2m11.2973.7

Université de Montréal

Synthèse de EFSM observatrices à partir de spécifications HAAD

par

Etienne Ogoubi

Département d'informatique et de recherche opérationnelle

Faculté des arts et des sciences

Mémoire présenté à la Faculté des Études Supérieures
en vue de l'obtention du grade de maîtrise (M.Sc)
en informatique

Avril, 2002

© Etienne Ogoubi, 2002

QA
76
U34
2002
V.038

Université de Montréal
Faculté des Études Supérieures

Ce mémoire intitulé

Synthèse de EFSM observatrices à partir de spécifications HAAD

présenté par

Etienne Ogoubi

a été évalué par un jury composé des personnes suivantes:

Michel Boyer	président-rapporteur
Eduard Cerny	directeur de recherche
Guy Bois	membre du jury

Mémoire accepté le 23 août 2002

Sommaire

En informatique, le processus de vérification de systèmes numériques constitue un grand défi pour les chercheurs. Plusieurs études proposent un nombre très varié de solutions et de méthodes de vérification. Nos travaux portent sur une méthode qui permet de vérifier la conception et l'implementation des protocoles des bus synchones (exemple le PCI). Pour ce faire, nous convertissons les chronogrammes de spécifications des protocoles de bus en Diagrammes à Actions Annotés et Hiérarchiques (HAAD), puis en machines à états observatrices, qui sont utilisées pour observer les activités sur le bus en simulation, en émulation et en méthodes formelles. La machine observatrice signale une erreur si le protocole du bus est violé. Les machines observatrices sont codées en RTL synthétisable dans les languages de description Verilog ou VHDL. Pour illustrer notre méthode, nous utilisons un sous-ensemble du protocole du bus PCI.

***Mots Clés**: Diagrammes à Actions Annotés et Hiérarchique,Diagramme à Action Feuille, Diagramme à Action Hiérarchique, Machine à États Finis, Machine à États Observatrice, Machine à États Finis Synchrone et Temporisée (STFSM), Peripheral Component Interconnect (PCI), Protocole du Bus PCI.*

Abstract

Efficient design verification is a major preoccupation in hardware systems design. We report on a method that assists the verification of the implementations of synchronous bus protocols (e.g., PCI). To this end we convert a Timing Diagram specification of the protocol in the form of Hierarchical Annotated Action Diagrams (HAAD) into synchronous state machines - checkers that can be used to observe the activity on the bus during simulation, emulation or formal verification and signal an error signal if a protocol violation is detected. The checkers are coded in synthesizable RT-level Verilog or VHDL. We illustrate the method on a subset of the PCI bus protocol.

Key Words: *Hierarchical Annotated Action Diagram (HAAD), Leaf Action Diagram (LAD), Hierarchical Action Diagram (HAD), Finite State Machines (FSM), Checker State Machine, Synchronous Timed Finite State Machines (STFSM), Peripheral Component Interconnect (PCI), PCI-Bus Protocol.*

Table Des Matières

Pages

Sommaire iii

Abstract iv

Table des matières v

Liste des figures ix

Liste des signes et des abbréviations xiii

Chapitre 1 : Introduction 1

1.1. La vérification des systèmes numériques 2

1.1.1. La simulation 2

1.1.2. Les méthodes formelles 3

1.1.3. Les méthodes symboliques 3

1.1.4. L'émulation 4

1.2. Proposition de nouvelles solutions pour la vérification de bus 5

1.3. Plan du mémoire 12

Chapitre 2 : Spécification d'un système par le HAAD 14

2.1. Introduction 14

2.2. Diagrammes à actions annotés et hiérarchique (HAAD) 16

2.2.1. Diagrammes à actions feuille 16

2.2.2. Annotation des diagrammes feuilles 22

2.2.3. Diagrammes à actions hiérarchiques 23

2.2.4. Annotation des diagrammes hiérarchiques 25

2.2.5. Description d'un diagramme à actions feuille en langage HAAD *26*

2.2.6. Diagrammes synchrones *28*

2.3. Conclusion 29

Chapitre 3 : Construction des machines observatrices à partir des spécifications HAAD 31

3.1. Introduction 31

3.2. Décomposition des diagrammes feuilles 32

3.2.1. La décomposition en parallèle d'un diagramme feuille *33*

3.2.2. La décomposition en cascade d'un diagramme feuille *35*

3.3. Machine à états finis synchrone et temporisée (STFSM) 40

3.4. La machine STFSM générique 46

3.5. Composition des STFSM en fonction des HAD 50

3.5.1. Les équations de coordination des STFSM *50*

3.6. Conclusion 55

Chapitre 4 : Validité de la méthode pour la synthèse des EFSM 56

4.1. Introduction 56

4.2. Démonstration de modèle 56

4.2.1. Démonstration de modèle *58*

4.2.2. Computation Tree Logic (CTL) *58*

4.3. Validité des STFSM génériques 59

4.3.1. Vérification du modèle STFSM générique *59*

4.3.2. Prototype de test pour le modèle générique du STFSM en verilog *62*

4.3.3. Spécification et vérification des propriétés pour le modèle générique du STFSM *64*

4.3.4. Vérification des compositions des STFSM *64*

4.4. Conclusion 66

Chapitre 5 : Étude du bus PCI: synthèse de EFSM observatrice et les résultats expérimentaux 67

5.1. Introduction 67

5.2. Opérations sur le bus PCI 68

5.3. Les groupes des signaux fonctionnels du bus PCI 69

5.3.1. Le signal d'horloge (CLK) *71*

5.3.2. Les signaux de contrôle *72*

5.4. Comportement des signaux de contrôle sur le PCI lors de la transaction de lecture 73

5.5. Comportement des signaux de contrôle sur le PCI lors de la transaction d'écriture 76

5.6. Interruptions prématurées des transactions sur le bus PCI 78

5.6.1. Les raisons pour lesquelles la source peut interrompre une transaction *78*

5.6.2. Les raisons pour lesquelles la destination peut interrompre une transaction *80*

5.7. Les contraintes temporelles lors d'une transaction sur le bus PCI 82

5.7.1. Prévenir la monopolisation du bus par la source *83*

5.7.2. Prévenir la monopolisation du bus par la destination *83*

5.8. Modèle HAAD synchrone des transactions de lecture et d'écriture 84

5.8.1. Traduction de la transaction de lecture en diagrammes feuilles transformables en STFSM *86*

5.8.2. Traduction de la transaction d'écriture en diagrammes feuilles transformables en STFSM *95*

5.9. Conclusion 101

Chapitre 6 : Conclusion - synthèse des EFSM observatrices à partir des HAAD 102

6.1. Travaux accomplis 103

6.2. Travaux à venir 104

Références..106

Annexe...xiv

Liste des Figures

Pages

FIGURE 1. Machine Observatrice sur un bus lors de la vérification d'une transaction.......7

