s, s_T) \mid s \in S_{predT} \wedge s_T \in S_T\}) \cup R_{\varphi_2} \cup R_{\varphi_1 \wedge \varphi_2}, F_{\varphi_2} \cup F_{\varphi_1 \wedge \varphi_2} \rangle$

- $\varphi = \mathbf{A}[\mathbf{\varphi_1 W \varphi_2}]$ procède de la même façon que $A[\varphi_1 U \varphi_2]$ sauf que la contrainte d'équité est $F_\varphi = \{F_{\varphi_1} \cup F_{\varphi_2} \cup F_{\varphi_1 \wedge \varphi_2}\}$.

7.2.2 Quantificateur existentiel E

Nous considérons les formules de la forme EGf, $E[fUg]$ où f et g *ne sont pas des conjonctions* de sous-formules.

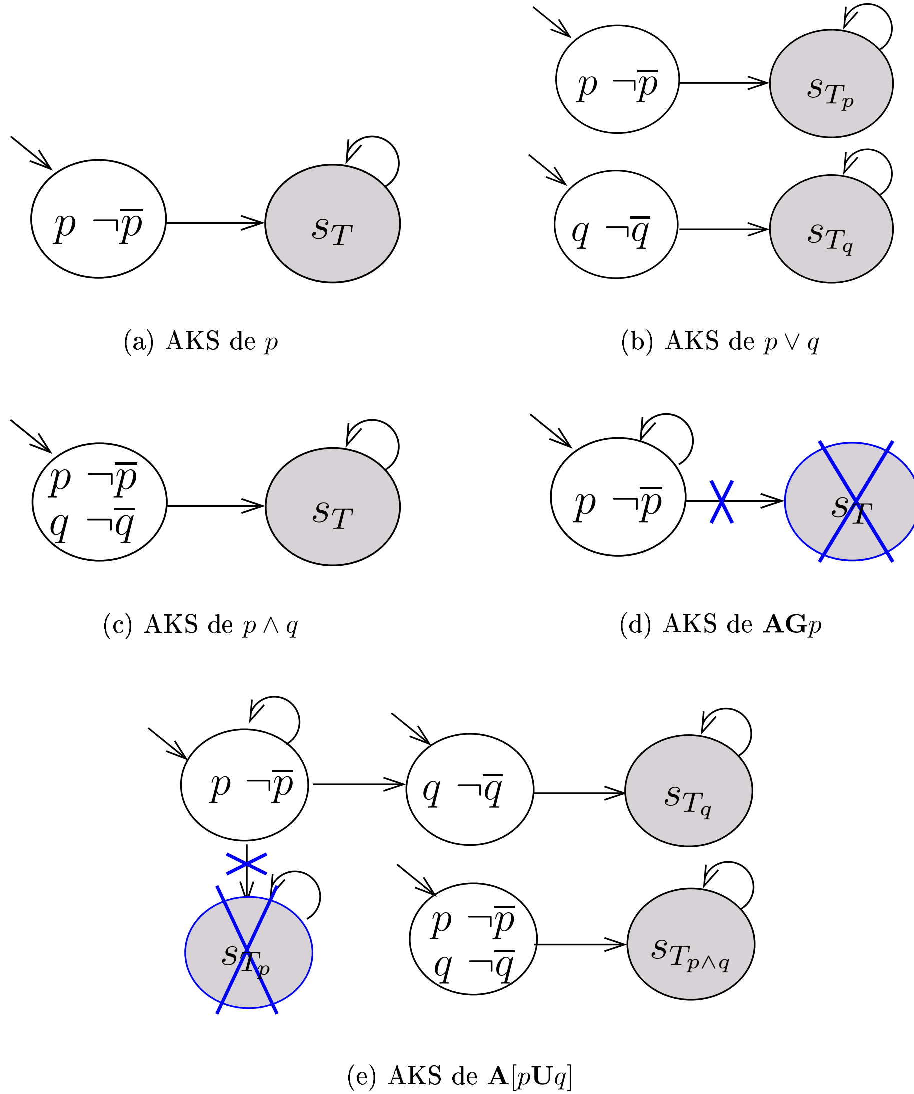
L'algorithme de construction des AKS, pour des formules CTL ayant comme premier quantificateur **E**, est similaire à l'algorithme précédent. La principale différence se trouve dans la représentation des chemins qui ne vérifient pas la formule considérée. En effet, dans la structure de Kripke abstraite, nous avons besoin de représenter les chemins qui à partir de l'état initial vérifient la formule mais aussi les chemins qui divergent. Nous avons décidé de représenter cette partie du dépliage de l'arbre d'exécution par un état absorbant s_D où toutes les propositions atomiques ont une valeur indéterminée. Cette état est différent de l'état s_T que nous avons défini précédemment. L'état s_D est atteind par des chemins qui ne valident par la formule.

Définition 7.8 *Soit S_D l'ensemble des états divergents s_D, S_{predD} est* l'ensemble des états prédécesseurs *d'un état de S_D : $\forall s \in S_{predD}$, $\exists s_D \in S_D$, tel que $s \rightarrow s_D$.*

Propriété 7.2 *Une structure de Kripke abstraite réduite à un état $s_D \in S_D$ simule toutes les structures de Kripke non vide (avec des comportements infinis)*

Un état composé (s,s') appartient à S_D si et seulement si $s \in S_D$ et $s' \in S_D$ ou $s \in S_T$ et $s' \in S_D$ ou $s \in S_D$ et $s' \in S_T$.

La définition suivante doit être ajoutée à la définition 7.7 pour les formules ayant pour pour premier quantificateur **E**.



(a) AKS de p (b) AKS de $p \vee q$

(c) AKS de $p \wedge q$ (d) AKS de $\mathbf{AG}p$

(e) AKS de $\mathbf{A}[p\mathbf{U}q]$

Fig. 7.2 – AKS : cas de base pour les formules avec un **A** comme premier quantificateur

Définition 7.9 *Soit $AKS(\varphi)$ une fonction qui associe une formule φ à une structure de Kripke abstraite, $AKS(\varphi)$ est définit par induction de la façon suivante. AP est un ensemble de propositions atomiques initiale, p et q sont des propositions atomiques, φ_1, φ_2 sont des formules CTL et $K_{\varphi_i} = \langle AP_i, S_i, S_{0_i}, \mathcal{L}_i, R_i, F_i \rangle$ sont des structures de Kripke abstraites liées à φ_i pour $i = 1, 2$.*

- $\varphi = \mathbf{EF}\boldsymbol{\varphi_1}$,
  - $K_{\varphi_1} = AKS(\varphi_1)$
  - Soit s un état tel que $\mathcal{L}(s) = \emptyset$.

  $K_\varphi = \langle AP_{\varphi_1}, \{s\} \cup S_{\varphi_1} \cup \{s_D\}, \{s\} \cup S_{0_{\varphi_1}}, \mathcal{L} \cdot \mathcal{L}_{\varphi_1}, R_{\varphi_1} \cup \{(s,s)\} \cup \{(s, s_i) \mid \forall s_i \in S_{0_{\varphi_1}}\} \cup \{(s, s_D)\}, F_{\varphi_1} \oplus \{s_D\} \rangle$

- $\varphi = \mathbf{EG}\varphi_\mathbf{1}$,
  - $K_{\varphi_1} = AKS(\varphi_1)$
  - $R = [S_{\varphi_1} \cap S_{predT}] \times S_{0_{\varphi_1}}$
  - $R' = [S_{\varphi_1} \cap S_{predT}] \times \{s_D \mid s_D \in S_D\}$

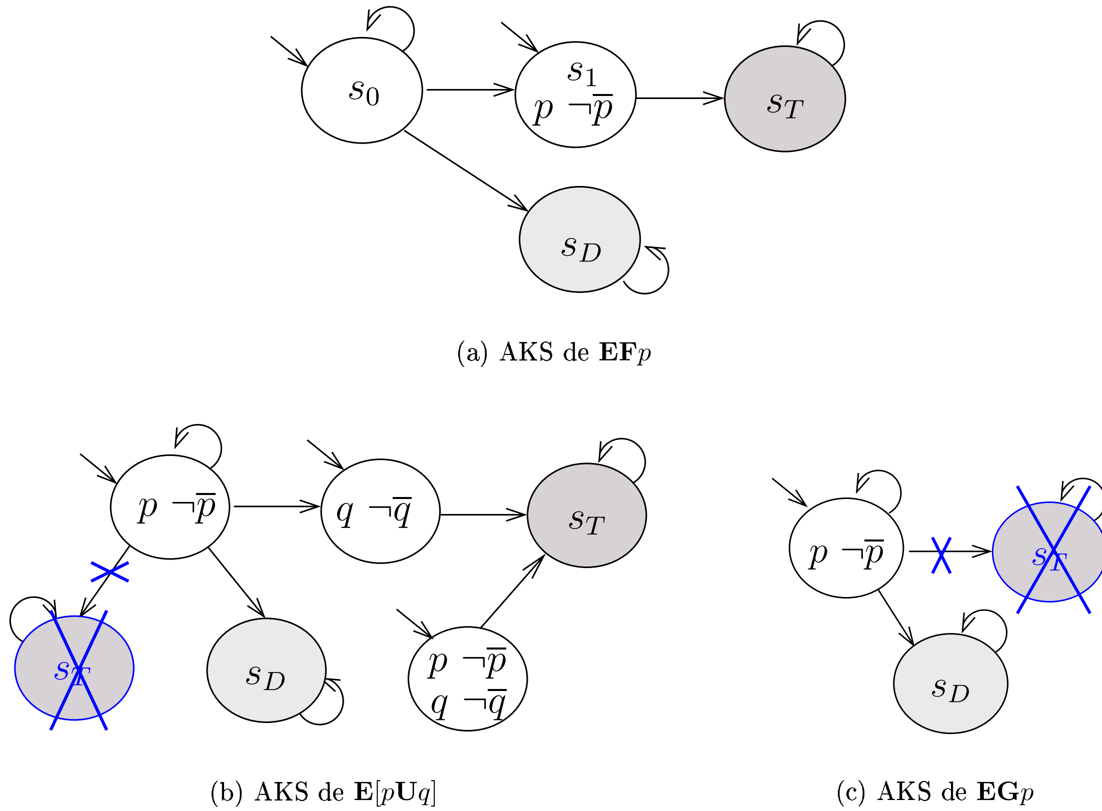
  $K_\varphi = \langle AP_{\varphi_1}, (S_{\varphi_1} \setminus S_T) \cup S_D, S_{0_{\varphi_1}}, L_{\varphi_1}, R \cup R' \cup (R_{\varphi_1} \setminus \{(s, s_T) \mid s \in S_{predT} \wedge s_T \in S_T\}, (F_{\varphi_1} \ominus S_T) \oplus \{s_D\}\rangle$
- $\varphi = \mathbf{E}[\varphi_\mathbf{1}\mathbf{U}\varphi_\mathbf{2}]$,
  - $K_{\varphi_1} = AKS(\varphi_1)$; $K_{\varphi_2} = AKS(\varphi_2)$; $K_{\varphi_1 \wedge \varphi_2} = AKS(\varphi_1 \wedge \varphi_2)$
  - $R = [S_{\varphi_1} \cap S_{predT}] \times [S_{0_{\varphi_1}} \cup S_D]$ $(s, s_D) \in R$.
  - Sot R' la relation de transition définie telle que, pour tout état $s \in S_{0_{\varphi_1}} \setminus S_{predT}$ et pour tout $(s_{0_1}, s_{0_2}) \in S_{0_{\varphi_1 \wedge \varphi_2}}$, $(s, s_0) \in R'$ ssi $(s, s_{0_1}) \in R_{0_{\varphi_1}}$.

  $K_\varphi = \langle AP_{\varphi_1} \cup AP_{\varphi_2}, (S_{\varphi_1} \setminus S_T) \cup S_{\varphi_2} \cup S_{\varphi_1 \wedge \varphi_2} \cup S_D, S_{0_{\varphi_1}} \cup S_{0_{\varphi_2}} \cup S_{0_{\varphi_1 \wedge \varphi_2}}, L_{\varphi_1}^{+AP_2} \cdot L_{\varphi_2}^{+AP_1}, R \cup R' \cup (R_{\varphi_1} \setminus \{(s, s_T) \mid s \in S_{predT} \wedge s_T \in S_T\}) \cup R_{\varphi_2} \cup R_{\varphi_1 \wedge \varphi_2}, (F_{\varphi_2} \cup F_{\varphi_1 \wedge \varphi_2}) \oplus \{s_D\}\rangle$
- $\varphi = \mathbf{E}[\varphi_\mathbf{1}\mathbf{W}\varphi_\mathbf{2}]$ procède comme $E[\varphi_1 U \varphi_2]$ sauf que l'ensemble de contraintes d'équité est $F_\varphi = \{F_{\varphi_1} \cup F_{\varphi_2} \cup F_{\varphi_1 \wedge \varphi_2}\} \oplus \{s_D\}$.

Pour que notre algorithme soit applicable à l'ensemble des formules CTL tel que nous l'avons définit dans la sous-section 7.2.1, nous devons ajouter des règles à l'algorithme qui traite le cas des formules avec un **A** comme premier quantificateur définition 7.7. Ces règles servent à traiter les états de divergents.