FIGURE 2. Résumé de l'approche utilisée pour la synthèse de la machine observatrice.......9

FIGURE 3. Diagramme à actions feuille représentant un fragment de la spécification de la fonction de lecture sur le PCI.......17

FIGURE 4. Diagramme à actions feuille avec des opérations.......19

FIGURE 5. Représentation graphique d'un diagramme à action feuille mettant en évidence les différents types de contraintes.......22

FIGURE 6. Spécification en langage HAAD du diagramme à actions feuille de la Figure 5.......27

FIGURE 7. Exemple de diagramme synchrone: Transaction réduite de lecture sur le bus PCI.......29

FIGURE 8. Transaction réduite de lecture sur le bus PCI représentant un diagramme feuille f.......33

FIGURE 9. Exemple de diagramme feuille résultant de l'application de la règle 1 de la décomposition parallèle sur le diagramme f.......34

FIGURE 10. Décomposition en cascade du diagramme feuille représenté par la figure 9.......35

FIGURE 11. Exemple de diagramme feuille résultant de l'application de la règle 3. i) de la décomposition cascade sur le diagramme feuille de la 9.......36

FIGURE 12. Composition parallèle..37
FIGURE 13. Composition en cascade...38
FIGURE 14. Modèle HAAD du fragment Frame_Irdy de la transaction de lecture du bus PCI. Recomposition des fragments pour donner fonctionnellement Frame_Irdy..39
FIGURE 15. Exemple de STFSM: boîte Attente_et_Fin_de_Transaction de la Figure 10...46
FIGURE 16. Modèle générique de STFSM à plusieurs cycles d'horloge...............48
FIGURE 17. Modèle générique de STFSM à 1 cycle d'horloge.............................48
FIGURE 18. Description schématique de la composition en cascade de 2 machines...51
FIGURE 19. Opération de boucle sur un STFSM à 1 cycle d'horloge.....................54
FIGURE 20. Machine à états pour générer le signal go..60
FIGURE 21. Machine à états réflecteur du signal ack_in...60
FIGURE 22. Schéma de prototype de test pour le modèle générique du STFSM.....63
FIGURE 23. Représentation schématique d'un agent source du bus PCI................70
FIGURE 24. Représentation schématique d'un agent destination du bus PCI..........71
FIGURE 25. Chronogramme de la transaction de lecture sur le bus PCI.................75
FIGURE 26. Chronogramme de la transaction d'écriture sur le bus PCI.................78
FIGURE 27. Modèle HAAD des transactions de lecture et d'écriture sur le bus PCI...84
FIGURE 28. Diagramme à action de la transaction de lecture sur le bus PCI...........85
FIGURE 29. Diagramme à action de la transaction d'écriture sur le bus PCI...........85
FIGURE 30. Diagramme feuille Frame_Irdy issu de la décomposition parallèle de la figure 28..86
FIGURE 31. Diagramme feuille Frame_Trdy issu de la décomposition parallèle de la figure 28..87

FIGURE 32. Diagramme feuille Frame_Devsel issu de la décomposition parallèle de la figure 28 87
FIGURE 33. Composition parallèle de diagrammes feuilles réduits issus de la décomposition de la figure 28 88
FIGURE 34. Modèle HAAD du fragment Frame_Irdy de la transaction de lecture du bus PCI 89
FIGURE 35. Modèle HAAD du fragment Frame_Trdy de la transaction de lecture du bus PCI 90
FIGURE 36. Modèle HAAD du fragment Frame_Devsel de la transaction de lecture du bus PCI 91
FIGURE 37. STFSM de "Étendue_de_Frame" montré dans la figure 34 92
FIGURE 38. STFSM "Active" montré dans la figure 34 92
FIGURE 39. STFSM de "Début_de_Transaction" montré dans la figure 34 93
FIGURE 40. STFSM de "État_Attente" montré dans la figure 34 93
FIGURE 41. STFSM de "Fin_de_Transaction" montré dans la figure 34 94
FIGURE 42. STFSM de "Attente_et_Fin_de_Transaction" montré dans la figure 34 94
FIGURE 43. STFSM de "Début_et_Fin_de_Transaction" montré dans la figure 34 95
FIGURE 44. Diagramme feuille Frame_Irdy issu de la décomposition parallèle de la figure 29 96
FIGURE 45. Diagramme feuille Frame_Trdy issu de la décomposition parallèle de la figure 29 96
FIGURE 46. Diagramme feuille Frame_Devsel issu de la décomposition parallèle de la figure 29 97
FIGURE 47. Composition parallèle de diagrammes feuilles réduits issus de la décomposition de la figure 25 97

FIGURE 48. Modèle HAAD du fragment Frame_Irdy de la transaction d'écriture du bus PCI....98
FIGURE 49. Modèle HAAD du fragment Frame_Trdy de la transaction d'écriture du bus PCI....99
FIGURE 50. Modèle HAAD du fragment Frame_Devsel de la transaction d'écriture du bus PCI....100

Liste des signes et abbréviations

EFSM	*Extended Finite State Machine*
HAAD	*Hierarchical Annoted Actions Diagram*
HAD	*Hierarchical Actions Diagram*
IP	*Intellectual Property*
LAD	*Leaf Actions Diagram*
PCI	*Peripheral Component Interconnect*
RTL	*Register-Transfer Level*
STFSM	*Synchronous Timed Finite State Machine*
TD	*Timing Diagram*

Remerciements

Je rends grâce à Dieu en notre Seigneur Jesus Christ pour toutes ces années.

Il y a un temps pour toute chose. Un temps pour commencer, et un temps pour finir ce qu'on a commencé. D'après la citation de Hudson Taylors, il y a trois étapes dans le cycle de vie de toute tâche: l'imposibilité, la difficulté et l'accomplissement.

Pour ma part, ce travail aurait été impossible sans l'aide de personnes spéciales. Je pense particulièrement au professeur Eduard Cerny, mon directeur. Je tiens à le remercier sincèrement pour m'avoir donné ce sujet, et avoir toujours été là pour moi. Chaque fois que j'ai franchi le seuil de son bureau, il a toujours été disponible pour aider, conseiller, soutenir, encourager et lorsque c'était nécessaire, reprimander et remettre les choses sur les rails. Je tiens à le remercier davantage pour le généreux soutien financié qu'il m'a accordé. Ce soutien est d'autant plus apprécié que cela est arrivé à un moment où j'étais prêt à tout laisser tomber par manque de moyens financiers.

Le professeur Eduard Cerny est tout simplement le meilleur directeur qu'un étudiant puisse souhaiter avoir.

Je remercie ma très chère Lady pour son aide. Son soutien de tous les jours est apprécié.

Je remercie mon ami Moustapha Azizi, le temoin attentif de mes nuits blanches au laboratoire, le soutien infaillible de la vérification de la méthode que nous avons proposée, et de la préparation de mes conférences.

Je remercie aussi mon ami Ismaël Ben Barka et ses parents qui m'ont soutenu sur beaucoup de plans, pour les bons moments et les privations que nous avons partagés au 1160 rue St. Mathieu à Montréal.

La rédaction du mémoire a été difficile lorsque je suis allé à Nortel Networks pour le projet OPC. Le travail à Nortel est tellement absorbant et épuisant que je me suis demandé si je finirais un jour mon mémoire. À Nortel, j'ai eu le soutien et les encouragements de plusieurs personnes comme Bogdan Boruslawski, Michel Langevin, David Fisher, David Carpenter, Paul Welton, et bien d'autres.

Finalement, l'accomplissement de ce mémoire est une reconnaissance à mes parents, surtout à mon père, Emile Koffi Ogoubi, pour qui j'ai beaucoup d'affection et de qui j'ai beaucoup reçu.

CHAPITRE 1

Introduction

The work actually began in 1969 when Intel agreed to build some chips for a Japanese calculator company. In the course of my work at Intel, I had occasion to look at the design for those chips and I felt it could be improved. One of the ways to improve it was to replace the chip family with a single chip general purpose computer and memory to contain programmes that would then programme that computer to perform the calculator functions. That basically is how the first microprocessor started.

T. Hoff, Monaco, 30-10-1996

De nos jours, le développement des systèmes électroniques a comme principal défi la vérification fonctionnelle. La complexité des systèmes microélectroniques et l'augmentation considérable du nombre de portes logiques dans les puces ont entrainé *de facto* une augmentation des efforts de vérification; 60% à 80% des efforts économiques et temporels dans le cycle de développement d'un système sont consacrés à la vérification. Pour réduire ces efforts de développement, plusieurs méthodes ont été créées, d'abord lors de la synthèse des systèmes et ensuite lors de leur vérification. Sur le plan de la synthèse, nous remarquons la construction de blocs fonctionnels spécifiques (Intellectual Property (IP) blocs), réutilisables dans la composition de systèmes plus complexes. Pour lier ces blocs fonctionnels, il est indispensable de définir des protocoles de communication appropriés pour le bus qui joue le rôle de liaison, et de vérifier ensuite si les blocs respectent les protocoles spécifiés pour ce bus. L'objectif ici n'est pas la vérification des composants qui participent à la communication, mais celle de leur comportement sur le bus qui les lie.

Sur le plan de la vérification, plusieurs méthodes utilisent de nouvelles approches de simulation et des méthodes formelles en combinant la simulation, l'émulation et les méthodes formelles pour s'assurer que le comportement des IPs connectés aux bus respectent leurs spécifications.

1.1. La vérification des systèmes numériques

La vérification est considérée comme l'ensemble des moyens mis en oeuvre pour s'assurer qu'un système numérique est conforme à sa spécification, tant aux points de vue qualitatif (l'ordre des événements), quantitatif (conservation de l'information traitée), que temporel (le comportement par rapport au temps). Nous distinguons trois grandes classes de méthodes de vérification: la *simulation*, les *méthodes formelles* et *symboliques,* et l'*émulation.*

1.1.1. La simulation

Par définition, la simulation est soit une reproduction expérimentale des conditions réelles dans lesquelles une opération complexe s'est produite, soit une représentation d'un système par un modèle analogue plus facile à étudier.

Techniquement, la méthodologie de la simulation consiste à soumettre à un système numérique réactif un ensemble de données appelé *stimuli* en entrée et à analyser les résultats de la réaction de ce système en sortie. Il faut ensuite déterminer si le comportement du système par rapport aux *stimuli* respecte ses spécifications.

Pour être efficace, le principe de la simulation doit tenir compte de toutes les combinaisons possibles des *stimuli*. Cependant, un problème se pose lorsqu'il s'agit de vérifier des systèmes numériques plus ou moins complexes. Le nombre de configurations *(entrée, sortie)* résultant de la combinaison des données des *stimuli* devient exponentiellement élevé. La simulation exhaustive se révèle alors trop longue et très coûteuse.

Une solution au problème de la simulation est l'utilisation de méthodes formelles qui sont considérées comme plus rigoureuses.

1.1.2. Les méthodes formelles [20]

Les méthodes formelles dans la conception des systèmes numériques représentent un ensemble de principes basé sur des approches mathématiques qui permettent d'établir formellement qu'une implémentation satisfait à sa spécification. Depuis l'avènement des méthodes formelles, des moyens comme la "*preuve de théorèmes*" et surtout la "*vérification d'équivalence*" et la "*vérification de modèle*" ont émergé comme des solutions pratiques pour certains aspects des problèmes de la vérification. Ces moyens peuvent être appliqués a posteriori (test, preuve, vérification et mesures) ou en cours de conception (validation de spécification, création de maquettes, dérivation et synthèse, prédiction des performances). Les références [10], [27], [18], [55], [22] et [45] sont des exemples d'études et d'applications des méthodes formelles.

Plusieurs travaux ont été réalisés afin de répondre aux questions sur le niveau d'abstraction de la description et sur les méthodes alternatives pour la vérification des propriétés des processeurs [22], [45], [53], [47]. D'autres méthodes ont été développées [55], [37] pour faciliter les spécifications et la vérification des systèmes numériques utilisant des chronogrammes (Timing Diagrams).

Bien que les méthodes formelles aient apporté une amélioration à la vérification de systèmes numériques par rapport à la simulation, cette amélioration reste néanmoins insuffisante.