Définition 7.10 *Règles additionnelles à appliquer à l'algorithme traitant les formules CTL avec* **A** *comme premier quantificateur.*

- $\varphi = \mathbf{AG}\varphi_\mathbf{1}$,
  - $S_\varphi = S_{\varphi_1} \setminus (S_T \cup S_D)$
  - *Soit R la relation de transition obtenu par la définition 7.7 (item* **AG***),* $R_\varphi = (R \setminus \{(s, s_D) \mid s \in S_{predD} \wedge s_D \in S_D)\} \cup \{(s, s_0) \mid s \in S_{predD} \wedge s_0 \in S_{0_{\varphi_1}}\}$
  - $F_\varphi = (F_{\varphi_1} \ominus (S_T \cup S_D)) \cup S_{predT} \cup S_{predD}$
- $\varphi = \mathbf{A}[\varphi_\mathbf{1}\mathbf{U}\varphi_\mathbf{2}]$,
  - $S_\varphi = S_{\varphi_1} \setminus (S_T \cup S_D)$
  - *Soit R la relation de transition obtenue par la définition 7.7 (item* **AU***),* $R_\varphi = (R \setminus \{(s, s_D) \mid s \in S_{predD} \wedge s_D \in S_D) \cup \{(s, s_0) \mid s \in S_{predD} \wedge s_0 \in S_{0_{\varphi_1}}\}$



(a) AKS de $\mathbf{EF}p$

(b) AKS de $\mathbf{E}[p\mathbf{U}q]$

(c) AKS de $\mathbf{EG}p$

Fig. 7.3 – AKS : cas de base pour les formules avec un **E** comme premier quantificateur

Exemple La figure 7.4 illustre l'AKS construite à partir de la formule $\mathbf{EG}(\mathbf{AF}(p \vee q))$. L'algorithme procède de la façon suivante :

1. Construction de $AKS(s_{T_p})$: création d'un état s_{T_p}
2. Construction de $AKS(p)$: création d'un état s_p tel que $p \in \mathcal{L}(s)$ et $\overline{p} \notin \mathcal{L}(s_p)$, ajout d'une transition de s_p à s_{T_p} ;
3. Construction de $AKS(s_{T_q})$: création d'un état s_{T_q}
4. Construction de $AKS(q)$: création d'un état s_q tel que $q \in \mathcal{L}(s)$ et $\overline{q} \notin \mathcal{L}(s_q)$, ajout d'une transition de s_q à s_{T_q} ;

2,4. Construction de $AKS(p \vee q)$: union de $AKS(p)$ et $AKS(q)$;

5. Construction de $AKS(\mathbf{AF}(p \vee q))$: ajout d'un état s tel que $\mathcal{L}(s) = \emptyset$, une transition vers s_p et une transition vers s_q ;
6. Construction de $AKS(\mathbf{EG}(\mathbf{AF}(p \vee q)))$: suppression des états s_T et des transitions menant à ces états, création d'un état s_D, ajout de transitions de s_p et s_q vers l'ensemble des états initiaux et vers s_D.

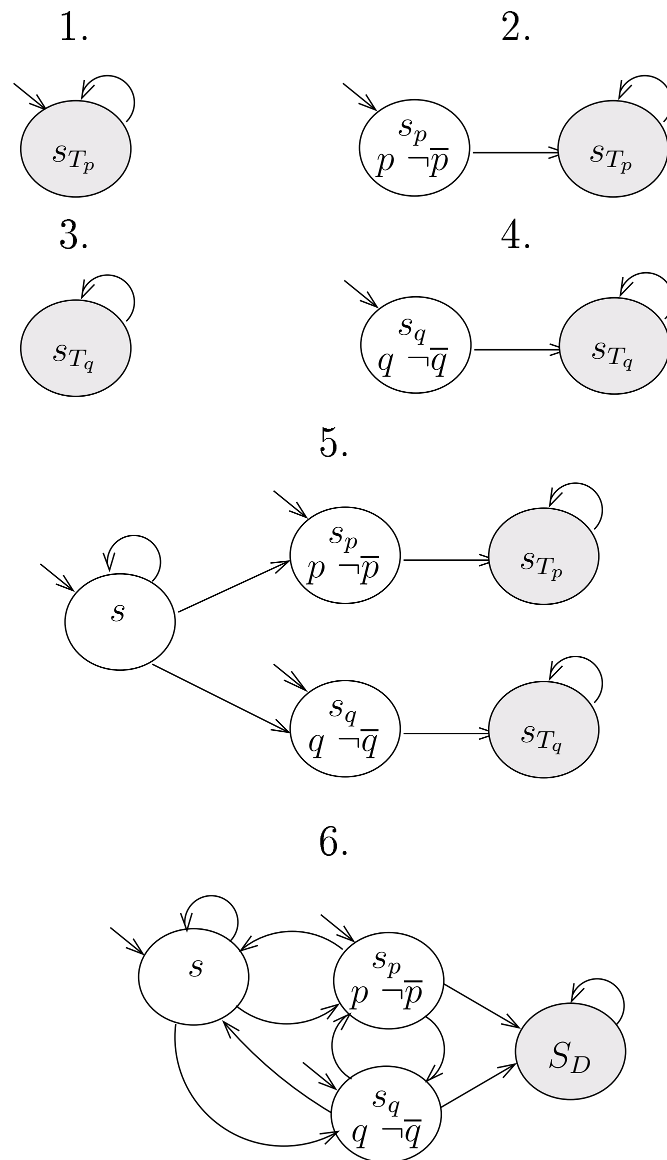


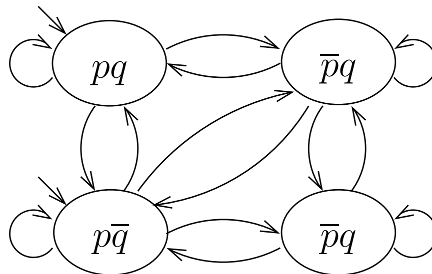
Fig. 7.4 – Construction de l'AKS de la formule $\mathbf{EG}(\mathbf{AF}(p \vee q))$

7.3 Propriétés d'un AKS construit à partir d'une formule

7.3.1 Propriétés de K_φ

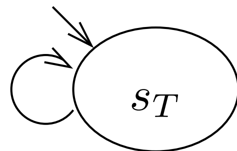
Une structure de Kripke concrète ne peut pas vérifier φ et $\overline{\varphi}$. En revanche, une structure abstraite peut vérifier à la fois φ et $\overline{\varphi}$. Nous allons montrer que notre structure abstraite construite à partir d'une formule φ est la structure la plus abstraite vérifiant φ et ne vérifiant pas $\overline{\varphi}$. Dans l'exemple figure 7.5, l'ensemble des structures ont le même ensemble de propositions atomique $AP = p, q$. Notre structure est plus précise et plus grande qu'une abstraction très "grossière". L'avantage de notre structure est que pour une propriété donnée, notre structure vérifie cette propriété et ne vérifie pas sa négation.

K_c



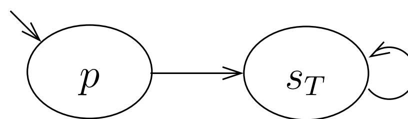
(a) Structure de Kripke concrète (2 valeurs) : $K_c \models p$

K_a



(b) Structure de Kripke abstraite sur (3 valeurs) : $K_a \models p$ et $K_a \models \overline{p}$

K_p



(c) $K_p = AKS(p)$; $K_p \models p$ et $K_p \not\models p$

FIG. 7.5 – Comparaison entre une structure concrète et des structures abstraites

Propriété 7.3 *L'AKS K_φ obtenue par les définitions 7.7,7.9 et 7.10 est telle que $\forall s_0 \in S_{0_\varphi}$, $K_\varphi, s_0 \models \varphi$.*

Preuve (Sketch)
Les preuves détaillées sont en annexe. Les preuves procèdent par induction sur la taille de la formule. ■

Comme dans les travaux de Bruns, Godefroid [BG99] et Shoham, Grumberg [SG04], notre fonction d'étiquetage $\mathcal{L}$ implique un ordre partiel ($\sqsubseteq$) entre états selon le niveau d'information connu pour chaque proposition atomique dans chacun des états. On a $s \sqsubseteq s'$ si il existe $p \in AP$ tel que p est *moins contraint* dans s' que dans s ($p \in \mathcal{L}(s)$ et $p \notin \mathcal{L}(s') \wedge p \notin \mathcal{L}(s')$) et pour tout $q \in AP$ et $q \neq p$, q as la même valeur de vérités dans s et dans s'. Nous pouvons en déduire, comme dans les travaux cités précédemment qu'il existe une relation de simulation entre $K \models \varphi$ et K_φ, notée $K \preceq K_\varphi$, avec $\preceq$ est une relation de préordre.

Propriété 7.4 *Pour toutes structures de Kripke K tel que $K \models \varphi$ et $K \not\models \overline{\varphi}$ il existe $K_\varphi = AKS(\varphi)$ et il existe une relation de simulation $\preceq$ tel que $K \preceq K_\varphi$.*

Le preuve est donnée en annexe. Elle procède par induction sur la taille de la formule, les cas de base sont les formules sans opérateur CTL imbriqué.

7.3.2 Composition des abstractions

Chaque composant abstrait aura une structure de Kripke abstraite K_φ obtenue par notre algorithme, où $\varphi \in P$. De plus, les hypothèses sur le comportement de l'environnement serons elle aussi traduites en structure de Kripke.

Définition 7.11 *Soit $C = \langle K, P, A\rangle$ un composant, le* composant abstrait *de C construit à partie de $\varphi \subseteq P$ est un 3-uplets $C_\varphi = \langle K_\varphi, \varphi, K_A\rangle$ où K_φ et K_A sont construits selon les algorithmes de la définition 7.7,7.9 et 7.10.*

Après avoir abstrait tous les composants à l'aide d'un sous-ensemble de leur spécification, nous voulons les composer afin d'obtenir une abstraction du système complet. Pour cela, il faut d'abord être sûr qu'une telle composition est possible, c'est à dire que tous les composants à combiner sont *compatibles* entre eux. Ils sont compatibles si leurs signaux de sorties sont des ensembles disjoints . Les systèmes que nous considérons sont des composant matériels, deux composants ne peuvent pas écrire simultanément sur le même signal de sortie. Une politique d'arbitrage garantit l'exclusivité en écriture sur les signaux de sorties partagés (comme les bus).

Le système abstrait complet est alors obtenu pas la composition de tous les composants abstraits par composition synchrone (Def. 7.4). Une conséquence de l'utilisation de composants compatibles est que l'introduction de nouveaux états inconsistants est impossible par la composition.

De plus, d'après les travaux de McMillan [McM00] appliqués à notre cadre de conception, nous pouvons composer un composant C_i avec d'autres composants $C_k \ldots C_m$ si et seulement si l'ensemble des spécifications des composants $C_k \ldots C_m$ implique les hypothèses sur les comportements de l'environnement de C_i : $\bigcup_{j=k \ldots m} P_j \implies A_i$.

Nous allons maintenant prouver que la compositions des abstractions des composants est bien une abstraction d'une système complet. Chaque composant est abstrait par des formules CTL, la preuve applique un raisonnement *assume-guarantee* [HQRT02].

Propriété 7.5 *Soient C_1, ..., C_n des composants concrets $\langle K_i, P_i, A_i\rangle$, et C_{φ_1}, ..., C_{φ_n} des composants abstraits $\langle K_{\varphi_i}, \varphi_i, K_{A_i}\rangle$ avec $\forall i,\ \varphi_i \subseteq P_i$. Soit $\mathcal{S} = C_1 \parallel \ldots \parallel C_n$ un système concret et $\mathcal{S}_A = C_{\varphi_1} || \ldots C_{\varphi_n}$ un modèle abstrait, alors $\mathcal{S}_A$ simule $\mathcal{S}$.*

Preuve Le résultat est l'application directe du raisonnement assume-guarantee. Exemple pour deux composantsC_1 et C_2 :

$$
\begin{array}{rcll}
K_1 \parallel K_{A_1} & \preceq & K_1 & \text{(propriété de la composition synchrone)} \\
K_1 & \preceq & K_{\varphi_1} & \text{(propriété 7.4)} \\
\hline
K_1 \parallel K_{A_1} & \preceq & K_{\varphi_1} & \text{(transitivité)} \\
K_2 \parallel K_{A_2} & \preceq & K_2 & \text{(propriété de la composition synchrone)} \\
K_2 & \preceq & K_{\varphi_2} & \text{(propriété 7.4)} \\
\hline
K_2 \parallel K_{A_2} & \preceq & K_{\varphi_2} & \text{(transitivité)} \\
K_{A_1} & \preceq & K_{\varphi_2} & \text{(hypothèse de la composition)} \\
K_{A_2} & \preceq & K_{\varphi_1} & \text{(hypothèse de la composition)} \\
\hline
K_1 \parallel K_2 & \preceq & K_{\varphi_1} \parallel K_{\varphi_2} & \text{(assume-guarantee)}
\end{array}
$$

■

Nous pouvons alors conclure que l'ensemble des formules ACTL est préservé de du système abstrait au système concrets.

7.3.3 Vérification du système abstrait par model checking

Jusqu'à maintenant, pendant l'abstraction d'un seul composant par rapport à une propriété *locale* φ, les signaux qui n'entrent pas en jeu pour la vérification de la formule ont été abstrait avec une valeur indéterminée. Cette valeur est considérée comme "sans intérêt". Pendant la vérification d'une propriété *globale* ψ (mettant en oeuvre plusieurs composants), la valeur de tels signaux est maintenant interprétée comme *inconnue*. Nous ne savons pas si la valeur du signal est vraie ou faux.

Nous utilisons un model checker classique sur 2 valeurs de vérité, où tous les signaux avec une valeur *indéterminée* sont mis à une valeur non déterministe. Si le model checker détermine que la propriété globale n'est pas vérifiée sur notre système abstrait, nous devons déterminer si le contre-exemple fourni est vrai (simulable sur le système concret) ou faux (il n'existe pas d'exécution correspondante possible du système concret) ([CGJ$^+$03]).

Propriété 7.6 Model checking de formules ACTL sur un système abstrait est conservative
Soient $\mathcal{S}$ un système concret et $\mathcal{S}_A$ un système abstrait obtenu par abstraction de certains (ou tous) composants de $\mathcal{S}$. Pour toutes formules ψ dans ACTL $[\mathcal{S}_A \models \psi] = \mathbf{true} \Rightarrow [\mathcal{S} \models \psi] = \mathbf{true}$

Preuve Directement à partir de l'existence d'une relation de simulation entre le système abstrait et le système concret propriété 7.5.