1.1.3. Les méthodes symboliques [33]

Les méthodes symboliques ont émergé comme solutions alternatives à la simulation et aux méthodes formelles. Elles consistent en une représentation compacte et symbolique des données à propager, alors que la simulation utilise la propagation des valeurs de ces données à travers le système. Les configurations sont alors basées sur

des ensembles de valeurs qui regroupent des données relatives. Dans ce cas, le nombre de configuration diminue considérablement. Les mêmes approches sont utilisées en méthodes formelles symboliques. Par exemple, Berkane et Cerny [44] pour explorer l'espace des états d'un automate temporisé, raisonnent en terme de zones temporelles relatives, ce qui consiste en un regroupement des états de l'automate dans des ensembles. Dans le même ordre d'idée, Bryant présente une nouvelle méthode dite "*simulation symbolique*" dans la référence [26], qui consiste à utiliser des valeurs symboliques spéciales pour encoder un ensemble de conditions d'opération d'un circuit éléctronique. Cette méthode peut-être appliquée à la fois à la vérification logique et temporelle. Par ailleurs, Seger et Bryant présentent dans la référence [50] une version modifiée de la simulation symbolique appelée "*évaluation symbolique de trajectoire*" (STE - Symbolic Trajectory Evaluation), dans laquelle les propriétés d'un système sont exprimées dans une forme restrainte de la logique temporelle. L'expression de la logique utilisée est limitée à la description des propriétés temporelles de circuits et à celle de leur transition des états. L'algorithme de décision utilisé par l'évaluation symbolique de trajectoire teste la validité d'une assertion en déterminant si oui ou non tout comportement causé par une séquence d'états qui satisfait un antécédent est un comportement qui satisfait son conséquent. Plusieurs autres travaux sur les méthodes symboliques ont été développés, comme les algorithmes itératifs proposés par Wong-Toi et Dill [41] qui opèrent sur des approximations de l'ensemble des propriétés temporelles du système.

1.1.4. L'émulation

Étymologiquement, l'émulation, du latin *æmulatus*, veut dire imiter ou exceller. L'émulation des systèmes numériques est considérée aujourd'hui comme la solution là où la simulation et les méthodes formelles et symboliques ont échoué. Elle consiste en l'interconnection d'un ensemble de puces électroniques reprogrammables, comme les FPGA (Field-Programmable Gate Arrays), et d'un ensemble de logiciels de synthèse et d'analyse, qui permettent de créer rapidement des représentations de systèmes

numériques ou des ASIC (Application Specific Integreated Circuits) au niveau matériel. Cette technique peut être utilisée très tôt pour vérifier exactement le système lors de sa conception. Elle permet aussi de détecter rapidement et exactement des erreurs à des niveaux d'abstraction très élevés ou à des niveaux portes logiques dans des circuits de plusieurs centaines de milliers, voire de millions de portes. Beaucoup de techniques dans le domaine de l'émulation logique ont été développées comme le SFE (Serial Fault Emulation) de la compagnie Meta Systems. Le SFE consiste à emuler séparément chaque circuit erronné du niveau "blocs fonctionnels" jusqu'au niveau "signaux" afin de déterminer des fautes ou des bloquages de signaux dans un état quelconque.

Bien que l'émulation soit vue comme solution d'avenir pour la vérification de systèmes numériques, plusieurs problèmes se posent toujours et les travaux pour réduire les efforts de vérification ne cessent de se diversifier.

Dans nos travaux, nous nous sommes intéressés à la vérification de bus, comme le PCI, qui permettent de lier plusieurs composants d'un système numérique.

1.2. Proposition d'une solution pour la vérification de bus

Dans le cas de la validation du protocole d'un bus, une des façons les plus simples de vérifier si les éléments numériques liés par ce bus respectent les spécifications de ce protocole en utilisant la simulation, l'émulation ou les méthodes formelles, est de connecter un observateur du protocole au bus. Cet observateur observe les activités sur le bus et signale toute violation du protocole de ce dernier. Le but de notre travail est de construire ce modèle observateur pour un bus synchrone à partir de sa spécification sous forme de Diagramme d'Actions Annotés Hiérarchiques ou HAAD [59].

Dans la référence [59], les auteurs présentent la méthode HAAD comme des scénarios de comportements finis ou de diagrammes qui, étant donnée une encapsulation appro-

priée, peuvent être composés pour former une description hiérarchique d'un comportement fini ou infini (cyclique). Par ailleurs, la méthode HAAD permet de généraliser la notion de chronogramme en incluant les signaux ou les ports de tous genres. L'attribution d'une valeur à un port est appelée ici une action. Une action peut représenter un changement de valeur sur un port (une transition ou un événement) ou peut être une mise à jour d'un signal sur un port sans nécessairement changer sa valeur. Les chronogrammes sont appelés dans la méthode HAAD, Diagramme à Action Feuille et Hiérarchique. Ces Diagrammes peuvent être annotés par des variables, des appels de procédures, etc., d'où vient le nom de Diagrammes d'Actions Annotés Hiérarchiques (Hiérarchical Annotated Action Diagrams (HAAD)).

Le modèle observateur synthétisé à partir des spécifications HAAD est implémenté sous la forme de machine à états finis étendue (Extended Finite State Machine (EFSM)) qui est décrite en langages HDL Verilog (ou VHDL) synthétisable et synchrone [60] [17] [52].

Lorsqu'elle est utilisée en émulation, la machine observatrice est instanciée avec le modèle du bus et connectée à ses ports d'entrée et de sortie, en les observant seulement. L'observateur génère un signal booléen qui indique si le comportement qui a été observé sur les ports du bus est conforme ou non à la spécification HAAD de ce bus. Figure 1.

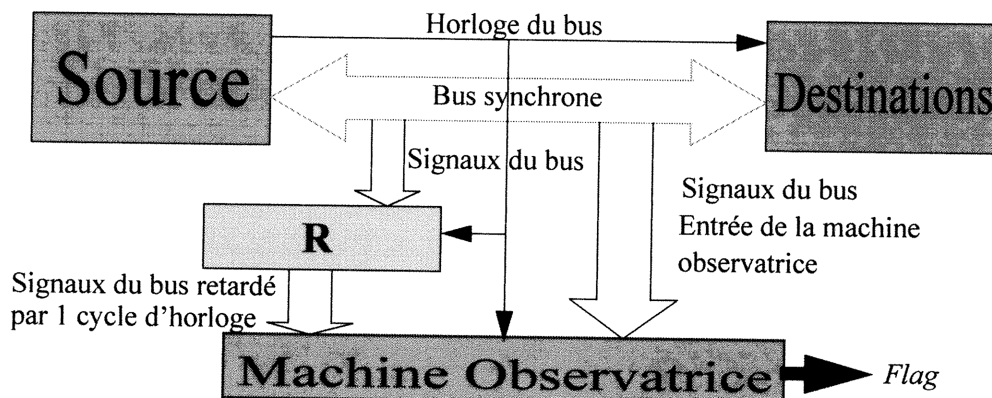


FIGURE 1. Machine Observatrice sur un bus lors de la vérification d'une transaction

La figure 1 est une représentation schématique d'une machine observatrice instanciée sur un bus afin qu'elle observe son comportement. L'initiateur de la transaction, représenté par la boîte *source* et les *destinations,* sont synchrones à l'horloge du bus. Toute transaction sur le bus synchrone est observée par la machine observatrice, comparée avec la transaction précédente (cycle d'horloge précédente) et enregistrée dans le registre R pour déterminer si la progression des événements est conforme à la spécification. Sur la figure 1, la machine observatrice est connectée au bus par l'ensemble des signaux que nous avons appelé "*Signaux du bus, entrée de la machine observatrice*". De même, le registre R est connecté à un ensemble de signaux appelé "*Signaux du bus, entrée du registre R*".