■

L'algorithme que nous avons défini permet la construction automatique d'une structure de Kripke abstraite. Notre abstraction tient compte de l'ensemble des spécifications déjà vérifiées de l'ensemble des composants du système. Notre abstraction est une traduction de propriétés CTL en structure de Kripke abstraite. Nous avons montré qu'à partir de l'abstraction de chacun des composants du systèmes, nous obtenons une abstraction du système complets. L'abstraction et le système concret sont lié par une relation de simulation, par conséquent l'ensemble des propriétés ACTL vraies dans le modèle abstrait est préservé dans le système concret. En revanche une propriété fausse peut ne pas être une "vraie fausse" propriété.

Cette algorithme est la base pour la mise en place du boucle de raffinement d'abstraction par analyse de contre-exemples. A chaque itération de la boucle, le raffinement se fera à l'aide des spécifications de chacun des composants.

Chapitre 8

Application à la plate-forme VCI-PI

Nous illustrons la méthode exposée au chapitre précédent sur la plate-forme décrite au chapitre 5 avec le wrapper B. Etant donnée une propriété globale (ACTL) mettant en oeuvre plusieurs composants, chaque composant est abstrait selon un ensemble choisi de propriétés CTL de sa spécification. La propriété globale est alors vérifiée sur le système complet abstrait.

Voici trois des propriétés globales que nous avons vérifiées (étendues à n initiateurs et m cibles) :

Propriété 1 : **AG**(initiator[i]_state = INIT_TRANS ⇒ **AF**(bus_arbiter_signal_gnt[i] = 1)) : Lorsque l'initiateur VCI initie une transaction, le contrôleur de bus finira par accorder l'accès au bus.

Propriété 2 : **AG**(initiator[i]_state = INIT_TRANS_W ⇒ **AF**(master_wrapper[i]_state = WRITE)) : Lorsque l'initiateur VCI initie une transaction d'écriture, le wrapper maître commencera une transaction d'écriture.

Propriété 3 : **AG**(initiator[i]_state = TRANS ⇒ **AF**(target[j]_signal_rsp = 1)) : Lorsque l'incitateur VCI réalise une transaction, la cible VCI répondra.

Dans chacune de ces propriétés, les propositions atomiques concernent seulement deux composants. Mais les autres composants peuvent jouer un rôle dans la suite d'actions du protocole de communication :

- Propriété 1 : Ses propositions atomiques portent sur l'initiateur VCI et le contrôleur du PI bus. Leur communication est réalisée par l'intermédiaire du wrapper maître.
- Propriété 2 : Ses propositions atomiques portent sur l'initiateur VCI et le wrapper maître. Le protocole de communication implique aussi le contrôleur du PI bus.
- Propriété 3 : Ses propositions atomiques portent sur l'initiateur VCI et la cible VCI. Le protocole de communication implique aussi le contrôleur de bus, les wrappers maîtres et esclaves.

8.1 Détails sur la boucle CEGAR de la propriété 1

Le système complet est vérifié à l'aide du model checker VIS [gro96]. L'ensemble des propositions qui ne sont pas concernées par la propriété à vérifier sont libres. Une variable libre correspond à la valeur *indéterminée* de notre méthodologie. Dans nos modèles cette

valeur est exprimée par une valeur non déterministe et le model checker VIS représente toutes les possibilités lors du calcul. Par conséquent, si une valeur d'un signal *indéterminé* implique que la propriété n'est pas vérifiée par notre système abstrait, VIS produit un contre-exemple. Si le contre-exemple n'a pas de chemin correspondant dans le système complet alors le signal *indéterminé* n'est pas assez contraint. Nous devons alors ajouter une contrainte sous forme de propriété CTL pour restreindre les comportements de ce signal dans notre modèle abstrait. C'est la même idée que le model checking optimiste et pessimiste utilisés dans le model checking généralisé de Bruns et Godefroid [BG00].

Nous allons appliquer la boucle su CEGAR pour vérifier la propriété globale 1 :

$$\mathbf{AG}(initiator[i]_state = INIT_TRANS \Rightarrow \mathbf{AF}(bus_arbiter_signal_gnt[i] = 1))$$

La première étape consiste à abstraire le système complet en affectant une valeur non déterministe à l'ensemble des signaux du système. Ensuite nous allons contraindre les signaux qui jouent un rôle dans le processus de vérification de la propriété globale . Pour cela, chaque composant qui dirige un signal "utile" est abstrait par une partie de sa spécification qui contraint ce signal. Dans notre exemple les différents composants à abstraire sont :

- l'initiateur VCI qui contraint le signal initiant la transaction ;
- le wrapper maître qui transmet la demande de l'initiateur VCI au contrôleur de bus ;
- Le contrôleur de bus qui contraint le signal autorisant la transaction.

Pour chacun de ces composants nous avons choisi arbitrairement des formules de leur spécification afin d'obtenir une abstraction, qui sera le point de départ de la boucle de CEGAR.

```
#** INITIATEUR VCI **#
AG AF (m_cmd_val = 1);

#** WRAPPER MAITRE **#
AG AF (m_pi_req = 1);

#** CONTROLEUR DU BUS **#
AG AF (m_pi_gnt = 1);
```

Fig. 8.1 – Initialisation de la boucle CEGAR

La spécification de l'initiateur dit qu'il y a aura infiniment souvent des demandes de transactions, cette propriété ne fait aucune hypothèse sur le comportement de l'environnement. Nous pouvons alors construire la structure de Kripke abstraite de l'initiateur VCI K_i.

Le wrapper maître enverra toujours une requête au contrôleur de bus avec l'hypothèse qu'il recevra un jour une demande de l'initiateur VCI. Nous construisons la structure de Kripke correspondante K_w.

Le contrôleur de bus enverra toujours la permission d'émettre au wrapper maître avec l'hypothèse que le signal requête du wrapper sera activé.

Les trois composants sont compatibles, nous pouvons exécuter le model checking sur la composition de ces trois abstractions. Le résultat montre que la propriété est fausse, cela provient d'une mauvaise séquence d'événements. L'analyse du contre-exemple exhibe un chemin où le signal qui donne l'accès au bus est positionné avant la demande de transaction de l'initiateur VCI, puis il existe un chemin infini où signal est à 0 et où l'initiateur demande une transaction. Le comportement des signaux n'est pas assez contraint. Il faut garantir que le signal du contrôleur de bus ne peut être positionné qu'après une demande de requête du wrapper maître et qu'une requête du wrapper maître ne soit effective qu'après une demande de l'initiateur VCI. Nous choisissons deux nouvelles propriétés de la spécification du wrapper maître et du contrôleur de bus afin d'obtenir deux nouvelles structures de Kripke abstraites K'_w et K'_b.

```
#** WARPPER MAITRE **#
AG A((m_cmd_val = 0) U ((cmd_val = 1) * AF (m_pi_req = 1)));

#** CONTROLEUR DU BUS **#
AG A((m_pi_req = 0) U ((m_pi_req = 1) * AF (m_pi_gnt = 1)));
```

FIG. 8.2 – Premier raffinement de la boucle CEGAR

La propriété globale est toujours fausse. Par l'analyse du contre-exemple, nous remarquons que notre abstraction n'est toujours pas assez contrainte. Nous avons contraint les signaux du contrôleur de bus et du wrapper maître à devenir actif après un certain événement mais il n'a aucune contrainte sur leur valeur avant l'événement. Nous ajoutons alors deux propriétés de la spécification des composants concernés qui forcent l'autorisation du bus et la demande de requête à rester inactif tant que l'événement n'est pas arrivé. Nous obtenons alors deux nouvelles structures de Kripke abstraites K''_w et K''_b.

```
#** WARPPER MAITRE **#
AG A((m_pi_req = 0) U (m_cmd_val = 1));

#** CONTROLEUR DU BUS **#
AG A((m_pi_gnt = 0) U (m_pi_req = 1)));
```

FIG. 8.3 – Second raffinement de la boucle CEGAR

Le modèle abstrait est maintenant composé de la manière suivante : $K_i \parallel (K_w \parallel K'_w \parallel K''_w) \parallel (K_b \parallel K'_b \parallel K''_b)$. Le model checking est exécuté avec succès, d'après la propriété 7.5 la propriété globale est aussi valide sur le système concret.

8.2 Résultats de l'abstraction

Nous avons procédé de la même façon pour chacune des propriétés globales. La vérification de la propriété 3 impliquent tous les composants, elle est composée de 10 propriétés.

Platform name	FSM depth	Number of BDD variables	BDD size (\# of nodes)	Number of Reachable states	Reachable states space analysis time	Property	Checking time
Concrete 1 master 1 slave	475	289	34 578	8,56E+06	50,61s	1	6,75s
						2	6,70s
						3	6,88s
All abstract	7	261	522	2,73E+16	2,1s	1	0,1s
	8	292	796	1.42E+21	5,3s	2	0,2s
	10	302	1 002	1,69E+22	8s	3	0,25s
Concerned component abstracted	12	360	2 296	8,77E+15	4,26s	1	0,31s
	12	320	1 982	5,63E+15	6,22s	2	0,36s
	10	302	1 002	1,69E+22	8s	3 (all)	0,25s
Concrete 2 masters 1 slave	604	436	161 846	3,10E+10	43min	1	46min
						2	19min
						3	48min
All abstract	10	395	1 005	2,83E+20	3,5s	1	0,8s
	11	442	1 079	4,93E+24	3,8s	2	0,52s
	14	501	1564	2,50E+26	12,1s	3	8,25s
Concerned component abstracted	12	532	14 025	6,49E+24	35,21s	1	0,46s
	12	544	14 492	7.49E+24	33,77s	2	0,47s
	14	501	1 564	2,50E+26	12,1s	3	8,25s

Tab. 8.1 – Résultats comparatifs des modèles concrets et des modèles abstraits

cette propriété a mis en évidence une erreur dans l'implémentation du protocole VCI entre le wrapper maître et la cible VCI.

Le tableau résume les profits en terme d'espace mémoire requit et l'espace d'états pour les trois propriétés globales. Les expériences ont été réalisées sur Pentium IV de 3.20GHz avec 2Go de RAM. Le model checking a été réalisé avec une méthode de réordonnancement dynamique utilisant l'algorithme sift. Nous avons d'abord passé les propriétés sur deux plates-formes concrètes. La première contient un initiateur VCI, un wrapper maître et un wrapper cible, un PI bus et une cible VCI. La seconde plate-forme contient en plus un second initiateur VCI et un second wrapper maître , de plus le contrôleur du PI bus a été modifié pour gérer les deux initiateurs. Ensuite le model checking a été appliqué sur deux types d'abstraction. La première correspond au cas où tous les composants ont été abstraits. Nous sommes parties d'une abstraction vide, c'est à dire tous les signaux d'interfaces des composants sont à une valeur indéterminée puis les signaux sont contraints par les abstractions des composants. Les composants dont les propositions atomiques n'interviennent pas dans la validité de la propriété n'apparaissent pas dans cette abstractions leurs signaux sont libres. Pour la seconde abstraction, nous sommes parties du composant concret et nous avons abstrait seulement les composants qui jouent un rôle dans la réalisation de la propriétés.

Il est clair que nous avons un gain en temps lors de le calcul des états accessibles et le model checking des trois propriétés globales sur les systèmes abstraits. Cela est dû à la réduction de la profondeur de l'arbre d'exécutions exploré (colonne FSM depth). Le

nombre de variables BDD n'est pas vraiment réduit mais le fait que de nombreux signaux sont libres dans l'abstraction impliquent que les structures de BDD sont plus petites (en nombre de nœuds).

Conclusions et perspectives

Conclusions

Dans cette thèse, nous avons formalisé une méthode incrémentale pour la conception et la vérification de composants matériels. Nous avons montré que cette approche fixe un cadre de conception proche de celui du concepteur. Il peut alors s'attaquer à un problème à la fois et ajouter les nouveaux comportements, étape par étape jusqu'à obtenir une implémentation complète du composant. Notre méthode garantit la non-régression des comportements existants tout au long de la conception d'un composant. De plus, la conception incrémentale permet de faire évoluer la spécification d'un composant. Nous avons énoncé trois théorèmes de transformations de propriétés CTL pour dériver une partie de la spécification de n'importe quel composant construit de manière incrémentale. Dans le cas particulier où l'incrément est lui aussi spécifié et le retour aux anciens comportements est identifiable, il est possible d'obtenir la spécification complète du composant incrémenté. Nous avons ensuite particularisé cette démarche aux architectures pipeline. L'ensemble des incréments possible a été formalisé. Nous avons pu alors exhiber quatre nouveaux théorèmes portant sur la préservation et la transformation de propriétés CTL pour ce type d'architecture. La conception incrémentale se rapproche des méthodes d'intégrations de services mais dans notre approche met en exergue le lien entre propriétés et composants. La conception incrémentale a été appliquée à la conception des wrappers VCI-PI et VCI-AMBA. Les résultats de vérifications sur une plate-forme composée de wrapper VCI-PI montrent que les transformations automatiques et les préservations de propriétés CTL permettent de gagner beaucoup de temps pendant la phase de model-checking.

Dans une seconde partie, nous nous sommes attaquées aux problème de la vérification de systèmes composés par abstraction de composants. Notre démarche repose sur une approche de vérification CEGAR, où l'ensemble des composants est déjà vérifié. Nous avons proposé un algorithme d'abstraction linéaire basé sur la spécification de chaque composant. L'algorithme construit une structure de Kripke abstraite qui représente une (ou plusieurs) formule(s) CTL φ. Nous avons montré que cette structure abstraite simule toute structure de Kripke vérifiant φ. Les premiers résultats mettent en évidence un gain certain en temps de vérification. Pour l'instant, l'algorithme d'abstraction et la boucle de raffinement n'ont pas été encore implémentés.

Perspectives

Nos travaux futurs sur la conception incrémentale porteraient sur le développement d'un atelier de conception. Nous aimerions proposer au concepteur un outil d'intégration d'incrément. L'utilisateur pourra, d'une part, définir les incréments et les ajouter à la main. Notre atelier vérifierait alors que les règles de la conception incrémentale sont respectées. D'autre part, il serait intéressant de proposer une intégration automatique des incréments.