Dans la référence [37], les auteurs indiquent qu'ils peuvent produire des spécifications en logique temporelle à partir de leur dialecte de diagrammes temporels. Ceci aurait pu aider à l'approche que nous avons utilisée pour la méthode de la synthèse des

machines observatrices. Toutefois, le style de la spécification est limité par rapport aux contraintes temporelles et ne tient pas compte de certains comportements spécifiques.

Notre approche peut être résumée comme suit, voir figure 2. Très souvent les spécifications des protocoles de bus sont des fragments de chronogrammes et de longues descriptions verbales. Premièrement, nous avons proposé de transformer ces spécifications dans des formats - HAAD - de chronogrammes plus complets et plus rigoureux. Ceci implique une subdivision des transactions du bus en des chronogrammes fonctionnels plus petits en utilisant des diagrammes à actions feuilles (Leaf Action Diagrams (LAD)) [59] pour ensuite les combiner en utilisant les opérateurs de la composition hiérarchique du HAAD. L'étape la plus difficile de cette démarche est celle de transformer les spécifications de protocoles en spécifications HAAD, dans la mesure où les spécifications des protocoles sont objets à des interprétation subjectives, à des incomplétudes et à des ambiguïtés. Les diagrammes à actions feuilles (LAD) de la spécification HAAD décrivent les transitions sur les signaux et les contraintes temporelles entre elles. Ces LADs sont alors transformés en Machines à États Finis Synchrones Temporisées (Synchronous Timed Finite State Machines (STFSM)), qui peuvent s'étendre à des opérations de données si la spécification du HAAD est annotée. Toutes les machines à états des diagrammes feuilles fonctionnent de manière concurrente. Les diagrammes à actions hiérarchiques (HAD) qui lient les diagrammes feuilles (LAD) de la définition de protocole HAAD sont utilisés pour imposer la coordination des opérations des machines à états LAD, en utilisant des signaux dérivés des valeurs des états de ces machines et des opérateurs hiérarchiques spécifiques. Le réseau de machines à états STFSM ainsi créé devient un réseau de machines à états finis étendus EFSM, dans lequel les variables temporelles de ces machines sont implémentées comme des compteurs réinitialisables des pulsions de l'horloge du système, et ensuite converti dans un langage synthétisable synchrone c'est-à-dire Verilog ou VHDL pour produire un modèle RTL de la machine observatrice. Aucune action du bus n'est générée par la machine observatrice. Elle n'a qu'une seule sortie (*flag*) qui

reste initialisée à 0 (*Pas d'erreur*) et garde cette valeur tant que la spécification du protocole de communication n'est pas violée. Chaque fois que le protocole de communication est violé, le signal en sortie de l'observateur devient 1 (*erreur*).

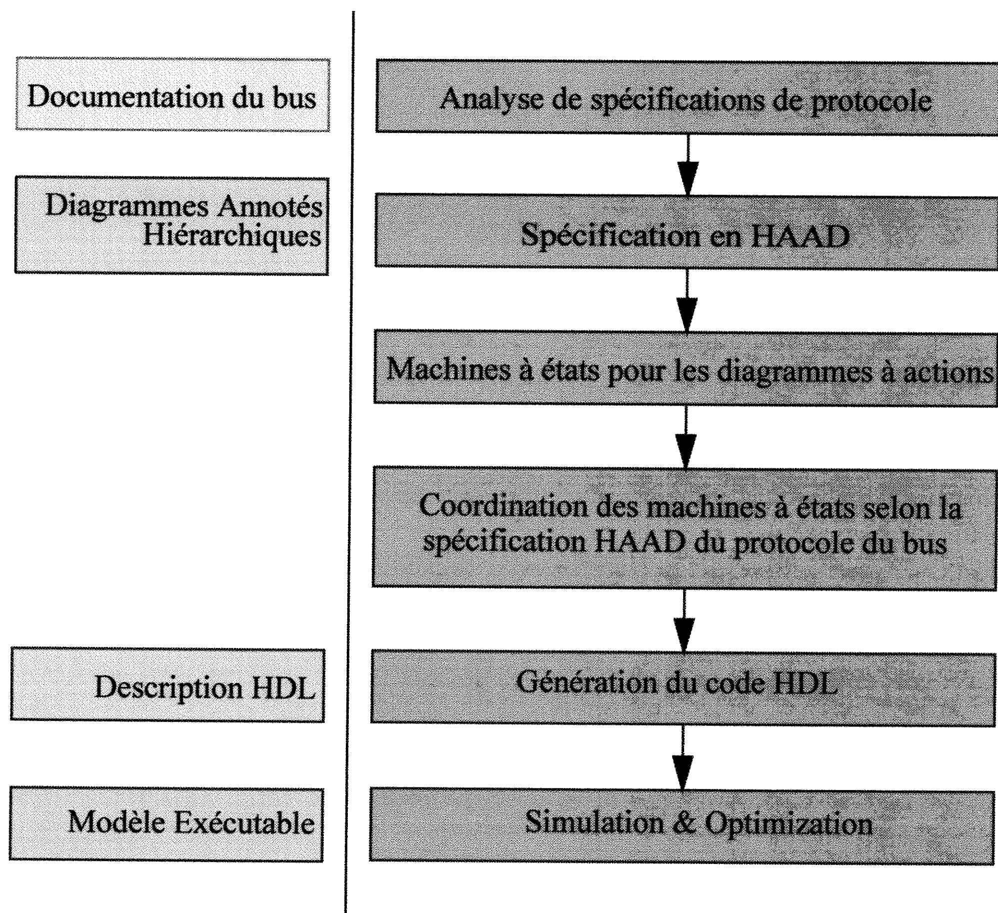


FIGURE 2. Résumé de l'approche utilisée pour la synthèse de la machine observatrice