Nos perspectives de recherches portent sur l'approche CEGAR pour la vérification de SOC. La première étape est d'implémenter l'algorithme d'abstraction. L'idée est de construire une structure directement utilisable par le model checker VIS. Pour restreindre la taille des structure de Kripke abstraite, l'emploi de structures de données symboliques nous semble indispensable.

Ensuite, nous aimerions mettre en place la boucle de raffinement d'abstraction à partir du contre-exemple. Lorsqu'une propriété n'est pas satisfaite par un modèle abstrait, le model checker fournit une trace d'exécution du modèle abstrait. Ce contre-exemple doit être analysé afin de déterminer si il correspond à une exécution possible du composant concret ou si c'est un faux contre-exemple (*spurious*). Le lien entre le contre-exemple abstrait et le modèle concret doit être clairement déterminé. Dans le cas d'un faux contre-exemple, nous devons contraindre itérativement le modèle abstrait en ajoutant de nouvelles propriétés afin de raffiner l'abstraction. Pour mettre en place cela nous devons répondre aux questions suivantes :

- Comment détecter un faux contre-exemple ?
- Comment obtenir un ensemble de propriétés plus fortes (prenant en compte plusieurs composants) à partir du contre-exemple ?

Il nous faut alors définir comment choisir l'ensemble de propriétés pertinentes à ajouter. Nous aimerions étudier différentes heuristiques pour choisir un sous-ensemble de propriétés pour chacun des composants afin de d'éliminer les faux contre-exemples dû à l'abstraction. Cela nous amène à une autre question : comment peut on renforcer l'abstraction si la spécification des composants est incomplète ? Dans ce cas, nous pouvons adopter l'approche standard développée pour le processus de vérification CEGAR, ou faire intervenir l'utilisateur pour compléter la spécification des composants. Cette dernière idée pose la question de la complétude et de la consistance d'une spécification.

Nous aimerions aussi nous positionner dans un cadre plus large de validation des SOC. Nous souhaiterions mettre en place une collaboration avec l'Université TU de Darmstadt dans l'idée d'appliquer nos méthodes d'abstractions au model checking et à la simulation.

Annexe A

Theorem 1 partial proof : Proofs of each basic cases

Let s' enriches s and p not concerned by the increment ($p \in K(W_i)$), W_i is a Moore machine at a step i of the design process, we have :

$K(W_i), s \models p \Leftrightarrow K(W_{i+1}), s' \models p$

Proof (⇒) By definition, if s' enriches s, s' contains a greater set of atomic propositions than s. As $s \models p$, p being an atomic proposition of s, then p is an atomic proposition of s', hence $s' \models p$.

(⇐) If p is not an atomic proposition concerned with the increment and $s' \in S_{K(W_{i+1})}$ enriches $s \in S_{K(W_i)}$, then $K(W_{i+1}), s' \models p \Rightarrow K(W_i) \models p$.

■

$K(W_i), s \models p \vee q \Leftrightarrow K(W_{i+1}), s' \models p \vee q$

Proof The proof proceeds as the previous one. ■

$K(W_i), s \models EXp \Leftrightarrow K(W_{i+1}), s' \models (e_qt \wedge EXp)$

Proof (⇒) If $s \models EXp$, there exists a state $t = (u, c) \in S_{K(W_i)}$ such that $s \to t$ and $t \models p$. Let be a state s' enriching s with e_qt; by Corollary 1 item 2 there exists $t' = (u', c') \in S_{K(W_{i+1})}$ such that $s' \to t'$, $u = u'$, and $proj(c', I_i) = c$. Hence $K(W_{i+1})$, $t' \models p$ and $K(W_{i+1})$, $s' \models e_qt \wedge EXp$.

(⇐) Let be $K(W_{i+1})$,$s' \models e_qt \wedge EXp$ and s' enriches s with e_qt; let be $t' = (u', c')$ such that $s' \to t'$ and $t' \models p$. By Corollary 1 item 4, there exists $t = (u, c) \in S_{K(W_i)}$, such that $u = u'$, and $proj(c', I_i) = c$, then as $p \in AP_{K(W_i)}$, $t \models p$. Moreover, s' simulates s, hence $K(W_i), s \models EXp$.

■

$K(W_i), s \models EFp \Leftrightarrow K(W_{i+1}), s' \models E[e_qtUp]$.

Proof (⇒) If $s \models p$, $s' \models p$ then $s' \models EFp$.

If $s \not\models p$, there exists a path σ in $K(W_i)$: $\sigma = s \to t \to \ldots \to r$, such that $r \models p$. By

Corollary 1 item 1, there exists a path σ' in $K(W_{i+1}) : s' \rightarrow t' \rightarrow \ldots \rightarrow r'$ such that s' enriches s with e_qt, t' enriches t with e_qt,..., and r' enriches r then $r \models p$, hence $s' \models E[e_qtUp]$.

($\Leftarrow$) Let $K(W_{i+1}), s' \models E[e_qtUp]$ and s' enriches s with e_qt. There exists a path σ' in $K(W_{i+1}) : s' \rightarrow t' \rightarrow \ldots \rightarrow r'$ such that for all $s' \leq u' < r'$, $u' \models e_qt$ and $r' \models p$. By Corollary 1 item 4, there exists a path σ in $K(W_i) : s \rightarrow t \rightarrow \ldots \rightarrow r$ such that for all $s \leq u < r$ u' simulates u and r' enriches r. Hence $r \models p$ if p is not concerned by the increment and $K(W_i), s \models EFp$.

■

$K(W_i), s \models EGp \Leftrightarrow K(W_{i+1}), s' \models EG(e_qt \wedge p)$.

Proof ($\Rightarrow$) If $K(W_i), s \models EGp$ there exists an infinite path σ in $K(W_i)$ s.t. $s \rightarrow t \rightarrow \ldots \rightarrow r \rightarrow \ldots$, and such that $s \models p$, $t \models p$, ..., $r \models p$, ... By Corollary 1 item 1, there exists an infinite path σ' in $K(W_{i+1}) : s' \rightarrow t' \rightarrow \ldots \rightarrow r' \rightarrow \ldots$, such that $s' \models p \wedge e_qt$, $t' \models p \wedge e_qt$, ..., $r' \models p \wedge e_qt$,... : $s' \models EG(e_qt \wedge p)$.

($\Leftarrow$) Let $K(W_{i+1}), s' \models EG(e_qt \wedge p)$ and s' enriches s with e_qt, $s \in S_{K(W_i)}$. There exists an infinite path σ' in $K(W_{i+1})$ $s' \rightarrow t' \rightarrow \ldots \rightarrow r' \rightarrow \ldots$ such that for all state u' of this path $u' \models p \wedge e_qt$. By Corollary 1 item 4, there exists an infinite path σ in $K(W_i) : s \rightarrow t \rightarrow \ldots \rightarrow r \rightarrow \ldots$ such that s' enriches s with e_qt, t' enriches t with e_qt ..., hence if p is not concerned by the increment, $s \models p$, $t \models p$. Hence, $K(W_i), s \models EGp$.

■

$K(W_i), s \models E[pUq] \Leftrightarrow K(W_{i+1}), s' \models E[e_qt \wedge pUq]$.

Proof ($\Rightarrow$) If $s \models q$, then $s' \models q$ hence $s' \models E[(e_qt \wedge p)Uq]$. If $s \not\models q$, there exists an infinite path $\sigma : s \rightarrow t \rightarrow \ldots \rightarrow r \rightarrow \ldots$, such that $s \models p$, $t \models p$, ..., $r \models q$. Let s' enriches s with e_qt, by Corollary 1 item 1, there exists an infinite path σ' in $K(W_{i+1}) : s' \rightarrow t' \rightarrow \ldots \rightarrow r' \rightarrow \ldots$, such that $s' \models p \wedge e_qt$, $t' \models p \wedge e_qt$, ..., r' enriches r and $r' \models q$. Hence, $s' \models E[(e_qt \wedge p)Uq]$.

($\Leftarrow$) Let $K(W_{i+1}), s' \models E[(e_qt \wedge p)Uq]$ and s' enriches s with e_qt, $s \in S_{K(W_i)}$. There exists an infinite path σ' in $K(W_{i+1}) : s' \rightarrow t' \rightarrow \ldots \rightarrow r' \ldots$ such that $s' \models p \wedge e_qt$, $t' \models p \wedge e_qt$, ..., $r' \models q$. By Corollary 1 item : 4, there exists an infinite path in $K(W_i)$ $\sigma = s \rightarrow t \rightarrow \ldots \rightarrow r \rightarrow \ldots$ such that s' enriches s with e_qt, t' enriches t with e_qt, ..., r' enriches r. If p and q are not concerned by the increment, σ satisfies $[pUq]$, hence $K(W_i), s \models E[pUq]$.

■

$K(W_i), s \models AXp \Leftrightarrow K(W_{i+1}), s' \models (e_qt \Rightarrow AXp)$.

Proof ($\Rightarrow$) In $K(W_i)$, p holds for all successors r of s. If s' enriches s with e_qt, for all successors r' of s', there exists a successor r of s such that r' enriches r (Corollary 1 item 3) hence $r' \models p$. Else (if s' enriches s with e_act), $s' \models e_act$, and $e_act = \neg e_qt$. Hence $s' \models e_qt \Rightarrow AXp$.

($\Leftarrow$) $K(W_{i+1}), s' \models e_qt \Rightarrow AXp$. s' enriches s, either with e_act (nothing to be said), or with e_qt (hypothesis). By Corollary 1 item 4, all successors of s' enriches states that are successors of s in $K(W_i)$ (with e_qt or e_act). All of them must verify p, hence $s \models AXp$.

■

$K(W_i), s \models AFp \Leftrightarrow K(W_{i+1}), s' \models AF(e_act \vee p)$.

Proof ($\Rightarrow$) In $K(W_i)$ for all infinite path $\sigma = s_0, \ldots s_n \ldots$, there exists a state s_k, $0 \leq k \leq n$ in which p is true. From Corollary 1 item 1, there exists some path in $K(W_{i+1})$ $\sigma' = s'_0, \ldots, s'_i, \ldots s'_n$, such that all the states s'_i enriches s_i with e_qt. Moreover, by constructing $K(W_{i+1})$ we have that there doesn't exist any transition t'_k from a state s'_k in σ' to a state s'_{k+1} labeled with e_qt and which is *n*ot in the following induction hypothesis :
If $s'_k \in \sigma'$ then $s'_k \models AF(p \vee e_act)$
From s'_0, there exists a path such that all the states verify e_qt and AFp, hence $s'_0 \models AF(p \vee e_act)$.
Let $s'_k \in \sigma'$, the set of these successors are as :
$s'_{k+1} = s_{k+1} \wedge e_act$ and thus verify AF(p ∨ e_act) or
$s'_{k+1} = s_{k+1} \wedge e_qt$,the transition from $s_k \rightarrow s_{k+1}$ is in σ' and thus verify $AF(p \vee e_act)$ (induction hypothesis).

($\Leftarrow$) Let s'_0 be a state in $S_{K(W_{i+1})}$ s.t. $s'_0 \models AF(e_act \vee p)$, and s'_0 enriches s_0 with e_qt.

1. if $s'_0 \models p$ then $s_0 \models p$, hence $s_0 \models AF(p)$.
2. if $s'_0 \not\models p$ then there exists three categories of successors : $t' \models e_qt \wedge p$; $t'' \models e_act$; $r' \models e_qt \wedge \neg p$ and $r' \models AF(e_act \vee p)$. By Corollary 1 item 4, there exists t and r in $S_{K(W_i)}$ s.t. they are successors of s_0 and $t \models p$ and one can prove by induction that $r \models AFp$. Hence $s_0 \models AFp$.

■

$K(W_i), s \models A[pUq] \Leftrightarrow K(W_{i+1}), s' \models A[pU((e_act \wedge p) \vee q)]$.

Proof ($\Rightarrow$)

1. Let be $s_0 \in S_{K(W_i)}$, if $s_0 \models q$, then we have $s_0 \models A[pUq]$. Let be $s'_0 \in S_{K(W_{i+1})}$, s'_0 enriches s_0 with e_qt then $s'_0 \models q$, hence $s'_0 \models A[pU((e_act \wedge p) \vee q)]$.
2. If $s_0 \not\models q$, $s_0 \models p$ and all its successors $Succ(s_0)$ verify $A[pUq]$. Let r in $Succ(s_0)$ be s.t. $r \models q$, then we have r' enriches r with e_qt verifies q. Hence $r' \models A[pU((e_act \wedge p) \vee q)]$. Let t in $Succ(s_0)$ be s.t. $t \models p$, then $\exists t', t''$ in $S_{K(W_{i+1})}$ s.t $t' \models p \wedge e_qt$ and $t'' \models p \wedge e_act$. By induction, one can prove $t' \models A[pU((e_act \wedge p) \vee q)]$. Hence all successors of s'_0 verify $s' \models, A[pU((e_act \wedge p) \vee q)]$ and s'_0 verifies it also.

($\Leftarrow$) Let s'_0 be a state in $S_{K(W_{i+1})}$ such that $s'_0 \models A[pU((e_act \wedge p) \vee q)]$ and s'_0 enriches s_0 with e_qt.

1. If $s'_0 \models q$ then $s_0 \models q$ and $s_0 \models A[pUq]$.
2. If $s'_0 \models p \wedge e_act$, contradiction with the hypothesis.

3. If $s'_0 \models p$ then there exists 3 categories of successor states : (see Figure ??) $t' \models p \wedge e_qt$, $t'' \models p \wedge e_act$ and $r' \models q$. Moreover, t' and t'' verify $A[pU((e_act \wedge p) \vee q)]$. By Corollary 1 item 4, there exists t and r in $S_{K(W_i)}$ s.t. they are successors of s_0 and $t \models p$ and $r \models q$. By incremental construction s_0 can not have a successor that verify $(\neg p \wedge \neg q)$ and one can prove, by induction that t verifies $A[pUq]$. Hence s_0 verifies $A[pUq]$.

■

$K(W_i), s \models AGp \Leftrightarrow K(W_{i+1}), s' \models A[pW(e_act \wedge p)]$.

Proof ($\Rightarrow$) 1. *(In $K(W_{i+1})$, a state that do not verify p belongs to an added behaviour).* Let be $t' \in S_{K(W_{i+1})}$, $t' \models \overline{p} \wedge e_qt$ and $\sigma' = s' \dots t'$. $t' = (u', c')$ doesn't simulate a state in $S_{K(W_i)}$ $(\forall s \in S_i\ s \models p)$. u' corresponds to an added state into $W_{i+1} (\in \Sigma_+)$. Hence, t' is only reachable from a sequence having a state where e_act holds (Corollary 1 item 3).

2. *In $K(W_{i+1})$, along paths reached from the initial state, all states verify e_qt and p until a state where e_act holds is reached.* Let be s' such that s' enriches s and a sequence $\sigma' = s' \dots r' \dots t'$ with $s' < r' \leq t'$, if $\nexists m'$ labeled with e_act such that $m' < r'$ then $r' \models p \wedge e_qt$ or $r' \models p \wedge e_act$.

3. *Infinite paths in $K(W_i)$ correspond to infinite paths labeled with e_qt in $K(W_{i+1})$.* By Corollary 1 item 1, if there exists some infinite path in $K(W_i)$, there exists some infinite path in $K(W_{i+1})$ labeled with e_qt. Then there exists some infinite path in $K(W_{i+1})$ which verify $p \wedge e_qt$.
We have thus $s' \models A[pW(e_act \wedge p)]$

($\Leftarrow$) Let s'_0 be a state in $S_{K(W_{i+1})}$ such that $s'_0 \models A[pU((e_act \wedge p) \vee q)]$ and s'_0 enriches s_0 with e_qt. All successor of s'_0 are such that : $t' \models p \wedge e_qt$ or $t'' \models p \wedge e_act$ and verifies $A[pU((e_act \wedge p) \vee q)]$. If p is not concerned by the increment, By Corollary 1 item 4 there exists $t \in S_{K(W_i)}$ such that $t \models p$. By incremental construction s'_0 can not have a successor were $\neg p$ holds and one can prove, by induction that t verifies AGp. Hence s_0 verifies AGp.

■

$K(W_i), s \models A[pWq] \Leftrightarrow K(W_{i+1}), s' \models A[pW((e_act \wedge p) \vee q)]$.

Proof ($\Rightarrow$) $s \models A[pWq]$ then all paths from s are such they verified pUq or Gp. In the first case, we use the same reasoning as $A[pUq]$. In the second case, by Corollary 1 item 1 these paths exist in $K(W_{i+1})$. The divergent behaviours are labeling with e_act (Corollary 1 item 3). We have thus $K(W_{i+1}), s' \models$ $A[pW((e_act \wedge p) \vee q)]$

($\Leftarrow$) Same reasoning as $A[pU((e_act \wedge p) \vee q)]$.

■

$K(W_i), s \models \neg p \Leftrightarrow K(W_{i+1}), s' \models \neg p$.

Proof The proof proceeds as the one concerning positive atomic propositions. ■

$K(W_i), s \models \neg\Psi \Leftrightarrow K(W_{i+1}), s' \models \neg\Psi'$.

Proof The proof is performed by first transforming the formulae into positive form and then applying the transformation proven above. We compare the result with the one obtain by transforming directly the negative form. The weak until operator (W) is related to the strong until operator (U) by the following equivalences :

$$A[pWq] = \neg E[\neg qU(\neg p \wedge \neg q)]$$

$$E[pUq] = \neg A[\neg qW(\neg p \wedge \neg q)]$$

We recall here that $\neg e_act = e_qt$

$$\begin{aligned}
(\neg EXp)' &= (AX\neg p)' \\
&= e_qt \Rightarrow AX\neg p \\
\neg(EXp)' &= \neg(e_qt \wedge EX\neg p) \\
&= \neg e_qt \vee AX\neg p \\
&= e_qt \Rightarrow AX\neg p \\
Hence \quad (\neg EXp)' &= \neg(EXp)'
\end{aligned} \tag{1}$$

$$\begin{aligned}
(\neg EFp)' &= (AG\neg p)' \\
&= A[\neg pW(e_act \wedge \neg p)] \\
\neg(EFp)' &= \neg E[e_qtUp] \\
&= A[\neg pW(e_act \wedge \neg p)] \\
Hence \quad (\neg EFp)' &= \neg(EFp)'
\end{aligned} \tag{2}$$

$$\begin{aligned}
(\neg EGp)' &= (AF\neg p)' \\
&= AF(e_act \vee \neg p) \\
\neg(EGp)' &= \neg EG(e_qt \wedge p) \\
&= AF(e_act \vee \neg p) \\
Hence \quad (\neg EGp)' &= \neg(EGp)'
\end{aligned} \tag{3}$$

$$
\begin{aligned}
(\neg E[pUq])' &= A[\neg qW(\neq p \wedge \neg p)]' \\
&= A[\neg qW((e_act \wedge \neg q) \vee (\neg p \wedge \neg q)] \\
\neg(E[pUq])' &= \neg E[(e_qt \wedge p)Uq] \\
&= A[\neg qW \neg(\neg(e_qt \wedge p) \wedge \neg q)] \\
&= A[\neg qW \neg((e_act \vee \neg p) \wedge \neg q)] \\
&= A[\neg qW((e_act \wedge \neg q) \vee (\neg p \wedge \neg q)] \\
Hence \quad & (\neg E[pUq])' = \neg(E[pUq])'
\end{aligned} \tag{4}
$$

$$
\begin{aligned}
(\neg E[pWq])' &= A[\neg qU(\neq p \wedge \neg p)]' \\
&= A[\neg qU((e_act \wedge \neg q) \vee (\neg p \wedge \neg q)] \\
\neg(E[pWq])' &= \neg E[(e_qt \wedge p)Wq] \\
&= A[\neg qU \neg(\neg(e_qt \wedge p) \wedge \neg q)] \\
&= A[\neg qU \neg((e_act \vee \neg p) \wedge \neg q)] \\
&= A[\neg qU((e_act \wedge \neg q) \vee (\neg p \wedge \neg q)] \\
Hence \quad & (\neg E[pWq])' = \neg(E[pWq])'
\end{aligned} \tag{5}
$$

$$
\begin{aligned}
(\neg AXp)' &= (EX\neg p)' \\
&= e_qt \wedge EX\neg p \\
\neg(AXp)' &= \neg(e_qt \Rightarrow AX\neg p) \\
&= e_qt \wedge EX\neg p \\
Hence \quad & (\neg AXp)' = \neg(AXp)'
\end{aligned} \tag{6}
$$

$$
\begin{aligned}
(\neg AFp)' &= (EG\neg p)' \\
&= EG(e_qt \wedge \neg p) \\
\neg(AFp)' &= \neg AF(e_act \vee p) \\
&= EG(e_qt \wedge \neg p) \\
Hence \quad & (\neg AFp)' = \neg(AFp)'
\end{aligned} \tag{7}
$$

$$
\begin{aligned}
(\neg AGp)' &= (EF\neg p)' \\
&= E[e_qtU\neg p] \\
\neg(AGp)' &= \neg A[pW(e_act \wedge p)] \\
&= E[\neg(e_act \wedge p)U(\neg p \wedge \neg(e_act \wedge p))] \\
&= E[(e_qt \vee \neg p)U\neg p] \\
&= E[e_qtU\neg p] \\
Hence \quad & (\neg AGp)' = \neg(AGp)'
\end{aligned} \tag{8}
$$

$$
\begin{aligned}
(\neg A[pUq])' &= (E[\neg qW(\neg p \wedge \neg q)])' \\
&= E[(e_qt \wedge \neg q)W(\neg p \wedge \neg q)] \\
\neg(A[pUq])' &= \neg A[pU((e_act \wedge p) \vee q)] \\
&= E[\neg((e_act \wedge p) \vee q)W(\neg p \wedge \neg((e_act \wedge p) \vee q))] \\
&= E[((_act \wedge \neg q) \vee (\neg p \wedge \neg q))W(\neg p \wedge \neg q)] \\
&= E[(e_qt \wedge \neg q)W(\neg p \wedge \neg q)] \\
Hence \quad & (\neg A[pUq])' = \neg(A[pUq])'
\end{aligned} \tag{9}
$$

$$
\begin{aligned}
(\neg A[pWq])' &= (E[\neg qU(\neg p \wedge \neg q)])' \\
&= E[(e_qt \wedge \neg q)U(\neg p \wedge \neg q)] \\
\neg(A[pWq])' &= \neg A[pW((e_act \wedge p) \vee q)] \\
&= E[\neg((e_act \wedge p) \vee q)U(\neg p \wedge \neg((e_act \wedge p) \vee q))] \\
&= E[((_act \wedge \neg q) \vee (\neg p \wedge \neg q))U(\neg p \wedge \neg q)] \\
&= E[(e_qt \wedge \neg q)U(\neg p \wedge \neg q)] \\
Hence \quad & (\neg A[pWq])' = \neg(A[pWq])'
\end{aligned} \tag{10}
$$

■

Annexe B

Wrappers VCI-PI

B.1 Wrapper maître A

B.2 Wrapper maître B

B.3 Wrapper maître B’

B.4 Wrapper maître C’