La synthèse des machines à états, comme celle des contrôleurs d'interface à partir des chronogrammes, a fait l'objet de plusieurs recherches qui ont commencé avec la thèse de G. Borriello [19], et ont pris de l'importance avec la co-vérification matériels-

logiciels (HW/SW) et la synthèse des systèmes basée sur les IP. Toutefois, la technique présentée par Borriello dans sa thèse pourrait être potentiellement utilisée dans la synthèse de la machine observatrice si elle pouvait tenir compte de la hiérarchie complexe de la spécification HAAD. D'autre part, dans la référence [56], Delgado et al. présentent comment convertir des chronogrammes synchrones isolés en des propriétés en logique temporelle. Cette approche aussi aurait pu être utilisé dans la synthèse des machines observatrices, mais comme dans le cas de la méthode de Borriello, celle-ci ne peut pas accepter des spécifications complètes et exécutables des protocoles de bus sous la forme HAAD, en particulier lorsque les diagrammes sont composés par les opérations de composition hiérarchique. Il existe plusieurs autres techniques pour déduire des protocoles à partir de la dérivation des structures d'implémentation comme l'interconnexion des composantes à partir de la spécification de système [19] [58] [36] [29]. Ces méthodes ainsi décrites pouvaient nous aider pour la construction du protocole de communication entre les diagrammes à actions feuilles (LAD), pour en faire une machine observatrice. Mais dans notre cas, la structure de communication est spécialisée, régulière et récursive, et les techniques générales de synthèse ne lui sont pas appropriées. Bref, un modèle HAAD a une hiérarchie comportementale et non structurale.

Pour prouver l'exactitude de la méthode proposée pour la synthèse de machines observatrices des protocoles de bus, nous l'avons vérifiée formellement en utilisant des modèles abstraits.

La vérification de l'exactitude de la méthode pour la synthèse de la machine à états observatrice est faite en deux étapes: la vérification par simulation et celle par méthodes formelles. Pour ce faire, nous avons développé des modèles abstraits de la machine observatrice. Ces modèles sont comme des moules dont les structures servent à construire les machines STFSM et permettent de vérifier leur exactitude.

La simulation seule ne permet pas de vérifier les opérations de synchronisation pour un protocole arbitraire, c'est-à-dire un protocole qui spécifie une hiérarchie HAAD

arbitraire, et d'affirmer sans équivoque son exactitude. C'est la raison pour laquelle, pour vérifier que les équations de synchronisation que nous avons proposées sont toujours vraies dans toutes les situations, nous avons utilisé une méthode formelle (vérification de modèle). Nous avons utilisé un modèle abstrait des diagrammes à actions feuille dans lequel nous avons introduit des signaux de synchronisation qui sont différents des signaux du bus et qui ne participent pas aux activités de ce dernier. Ensuite, nous avons utilisé une approche de composition que nous avons vérifiée par VIS, un vérificateur de modèle [3]. La vérification comporte les phases suivantes:

1. Le modèle abstrait satisfait les propriétés des équations de synchronisation, et
2. La propriété des équations de composition des machines à états représentées par les instances des modèles abstraits des diagrammes à actions feuilles doit être préservée. Ceci veut dire que les interfaces d'une machine représentant une interconnexion de STFSM doivent êtres équivalentes aux interfaces des STFSM elles-mêmes. Ceci nous permet de parler d'une encapsulation récursive.

De ce fait, nous pouvons affirmer et conclure que les machines à états qui font partie du réseau que constitue la machine observatrice sont correctement synchronisées, quelque soit la hiérarchie HAAD donnée.

Les spécifications abstraites et les vérifications hiérarchiques ont permis de travailler sur des circuits de densité très grande, par des approches indirectes. Ces approches sont possibles grâce à la preuve par théorème et à la vérification de modèles comme celles présentées dans les références [10] [43] [23]. Notre modeste contribution est de montrer comment ces techniques peuvent être utilisées pour démontrer l'exactitude de protocoles de synchronisation dans un réseau de machines à états qui implémente une hiérarchie comportementale arbitraire.

1.3. Plan du mémoire

La suite de ce document se présente comme suit:

Dans le chapitre 2, nous faisons un résumé de la méthode de diagrammes annotés à actions hiérarchiques -HAAD-. Nous présentons premièrement les diagrammes à actions de type feuille qui constituent la base de cette méthode. Les diagrammes feuilles sont des chronogrammes classiques augmentés de données et de procédures. Nous présentons dans la deuxième partie de ce chapitre les diagrammes hiérarchiques qui permettent de composer les diagrammes feuilles pour en faire des spécifications complètes. Nous donnons dans la dernière partie de ce chapitre une brève description des diagrammes à actions hiérarchiques synchrones.

Dans le chapitre 3, nous présentons la méthode de construction de la machine à états observatrice. Nous définissons en premier lieu une machine appelée *machine à états finis synchrone temporisée (STFSM)* et la traduction d'un diagramme feuille en STFSM. Nous parlons ensuite de la coordination des STFSM en fonction des diagrammes à actions hiérarchiques, et enfin de la construction de machines STFSM pour les différentes opérations hiérarchiques.

Dans le chapitre 4, nous présentons la vérification de la méthode pour la synthèse de la machine à états observatrice. Ce chapitre est composé de deux parties principales, à savoir: la vérification des STFSM génériques et la vérification de la composition des STFSM en fonction des opérateurs HAAD.

Dans le chapitre 5, nous appliquons la méthode de la synthèse des machines à états observatrices aux fonctions de lecture et d'écriture du bus PCI. Nous donnons premièrement une description des opérations de lecture et d'écriture sur le bus PCI en nous basant sur les signaux de contrôle du bus. Nous présentons également dans la suite de ce chapitre la décomposition des fonctions de lecture et d'écriture en sous

fonctions, selon leur spécification HAAD, leur traduction en machines à états synchrones temporisées et leur composition pour créer une machine à état observatrice.

Nous concluons ce mémoire avec le chapitre 6 dans lequel nous résumons toutes les étapes que nous avons parcourues lors de ce projet et nous proposons une extension de ces travaux aux éléments de la méthode HAAD qui n'ont pas été traités dans ce projet. Nous proposons aussi des perspectives pour des travaux futurs à savoir le développement d'un compilateur de HAAD pouvant générer des codes source RTL pour la synthèse de la machine observatrice.