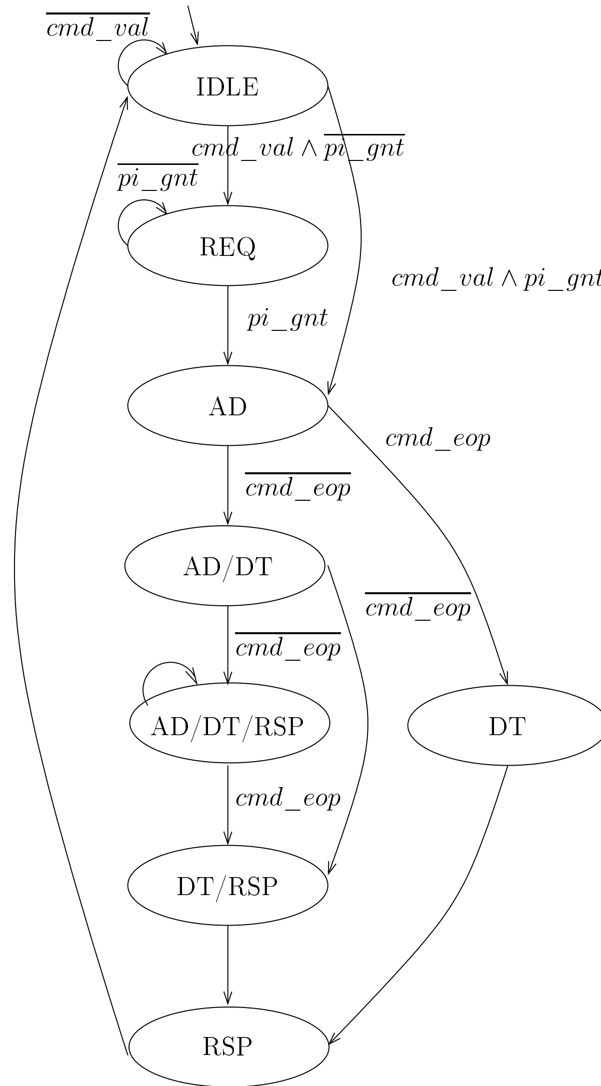


Fig. B.1 – Automate du wrapper A

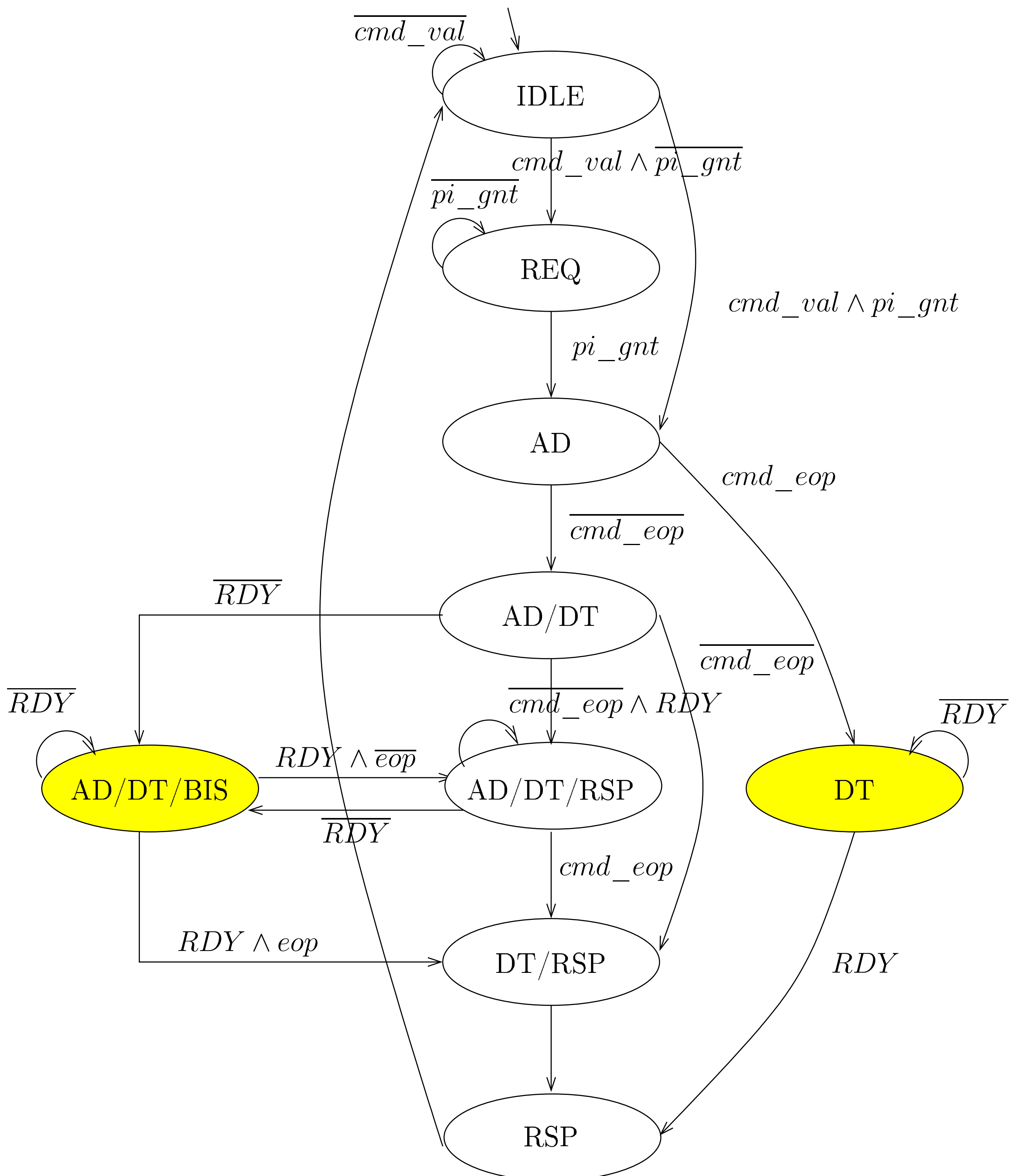


FIG. B.2 – Automate du wrapper B

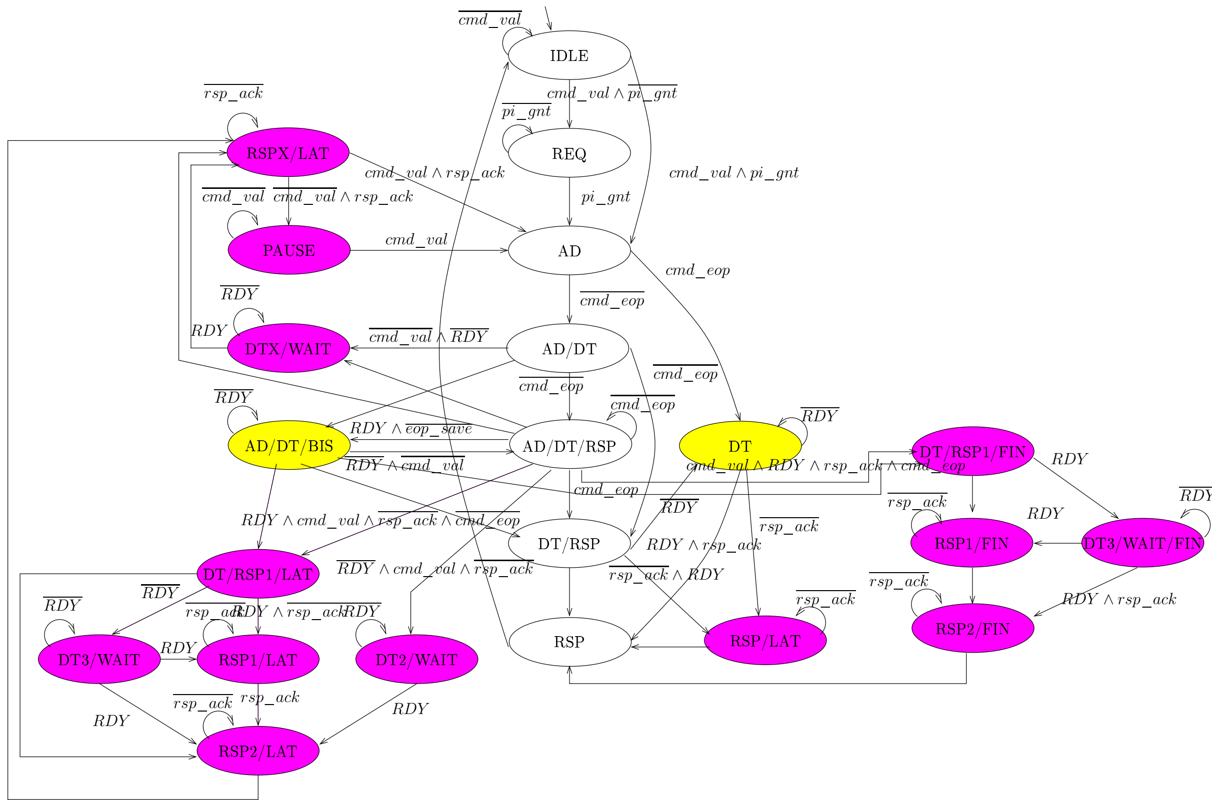


FIG. B.3 – Automate du wrapper B'

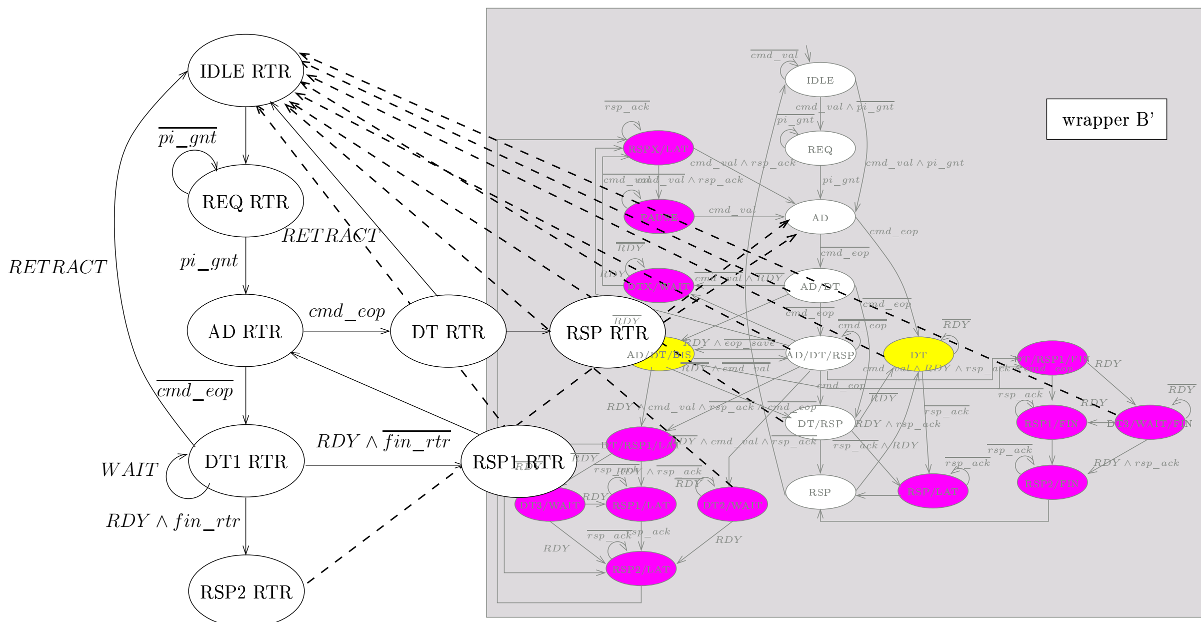


FIG. B.4 – Automate de l'incrément RETRACT et des connexions avec B'

Annexe C

Abstraction de composant

The proof of Property 7.3 stating that K_φ is a model of φ needs preliminary properties about the structure we obtain from Definition 7.7.

Propriété C.1 *An AKS contains only state s such that*

- $s \in S_T$.
- $s \in S_0$, *s is an initial state.*
- *s is a state obtained by a composition of states (s_f,s_g) and s_f, s_g are in* $S_T \cup S_{0_g} \cup S_{0_f}$.

Proof (Sketch) Property C.1
The proof proceed by induction over the formula. All operator except the G, U operators and the conjunction preserves the previously existing initial state and may add some new initial states. The G operator preserves all initial states and delete all states belonging to S_T. In case of composition of K_f and K_g one has to prove that all states obtained are composed with state of S_f and S_g that verifies the property. In case of an until operator $A[fUg]$, one combines K_f, K_g and $K_{f\wedge g}$; hence K_g and $K_{f\wedge g}$ respect the 3 conditions above ($K_{f\wedge g}$ by the definition of the parallel composition) ; K_f is modified in a same way as the globally operator, hence K'_f respects the 3 conditions as AGf do. The abstract Kripke structure obtained from an U operator, respects the rules of the conjunction and the G operator. ■

Property 7.3 The AKS K_φ built by definitions 7.7,7.9 and 7.10 is such that $\forall s_0 \in S_{0_\varphi}$, $K_\varphi, s_0 \models \varphi$.

Proof Property 7.3 : $\forall s_0 \in S_{0_\varphi}$, $K_\varphi, s_0 \models \varphi$.
The proof proceeds by induction over the formula length. Basic cases are CTL formulae with no nested operator.

$\varphi = \mathbf{p}$, the resulting AKS contains one initial state s_0 such that $p \in \mathcal{L}(s_0)$, hence $K_\varphi \models p$.

$\varphi = \mathbf{p} \wedge \mathbf{q}$, the resulting AKS is built from K_p and K_q. The initial state s_0 is such that $s_0 = (s_{0_p}, s_{0_q})$. We have $p \in \mathcal{L}(s_{0_p})$ and $q \in \mathcal{L}(s_{0_q})$, thus p and $q \in (s_0)$, hence, $K_\varphi \models p \wedge q$.

$\varphi = \mathbf{p} \vee \mathbf{q}$, the resulting AKS contains two initial states s_{0_p} and s_{0_q} such that
- $p \in \mathcal{L}(s_{0_p}$, thus $s_0 \models p \vee q$.
- $q \in \mathcal{L}(s_{0_q})$, thus $s'_0 \models p \vee q$.

The formula φ holds for all initial state of the resulting AKS.

$\varphi = \mathbf{AFp}$, the resulting AKS K_{AFp} contains exactly two initial states s_0 and s'_0 such that
- $p \in \mathcal{L}(s_0)$, thus $s_0 \models AFp$.
- $p \notin \mathcal{L}(s'_0)$, there exists two outgoing transitions from s'_0, one to itself and one to s_0. As s'_0 is not in the fairness F_{AFp} s_0 will always be reach, thus $s'_0 \models AFp$

The formula φ holds for all initial state of K_{AFp}.

$\varphi = \mathbf{AGp}$, the resulting AKS is built from K_p such that initial state s_0 is the initial state of K_{AGp}. Thus p holds in the initial state of K_{AGp}. Furthermore $s_T \in S_T$ does not belong to S_{AGp} and there exists only one transition from s_0 to s_0. The formula φ holds for the single initial state of K_{AGp}.

$\varphi = \mathbf{A[pUq]}$, the resulting AKS is built from K_p, K_q and $K_{p\wedge q}$. The initial state of $K_{A[pUq]}$ $S0_{A[pUq]} = \{s_0 \in S_{0_p}\} \cup \{s'_0 \in S_{0_q}\} \cup \{s''_0 \in S_{0_{p\wedge q}}\}$.
- $q \in \mathcal{L}(s'_0)$, thus $s'_0 \models A[pUq]$.
- $p, q \in \mathcal{L}(s''_0)$, thus $s'_0 \models A[pUq]$.
- $p \in \mathcal{L}(s_0)$ and $q \notin \mathcal{L}(s_0)$ and there exists two outgoing transitions from s_0 : one to itself and one to s'_0. Moreover, s_0 is not in the fairness constraint of $K_{A[pUq]}$ thus state s'_0 will always be reached, $s_0 \models A[pUq]$.

The formula φ holds for all initial states of $K_{A[pUq]}$.

Induction Hypothesis : let K_f and K_g be two AKS such that f holds for all initial state in S_{0_f} and g holds for all initial state in S_{0_g} .

$\varphi = \mathbf{f} \vee \mathbf{g}$, the resulting AKS is built from K_f and K_g such that $S_{0_{f\vee g}} = S_{0_f} \cup S_{0_g}$. By induction hypothesis we have
- $\forall s_f \in S_{0_f}, s_f \models f$, thus $s_f \models f \vee g$
- $\forall s_g \in S_{0_g}, s_g \models g$, thus $s_g \models f \vee g$

The formula φ holds for all initial state of $K_{f\vee g}$.

$\varphi = \mathbf{f} \wedge \mathbf{g}$, the resulting AKS is built from K_f and K_g by parallel composition. By induction hypothesis f holds in all initial state in S_{0_f} and g holds in all initial state in S_{0_g}. The parallel composition insures that a path in $K_f \parallel K_g$ is a fair path if and only if its restriction to each component results in a fair path. thus $f \wedge g$ holds for all initial state of $K_{f\wedge g}$.

$\varphi = \mathbf{AFf}$, the resulting AKS is built from K_f in which a new initial state s_0 is added. There exists a transition from s_0 to itself and a transition from s_0 to all initial states of K_f. Moreover s_0 is not in the fairness constraint of K_{AFf}, thus all paths from s_0 will always reach an initial state of K_f. By induction hypothesis f holds for all initial states of K_f thus $s_0 \models AFf$. Moreover, f holds for all initial states that belonging to K_f, thus

AFf holds for all initial states of K_{AFf}.

$\varphi = \mathbf{AGf}$, the resulting AKS is built from K_f and $S_{0_{AGf}} = S_{0_f}$. By property C.1 all states in S_{AGf} are initial, or composed state.

1. By induction hypothesis, φ holds in all states $s_0 \in K_{AGf}$.
2. All composed state comes from a nested until operator, the premises of the until formulae holds in all such state, and by construction of K_{AGf} all states in S_{predT}, now reach an initial state where the second part of the property will be met.

Hence φ holds in all composed state.

$\varphi = \mathbf{A[fUg]}$, the resulting AKS is built from K_f, K_g and $K_{f\wedge g}$, and $S_{0_{A[fUg]}} = S_{0_f} \cup S_{0_g} \cup S_{0_{f\wedge g}}$.

- g holds in all initial state $s_g \in S_{0_g}$, thus $A[fUg]$ holds for all s_g.
- g and f holds in all initial states $s_{f\wedge g} \in S_{0_{f\wedge g}}$,thus $A[fUg]$ holds for all $s_{f\wedge g}$.
- f holds in all initial state $s_f \in S_{0_f}$, and all states are not in the fairness constraint of $K_{A[fUg]}$.
  - For all states $s_f \in S_{0_f} \cap S_{predT}$ there exists an outgoing transition to each initial state of S_{0_g}, $A[fUg]$ holds for all s_f.
  - For all states $s'_f \in S_{0_f} \setminus S_{predT}$ there exists an outgoing transition to each initial state of $S_{0_{f\wedge g}}$, $A[fUg]$ holds for all s'_f.
  - For all intermediary state s''_f, we prove that $s'_f \models f$ by applying the same reasoning as operator G.

■

Property 7.4 For all Kripke structure K such that $K \models \varphi$ and $K \not\models \overline{\varphi}$ there exists $K_\varphi = AKS(\varphi)$ and there exists a simulation relation $\preceq$ such that $K \preceq K_\varphi$.

Proof The proof proceeds by induction on the structure of φ, the basic cases are the atomic proposition et the CTL formulae with no nested operators. Let $K, s_0 \models \varphi$ and $K_\varphi, s'_0 \models \varphi$, $K_\varphi = AKS(\varphi)$, there exists a simulation relation $H \subseteq S_K \times S_{K_\varphi}$.

- $\varphi = \mathbf{p}$
  1. $p \in \mathcal{L}(s'_0)$ thus $\mathcal{L}(s'_0) \subseteq \mathcal{L}(s_0)$
  2. H :$(s_0, s'_0) \in H$ and for all s such that $s \xrightarrow{*} s$ in K, $(s, s_T) \in H$.
- $\varphi = \mathbf{A[pUq]}$

  $K, s_0 \models A[pUq]$ implies that all paths $\pi = s_0 \xrightarrow{*} s_j$ are such that $s_j \models q$ and for all $0 \leq i \leq j$ $s_i \models p$. In K_φ , there exists a path pi such that $\pi' = s'_0 \xrightarrow{*} s'_0 \rightarrow s'_1 \rightarrow s_T$ with $s'_1 \models q$ and $s'_0 \models p$.
  1. $\mathcal{L}(s'_i) \subseteq \mathcal{L}(s_i)\ \forall i \leq j$
  2. H is defined such that : For all $0 \leq i < j$, $(s_i, s'_0) \in H$, and $(s_j, s'_1) \in H$, and for all $k > j$, $(s_k, s_T) \in H$.

- $\varphi = \varphi_1 \wedge \varphi_2$ The abstract structure K_φ is built from K_{φ_1}, K_{φ_2}, by induction hypothesis, K_{φ_1} simulates all $K \models \varphi_1$ and K_{φ_2} simulates all $K \models \varphi_2$, By the property of the simulation relation preorder we have the following rules :

1	K	$\preceq$	K_{φ_1}	(Induction hypothesis)
2	$K \parallel K_{\varphi_2}$	$\preceq$	$K_{\varphi_1} \parallel K_{\varphi_2}$	(Parallel composition)
3	K	$\preceq$	K_{φ_2}	(Induction hypothesis)
4	$K \parallel K_{\varphi_1}$	$\preceq$	$K_{\varphi_1} \parallel K_{\varphi_2}$	(Parallel composition)
5	$K \parallel K$	$\preceq$	$K_{\varphi_1} \parallel K_{\varphi_2}$	(assume-garantee reasonning line 2 and 4)
6	K	$\preceq$	$K \parallel K$	(Parallel composition)
	K	$\preceq$	$K_{\varphi_1} \parallel K_{\varphi_2} = K_{\varphi_1 \wedge \varphi_2}$	(Simulation relation transitivity)

- $\varphi = \mathbf{A}[\varphi_1 \mathbf{U} \varphi_2]$
  The abstract structure K_φ is built from K_{φ_1}, K_{φ_2} and $K_{\varphi_1 \wedge \varphi_2}$ and by induction hypothesis all $s'_{0_2} \in S_{0_{\varphi_2}}$ simulates s_{0_2} s.t $K, s_{0_2} \models \varphi_2$, all $s'_{0_1} \in S_{0_{\varphi_1}}$ simulates s_{0_1} s.t $K, s_{0_1} \models \varphi_1$, all $(s_{0_1}, s_{0_2})' \in S_{0_{\varphi_1 \wedge \varphi_2}}$ simulates s_{0_12} s.t $K, s_{0_12} \models \varphi_1 \wedge \varphi_2$. Since $K, s_0 \models \varphi$, either

  1. $s_0 \models \varphi_2$, then there exists s'_0 such that $s'_0 \models \varphi_2$ and $s'_0 \in K_{\varphi_2}$, by induction hypothesis K_{φ_2} simulates all $K \models \varphi_2$.
  2. $s_0 \models \varphi_1$ and there exists $\pi = s_0 \xrightarrow{*} s_j$ such that $s_j \models \varphi_2$ and for all $i \leq j$ $s_i \models \varphi_1$.
     (a) If $s_i \models \varphi_1 \wedge AX(\varphi_1 \wedge \varphi_2)$, there exists $s'_i \in K_\varphi$ such that $s'_i \in K_{\varphi_1}$ and by induction hypothesis K_{φ_1} simulates all $K \models \varphi_1$. Moreover there exists an outgoing transition to an initial states of $K_{\varphi_1 \wedge \varphi_2}$ and by induction hypothesis $K_{\varphi_1 \wedge \varphi_2}$ simulates all states $K \models \varphi_1 \wedge \varphi_2$.
     (b) If $s_i \models \varphi_1 \wedge AX\varphi_2$ and then there exists $s'_i \in K_\varphi$ such that $s'_i \in K_{\varphi_1}$ and by induction hypothesis K_{φ_1} simulates all $K \models \varphi_1$. Moreover there exists an outgoing transition to an initial states of K_{φ_2} and by induction hypothesis K_{φ_2} simulates all states $K \models \varphi_2$

■

Bibliographie

[AdAdS+06] B.T. Adler, L. de Alfaro, L. Dias da Silva, M. Faella, A. Legay, V. Raman, and P. Roy. Ticc : A Tool for Interface Compatibility and Composition. In Thomas Ball and Robert B. Jones, editors, *CAV'06 : Proceedings of 18th International Conference of Computer Aided Verification*, volume 4144 of *Lecture Notes in Computer Science*, pages 59–62, Seattle, WA, USA, 2006.

[ADT+03] A. Aljer, P. Devienne, S. Tison, J-L. Boulanger, and G. Mariano. BHDL : Circuit Design in B. In IEEE Computer Society, editor, *ACSD '03 : Proceedings of the Third International Conference on Application of Concurrency to System Design*, page 241, Washington, DC, USA, 2003.

[AH99] R. Alur and T.A. Henzinger. Reactive modules. *Formal Methods in System Design*, 15(1) :7–48, 1999.

[AHK97] Rajeev Alur, Thomas A. Henzinger, and Orna Kupferman. Alternating-Time Temporal Logic. In Willem P. de Roever, Hans Langmaack, and Amir Pnueli, editors, *COMPOS*, volume 1536 of *Lecture Notes in Computer Science*, pages 23–60. Springer, 1997.

[AHM+98] R. Alur, T.A. Henzinger, F.Y. C. Mang, S. Qadeer, S.K. Rajamani, and S. Tasiran. MOCHA : Modularity in Model Checking. In *CAV'98 : Proceedings of the 10th International Conference on Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 521–525, London, UK, 1998.

[all] Alliance, a free VLSI cad system. `http://www-asim.lip6.fr/recherche/alliance/`.

[ARM99] ARM limited. *AMBA Specification*, 2.0 edition, 1999. ARM Limited ©. All right reserved.

[B-C97] B-Core(UK) Ltd. *B-Toolkit User's Manual*, release 3.7 edition, 1997.

[BA05] F. Badeau and A. Amelot. Using B as a High Level Programming Language in an Industrial Project : Roissy VAL. In *ZB 2005 : Formal Specification and Development in Z and B*, volume 3455/2005 of *Lecture Notes in Computer Science*, pages 334–354. Springer Berlin / Heidelberg, 2005.

[BBDEL96] I. Beer, S. Ben-David, C. Eisner, and A. Landver. RuleBase : an industry-oriented formal verification tool. In *DAC '96 : Proceedings of the 33rd annual conference on Design automation*, ACM Press, pages 655–660, New York, NY, USA, 1996.

[BBFM99] P. Behm, P. Benoit, A. Faivre, and J-M. Meynadier. Météor : A Successful Application of B in a Large Project. In Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, *World Congress on Formal Methods*, volume 1708 of *Lecture Notes in Computer Science*, pages 369–387. Springer, 1999.

[BCG88] M.C. Browne, E.M. Clarke, and O. Grumberg. Characterizing Finite Kripke Structures in Propositional Temporal Logic. *Theoritical Computer Science*, 59(1-2) :115–131, 1988.

[BCM+92] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking : 10^{20} States and Beyond. *Information and Computation*, 98(2) :142–170, 1992. Special issue for best papers from LICS'90.

[BG99] G. Bruns and P. Godefroid. Model Checking Partial State Spaces with 3-Valued Temporal Logics. In Nicolas Halbwachs and Doron Peled, editors, *CAV'99 : Proceedings of the 11th conference on Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 274–287. Springer, 1999.

[BG00] G. Bruns and P. Godefroid. Generalized Model Checking : Reasoning about Partial State Spaces. In Nicolas Halbwachs and Doron Peled, editors, *CONCUR'2000 : Proceedings of the 11th International Conference on Concurrency Theory*, volume 1877 of *Lecture Notes in Computer Science*, pages 168–182. Springer, 2000.

[BJP05] R. Bloem, B. Jobstmann, and A. Pnueli. Property-based Logic Synthesis for Rapid Design Prototyping. Technical report, Prosyd Deliverable D2.2/1, 2005.

[BR02] T. Ball and S.K. Rajamani. The SLAM project : debugging system software via static analysis. In *POPL '02 : Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–3, New York, NY, USA, 2002.

[Bra03] C. Braunstein. Transformation de propriétés CTL lors d'une méthode de conception incrémentale des wrappers VCI/PI. Rapport de DEA, LIP6/ASIM, 2003.

[Bry86] R.E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 35(8) :677–691, 1986.

[Büt05] W. Büttner. Is Formal Verification Bound to Remain a Junior Partner of Simulation ? In Wolfgang J. Paul Dominique Borrione, editor, *CHARME'2005 : Proceedings of the Correct Hardware Design and Verification Methods, 13th IFIP WG 10.5 Advanced Research Working Conference*, volume 3725 of *Lecture Notes in Computer Science*, page 1. Springer, 2005.

[Cad99] Cadence. *Formal Verification Using Affirma FormalCheck*, October 1999.

[CC77] P. Cousot and R. Cousot. Abstract interpretation : A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL'77 : Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, Munich, Germany, 1977.

[CDEG03] M. Chechik, B. Devereux, S. M. Easterbrook, and A. Gurfinkel. Multi-Valued Symbolic Model-Checking. *ACM Transactions on Software Engineering and Methodology*, 12(4) :371–408, 2003.

[CE81] E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In Dexter Kozen, editor, *Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 1981.

[CEPA+02] J-M. Couvreur, E. Encrenaz, E. Paviot-Adet, D. Poitrenaud, and P-A. Wacrenier. Data decision diagrams for petri net analysis. In Javier Esparza and Charles Lakos, editors, *ICATPN'02 : Proceedings of the 23rd International Conference on Applications and Theory of Petri Nets*, volume 2360 of *Lecture Notes in Computer Science*, pages 101–120. Springer, 2002.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2) :244–263, 1986.

[CGJ+03] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided bstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5) :752–794, 2003.

[CGL94] E.M. Clarke, O. Grumberg, and D.E. Long. Model Checking and Abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5) :1512–1542, 1994.

[CGP99] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[Cla03] E.M. Clarke. Counterexample-guided abstraction refinement. In *TIME*, page 7. IEEE Computer Society, 2003.

[CLM89] E. M. Clarke, D. E. Long, and K. L. McMillan. Compositional model checking. In *LICS'89 : Proceedings of the 4th Annual Symposium on Logic in Computer Science*, pages 353–362. IEEE Computer Society, 1989.

[CM01] D. Cansell and D. Méry. Abstraction and Refinement of Features. In S. Gilmore and M. Ryan, editors, *Language Constructs for Designing Features*, pages 65–84. Springer-Verlag, 2001.

[COJ+00] E.M. Clarke, O.Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In E.A. Emerson and A.P. Sistla, editors, *CAV'00 : Proceedings of the 12th International Conference on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2000.

[CRS01] F. Cassez, M. Ryan, and P-Y. Schobbens. Proving Feature Non-Interaction with Alternating-Time Temporal Logic. In S. Gilmore and M. Ryan, editors, *Language Constructs for Describing Features*, pages 85–104. Springer Verlag, 2001.

[CTM05] J-M. Couvreur and Y. Thierry-Mieg. Hierarchical Decision Diagrams to Exploit Model Structure. In Farn Wang, editor, *FORTE : Proceedings of the*

25th IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems, volume 3731 of *Lecture Notes in Computer Science*, pages 443–457, Taipei, Taiwan, 2005. Springer.

[DAC99] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in Property Specifications for Finite-State Verification. In ACM, editor, *ICSE' 99. Proceedings of the 1999 International Conference on Software Engineering*, pages 411–420, Los Angeles, CA, USA, 1999.

[dAdSF+05] Luca de Alfaro, Leandro Dias da Silva, Marco Faella, Axel Legay, Pritam Roy, and Maria Sorea. Sociable Interfaces. In Bernhard Gramlich, editor, *FroCos'05 : Proceedings of the 5th International Workshop on Frontiers of Combining Systems*, volume 3717 of *Lecture Notes in Computer Science*, pages 81–105, Vienna, Austria, 2005. Springer.