Avant de parler de la synthèse de la machine observatrice, nous présentons dans le chapitre suivant la méthode HAAD [1] sur laquelle se base notre projet.

CHAPITRE 2 *Spécification d'un système par le HAAD*

About 1000 instructions is a reasonable upper limit
for the complexity of the problems now envisioned.
- Herman Goldstine and John Von Neumann (1946)

2.1. Introduction

Les méthodes traditionnelles de spécification d'un système numérique sont basées sur la description informelle ou semi-formelle, en anglais pour la plupart, et sur des chronogrammes (Timing Diagrams - TD). Ces méthodes se sont avérées insuffisantes et incomplètes avec l'évolution prodigieuse des systèmes numériques, et la complexité de plus en plus grande de leurs fonctions, dans la mesure où les spécifications informelles sont le plus souvent sujettes à confusion et que les chronogrammes ne représentent que partiellement une fonction et une seule à la fois.

Bien concevoir un système commence par une spécification claire et adéquate qui représente le système dans son ensemble. L'évolution des systèmes numériques et leur complexité nous mène à proposer des techniques de spécifications formelles et rigoureuses pour leur conception.

La spécification formelle a pour objectif de décrire le comportement d'un système et de ses propriétés avant sa mise en oeuvre. Ces techniques sont plus souvent utilisées dans le cas des systèmes critiques (c'est-à-dire ceux pour lesquels le coût d'une erreur de fonctionnement est largement supérieur au coût de la conception) car il est alors possible de raisonner sur la spécification (prouver des propriétés) et de la vérifier avant l'implémentation du système.

La spécification formelle utilise un langage dont la syntaxe et la sémantique sont basées sur des procédés mathématiques. Bien que les propriétés des systèmes numériques incluent les comportements fonctionnels et temporels, les performances et les structures internes, leur spécification n'a connu un succès qu'en ce qui concerne les propriétés fonctionnelles. La tendance actuelle est d'intégrer différents langages de spécification, chacun capable de traiter différents aspects, de manipuler les performances, les contraintes temporelles et les contraintes de sécurité et architecturales de ces systèmes.

Plusieurs formalismes ont été utilisés pour représenter les spécifications à savoir:

1. La logique classique, la logique propositionnelle, la logique des prédicats de premier ordre, la logique d'ordre supérieur, la logique temporelle, etc...
2. Les automates à états finis sur des mots finis, les automates à états finis sur des mots infinis, les automates temporisés, etc...

Des méthodes formelles, telles que le Z, VDM et Larch [32], permettent de spécifier le comportement de systèmes séquentiels. Les états sont décrits en structures mathématiques ensemblistes (relations, fonctions). Les transitions sont données en terme de "*pré*" et "*post*" conditions. D'autres méthodes comme le *Constraint Satisfaction Problem (CSP)* [34], le *Constraint Logic Programming (CLP)* [8] [42], les logiques temporelles et les automates à entrées/sorties se concentrent sur la spécification des systèmes concurrents.

Un autre aspect des techniques de spécification est l'acte de décrire précisément les choses et d'obtenir une profonde compréhension du système; cela élimine des inconsistances, des ambiguïtés et des incomplétudes.

La méthode HAAD, dont la présentation se fait dans la suite de ce chapitre, est une description formelle et intuitive de diagrammes temporisés, vus comme des *scénarios* de comportements d'un système qui, en leur donnant des encapsulations appropriées, peuvent être composés pour donner une description finie ou infinie (cyclique) d'un comportement d'un système [59]. Cette méthode est constituée de deux niveaux de description: *les diagrammes à actions de feuilles et les diagrammes à actions hiérarchiques*.

2.2. Diagrammes à actions annotés et hiérarchiques (HAAD)

2.2.1. Diagrammes à actions feuille

Un diagramme à actions feuille est tout d'abord un chronogramme traditionnel, représentant un comportement élémentaire d'un système numérique. Par exemple, la fonction de lecture ou d'écriture sur le bus PCI. Un diagramme feuille est constitué d'un ensemble de *ports* et d'*événements* qui se produisent sur ces ports. Ces événements sont liés entre eux par des contraintes temporelles. Pour mieux comprendre les caractéristiques d'un diagramme à actions feuille, considérons l'exemple de la figure 3, qui est un fragment de la spécification de la fonction de lecture sur le bus PCI, basée sur la description de la référence "PCI System Architecture" version 2.1. [57].

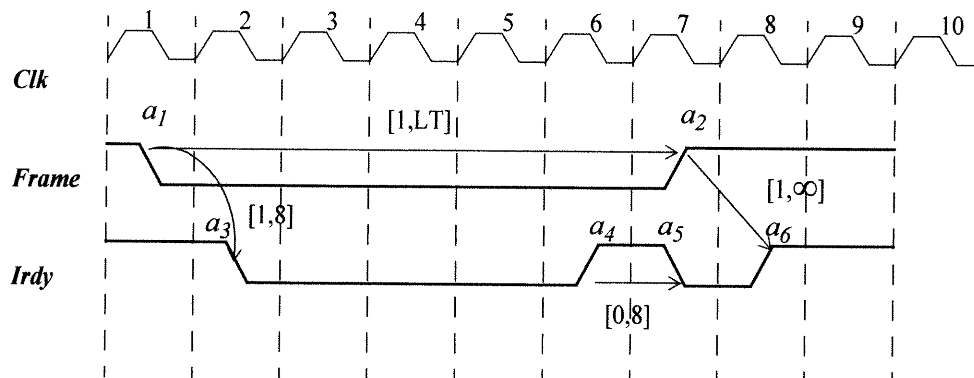


FIGURE 3. Diagramme à actions feuille représentant un fragment de la spécification de la fonction de lecture sur le PCI

La figure 3 montre que le diagramme feuille est défini par un ensemble de *ports*: *CLK*, *FRAME* et *IRDY*, qui ne sont autres que les liens de communication entre le système numérique et son environnement. Chaque port a un type. Celui-ci peut être tout type compatible avec le langage VHDL. Dans la référence [57], la spécification de la fonction de lecture ou d'écriture montre deux genres de types pour les ports: le type *booléen* encore appelé *std_logic* en VHDL, comme le *FRAME*, *IRDY*, et le type *donnée, adresse ou bus* comme AD[31:0] ou AD[63:0] qui représentent respectivement un bus d'adresse de 32 ou 64 bits. Les ports ont aussi une direction: *entrée (in), sortie (out)* ou *bidirectionnelle (inout)*, ou encore des ports internes invisibles de l'extérieur, que nous appelons *signaux*.