[dAH01] L. de Alfaro and T.A. Henzinger. Interface automata. In *ESEC/FSE-9 : Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 109–120, New York, NY, USA, 2001. ACM Press.

[DGG97] D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems*, 19(2) :253–291, 1997.

[DN05] D. Dams and K. S. Namjoshi. Automata as Abstractions. In Radhia Cousot, editor, *VMCAI'05 : Proceedings of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation*, volume 3385 of *Lecture Notes in Computer science*, pages 216–232, Paris, France, 2005. Springer.

[ECD+03] S.M. Easterbrook, M. Chechik, B. Devereux, A. Gurfinkel, A. Lai, V. Petrovykh, A. Tafliovich, and C. Thompson-Walsh. χChek : A Model Checker for Multi-Valued Reasoning. In *ICSE'03 : Proceedings of the 25th International Conference on Software Engineering*, pages 804–805. IEEE Computer Society, 2003.

[Fit91] M. Fitting. Bilattices and the Semantics of Logic Programming. *Journal of Logic Programming*, 11(1&2) :91–116, 1991.

[GC06] A. Gurfinkel and M. Chechik. Why Waste a Perfectly Good Abstraction ?. In Holger Hermanns and Jens Palsberg, editors, *TACAS'06 : Proceedings of the 12th international conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 3920 of *Lecture Notes in Computer Science*, pages 212–226, Vienna, Austria, 2006. Springer.

[GL91] O. Grumberg and D.E. Long. Model Checking and Modular Verification. In *International Conference on Concurrency Theory*, volume 527 of *Lecture Notes in Computer Science*, pages 250–263. Springer Verlag, 1991.

[GL00] A. Goel and W. R. Lee. Formal Verification of an IBM CoreConnect Processor Local Bus Arbiter Core. In ACM, editor, *DAC'00 : Proceedings of*

the 37th Conference on Design Automation, pages 196–200, Los Angeles, CA, USA, 2000.

[gro96] The VIS group. VIS : A System for Verification and Synthesis. In Rajeev Alur and Thomas A. Henzinger, editors, *CAV'96 : Proceedings of the 8th International Conference on Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 428–432, New Brunswick, NJ, USA, 1996. Springer-Verlag.

[gro05] Model Checking Systems group. *RuleBase Parallel Edition,User's Guide.* IBM Labs, Haifa, Israel, 2005. verion 1.26.

[Gru97] Orna Grumberg, editor. *Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings*, volume 1254 of *Lecture Notes in Computer Science*. Springer, 1997.

[Gru05] Orna Grumberg. Abstraction and Refinement in Model Checking. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *FMCO'05 : 4th International Symposium on Formal Methods for Components and Objects,*, volume 4111 of *Lecture Notes in Computer Science*, pages 219–242. Springer, 2005.

[GS97] S. Graf and H. Saïdi. Construction of Abstract State Graphs with PVS. In Grumberg [Gru97], pages 72–83.

[Gup92] A. Gupta. Formal Hardware Verification Methods : A Survey. *Formal Methods in System Design*, 1(2/3) :151–238, 1992.

[GWC06] A. Gurfinkel, O. Wei, and M. Chechik. Systematic Construction of Abstractions for Model-Checking. In E. Allen Emerson and Kedar S. Namjoshi, editors, *VMCAI'06 : Proceedings of the 7th International Conference on Verification, Model Checking, and Abstract Interpretation*, volume 3855 of *Lecture Notes in Computer Science*, pages 381–397, Charleston, USA, 2006. Springer.

[HJMS03] T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software Verification with BLAST. In *SPIN'2003 : Proceedings of the 10th International SPIN Workshop*, volume 2648 of *Lecture Notes in Computer Science*, pages 235–239, USA, 2003. Springer. Tool paper.

[HLQR99] T. A. Henzinger, X. Liu, S. Qadeer, and S. K. Rajamani. Formal Specification and Verification of a Dataflow Processor Array. In Ellen Sentovich Jacob K. White, editor, *ICCAD'99 : Proceedings of the 1999 IEEE/ACM International Conference on Computer-Aided Design*, pages 494–499, San Jose, USA, 1999.

[Hol97] G.J. Holzmann. The Model Checker Spin. *IEEE Transactions on Software Engineering*, 23(5) :279–295, May 1997. Special issue on Formal Methods in Software Practice.

[HQR98] T.A. Henzinger, S. Qadeer, and S.K. Rajamani. You Assume, We Guarantee : Methodology and Case Studies. In Moshe Y. Vardi Alan J. Hu, editor,

CAV'98 : Proceedings of the 10th International Conference on Computer Aided Verification, volume 1427 of *Lecture Notes in Computer Science*, pages 440–451, Vancouver, Canada, 1998. Springer-Verlag.

[HQRT02] T.A. Henzinger, S. Qadeer, S.K. Rajamani, and S. Tasiran. An Assume-Guarantee Rule for Checking Simulation. *ACM Transactions on Programming Languages Systems*, 24(1) :51–64, 2002.

[JKSC05] H. Jain, D. Kroening, N. Sharygina, and E.M. Clarke. Word Level Predicate Abstraction and Refinement for Verifying RTL Verilog. In Andrew B. Kahng William H. Joyner Jr., Grant Martin, editor, *DAC'05 : Proceedings of the 42nd Design Automation Conference*, ACM, pages 445–450, San Diego, USA, 2005. ACM.

[JKSC07] H. Jain, D. Kroening, N. Sharygina, and E. Clarke. VCEGAR : Verilog CounterExample Guided Abstraction Refinement. In *TACAS'07 : Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2007. Accepted paper.

[JM01] R. Jhala and K.L. McMillan. Microarchitecture Verification by Compositional Model Checking. In Alain Finkel Gérard Berry, Hubert Comon, editor, *CAV'01 : Proceedings of the 13th International Conference on Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, pages 396–410, Paris, France, 2001. Springer.

[KG99] C. Kern and M.R. Greenstreet. Formal Verification in Hardware Design : a Survey. *ACM Transactions on Design Automation of Electronic Systems*, 4(2) :123–193, 1999.

[Kro99] T. Kropf. *Introduction to Formal Hardware Verification*. Springer Verlag, Secaucus, NJ, USA, 1999.

[KVW00] O. Kupferman, M.Y. Vardi, and P. Wolper. An Automata-Theoretic Approach to Branching-Time Model Checking. *Journal of the ACM*, 47(2) :312–360, 2000.

[Lan96] K. Lano. *The B Language and Method, A Guide to Practical Formal Development*. FACIT. Springer-Verlag, UK, 1996.

[LGS+95] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property Preserving Abstractions for the Verification of Concurrent Systems. *Formal Methods in System Design*, 6(1) :11–44, 1995.

[LN91] B. Lin and A.R. Newton. Implicit manipulation of equivalence classes using binary decision diagrams. In *ICCD '91 : Proceedings of the 1991 IEEE International Conference on Computer Design on VLSI in Computer & Processors*, pages 81–85, Washington, DC, USA, 1991. IEEE Computer Society.

[Lon93] D. E. Long. *Model Checking, Abstraction, and Compositional Verification*. PhD thesis, Carnegie Mellon University, 1993.

[MAB06] K. Morin-Allory and D. Borrione. Proven Correct Monitors from PSL Specifications. In Georges G. E. Gielen, editor, *DATE'06 : Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1246–1251,

Munich, Germany, 2006. European Design and Automation Association, European Design and Automation Association, Leuven, Belgium.

[MBA02] G. Mariano, J-L. Boulanger, and A. Aljer. Formalization of Digital Circuits Using the B Method. In *CompRail VIII : 8th International Conference on Computer Aided Design, Manufacture and Operation in the Railway and other Advanced Mass Transit Systems*, Lemnos, Greece, 12 - 14 June 2002.

[McM93] K.L. McMillan. *Symbolic Model Checking.* Kluwer Academics Publishers, 1993.

[McM97] Kenneth L. McMillan. A Compositional Rule for Hardware Design Refinement. In Grumberg [Gru97], pages 24–35.

[McM98] K.L. McMillan. Verification of an Implementation of Tomasulo's Algorithm by Compositional Model Checking. In Moshe Y. Vardi Alan J. Hu, editor, *CAV'98 : Proceddings of the 10th International Conference on Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 110–121, Vancouver, Canada, 1998. Springer.

[McM00] K.L. McMillan. A Methodology for Hardware Verification Using Compositional Model Checking. *Science of Computer Programming*, 37(1–3) :279–309, 2000.

[Mil71] R. Milner. An Algebraic Definition of Simulation Between Programs. In *IJCAI'71 : Proceedings of the 2nd International Joint Conference on Artificial Intelligence*, pages 481–489, 1971.

[NV95] R. De Nicola and F.W. Vaandrager. Three logics for branching bisimulation. *Journal of the ACM*, 42(2) :458–487, 1995.

[On-00] On-Chip Bus Development Working Group. *Virtual Component Interface Standard (VCI).* VSI Alliance, 2000.

[one] Onespin solutions. `http://www.onespin-solutions.com/`.

[Ope94] Open Microprocessors System Initiatives. *OMI324 : PI-Bus Standard Specification.* Siemens, Munich, Germany, 1994.

[Oss02] J. Ossima Khéba. Les Wrappers PI/VCI et Etude de Faisabilité des Wrappers AMBA/VCI. Rapport de stage, LIP6 dept. ASIM, 2002.

[Oss03] J. Ossima Khéba. Spécification des Wrappers AHB/VCI. Rapport de DEA, LIP6 dept. ASIM, 2003.

[Par81] D. Park. Concurrency and Automata on Infinite Sequences. In Peter Deussen, editor, *Proceedings of the 5th GI-Conference on Theoretical Computer Science*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer, 1981.

[PdAHSV02] R. Passerone, L. de Alfaro, T.A. Henzinger, and A.L. Sangiovanni-Vincentelli. Convertibility Verification and Converter Synthesis : Two Faces of The Same Coin. In *ICCAD '02 : Proceedings of the 2002 IEEE/ACM International Conference on Computer-Aided Design*, pages 132–139, New York, USA, 2002. ACM Press.

[Pét03] F. Pétrot. Intégration des systèmes matériel/logiciel. Habilitation à diriger des recherches, Universitée Pierre et Marie Curie, 2003.

[Pev95] V. Pevtschin. The Open Microprocessor Systems Initiative : A Strategy Towards Integrated System Design. In *RSP'95 : Proceddings of the 6th IEEE International Workshop on Rapid System Prototyping*, pages 2–11, 1995.

[PH04] D.A. Patterson and J.L Hennessy. *Computer Organization and Design : The Hardware/Software Interface, Third Edition (The Morgan Kaufmann Series in Computer Architecture and Design) (The ... Series in Computer Architecture and Design).* Morgan Kaufmann, third edition, August 2004.

[PMT02] H. Peng, Y. Mokhtari, and S. Tahar. Environment Synthesis for Compositional Model Checking. In *ICCD'02 : Proceedings of the 20th International Conference on Computer Design*, pages 70–, Freiburg, Germany, 2002. IEEE Computer Society.

[PR99] M.C. Plath and M.D. Ryan. SFI : a Feature Integration Tool. In R. Berghammer and Y. Lakhnech, editors, *Tool Support for System Specification, Development and Verification*, Advances in Computing Science, pages 201–216. Springer, 1999.

[PR01] M. Plath and M. Ryan. Feature Integration using a Feature Construct. *Science of Computer Programming*, 41(1) :53–84, 2001.

[PRSV98] R. Passerone, J.A. Rowson, and A.L. Sangiovanni-Vincentelli. Automatic Synthesis of Interfaces Between Incompatible Protocols. In *DAC'98 : Proceedings of the 35th Conference on Design Automation*, pages 8–13, 1998.

[PTK03] H. Peng, S. Tahar, and F. Khendek. Comparison of SPIN and VIS for Protocol Verification. *International Journal on Software Tools for Technology Transfer (STTT)*, 4(2) :234–245, 2003.

[SG04] S. Shoham and O. Grumberg. Monotonic Abstraction-Refinement for CTL. In Andreas Podelski Kurt Jensen, editor, *TACAS'04 : Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *Lecture Notes in Computer Science*, pages 546–560, Barcelona, Spain, 2004. Springer.

[SNBE06] M. Schickel, V. Nimbler, M. Braun, and H. Eveking. On Consistency and Completness of Property-Sets : Exploiting the Property-Based Design Process. In *FDL'06 : Proceeding of Forum on specification and Design Languages*, 2006.

[soc] The SoCLib Project. `http://soclib.lip6.fr/`.

[Spa01] J. Sparsø. Asynchronous Circuit Design - A Tutorial. In *Chapters 1-8 in "Principles of asynchronous circuit design - A systems Perspective"*, pages 1–152. Kluwer Academic Publishers, London, dec 2001.

[STE98] STERIA Technologie de l'information. *Atelier B, Manuel Utilisateur.* Aix-en-Provence, France, 1998.

[tex] Texas-97 Verification Benchmarks. http://www.cad.eecs.berkeley.edu/Respep/Research/vis/texas-97/.

[Uni] Université Pierre et Marie Curie (UPMC), Paris 6. UE : Achitecture des systèmes intégrés, M1. http://www-master.ufr-info-p6.jussieu.fr/ue/archi/.

[Upt92] D.M. Upton. A Flexible Structure for Computer-Controlled Manufacturing Systems. *Manufacturing Review*, 5(1) :58–74, 1992.

[vsi] VSI Alliance. http://www.vsi.org/.

[XB03] F. Xie and J.C. Browne. Verified Systems by Composition from Verified Components. In *ESEC/FSE 2003 : Proceedings of the 11th ACM SIGSOFT Symposium on Foundations of Software Engineering 2003 held jointly with 9th European Software Engineering Conference*, pages 277–286, Helsinki, Finland, 2003. ACM Press.