Sur chaque *port* du diagramme à actions feuille, nous avons des *actions* ou des *messages*. Les *actions* sont des changements d'état ou de valeur sur les ports en un instant donné (par exemple, dans la figure 3, le changement de valeur de 1 à 0 sur *IRDY* au deuxième cycle d'horloge). Les *messages* consistent en une mise à jour des valeurs des *ports*. Ces actions quant à elles gardent les propriétés des ports comme leur

direction sauf pour les *ports* d'*entrée-sortie (inout)* où chaque action doit être suivie de sa direction.

Dans la description des diagrammes à actions de type feuille, référons-nous à la figure 4. Chaque action possède l'une des étiquettes suivantes:

1. *constante,* qui dénote un symbole ou une valeur constante que le *port* doit atteindre et rester stable sur cette valeur jusqu'à ce qu'une nouvelle action se produise.
2. *valid,* qui indique que le port peut recevoir une valeur booléenne quelconque et gardera cette valeur jusqu'à ce qu'une prochaine action se produise.
3. *Don't-care,* qui signifie que le port a un comportement arbitraire ou non spécifié.

Tel que signalé précédemment, des actions peuvent être liées les unes aux autres par des relations temporelles. Une relation temporelle notée

$$C_{ij} = (a_i, a_j, [l_{ij}, u_{ij}])$$

indique que les temps d'occurrences t_i et t_j des actions a_i et a_j doivent satisfaire la relation suivante

$$l_{ij} \leq t_j - t_i \leq u_{ij}$$

Une contrainte est représentée par une flèche entre deux actions a_i et a_j, et étiquetée par un intervalle de temps. Dans la figure 3, l'exemple suivant

$$a_1 \xrightarrow{[1,LT]} a_2$$

indique que l'action a_1 si elle a lieu à un temps que l'on considère comme le temps t_1, alors l'action a_2 aura lieu au plus tôt à la prochaine unité de temps ou au plus tard au temps d'occurrence $LT > 0$ (LT représente le temps maximal que doit durer une transaction de lecture ou d'écriture sur le bus PCI). On appelle a_1 l'action *source* et a_2 l'action *puits*.

Se référant à l'inégalité exprimée plus haut, si $l_{ij} > 0$ alors la relation qu'elle exprime est une *précédence*, tant que l'action a_i a toujours lieu avant l'action a_j dans le temps (exemple de la figure 4). Si $l_{ij} \leq 0 \leq u_{ij}$ alors la relation est une *concurrence*. Dans une relation de *concurrence*, les actions a_i et a_j peuvent avoir lieu dans un ordre quelconque. L'essentiel est que la séparation de temps entre les temps d'occurrences t_i et t_j soit d'au moins l_{ij} unités de temps ou d'au plus u_{ij} unité de temps.

Lorsqu'une action *puits* dépend de plusieurs relations d'actions *sources* du même genre, celles-ci sont composées par les opérateurs *conjonctive*, *latest* ou *earliest,* et forment les contraintes temporelles. Voir Figure 4.

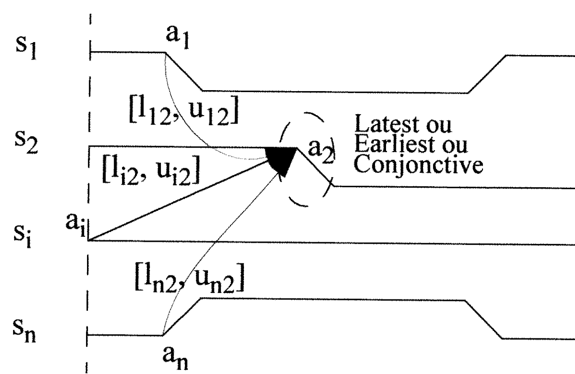


FIGURE 4. Diagramme à actions feuille avec des opérations

Soit $\{(a_1, t_1), \ldots, (a_n, t_n)\}$, l'ensemble des couples des événements et de leur temps d'occurrence, et soit $C_j = \{c_{ij} | i \in I\}$, l'ensemble de contraintes temporelles qui ont un événement puits a_j.

Un opérateur *Conjonctive (linéaire)* indique que les temps d'occurrences t_i et t_j doivent satisfaire la relation suivante

$$max(t_i + l_{ij}) \leq t_j \leq min(t_i + u_{ij}) \qquad avec \qquad i \in I$$

Le temps d'occurrence est un élément de l'intervalle $[t_i + l_{ij}, t_i + u_{ij}]$, avec i appartenant à I; ceci signifie que toutes les relations de C_j doivent êtres satisfaites simultanément.

Un opérateur Latest (max), indique que les temps d'occurrence doivent satisfaire la relation suivante

$$max(t_i + l_{ij}) \le t_j \le max(t_i + u_{ij}) \qquad avec \qquad i \in I$$

Ceci signifie que le temps d'occurrence t_j est déterminé par la dernière action *source* a_i qui s'est produite en fonction des bornes l_{ij} et u_{ij}. La relation entre les actions *sources* et l'action *puits* est dite *causale*. Ce qui implique que la relation temporelle C_{ij} est une relation de précédence où l'action a_j est le *prédécesseur* de toutes les actions a_i avec i appartenant à I.

Un opérateur Earliest (min) indique que les temps d'occurrence doivent satisfaire la relation suivante

$$min(t_i + l_{ij}) \le t_j \le min(t_i + u_{ij}) \qquad avec \qquad i \in I$$

Ceci signifie que le temps d'occurrence t_j de a_j est déterminé par la première action *source* a_i qui s'est produite. La relation entre l'action *source* et l'action *puits* est aussi *causale*. Ce qui implique que la relation temporelle C_{ij} est une relation de précédence où l'action a_j est le *prédécesseur* de la première action a_i avec i appartenant à I.

Les contraintes de temps peuvent être classées en deux genres: les contraintes *assume* et les contraintes *commit*. Les contraintes de temps *assume* indiquent des suppositions faites sur le temps d'occurrence des *actions entrées (inputs)*.

Les contraintes du genre *commit* spécifient l'intervalle de temps d'occurence des actions *sorties (output)*. L'instant d'occurence doit être à l'intérieur de l'intervalle temporel défini par la contrainte *commit* dont l'action est le *puits*. Les bornes temporelles des contraintes *commit* doivent avoir des valeurs finies.

Pour faciliter la composition hiérarchique, tout diagramme à actions feuille est délimité par deux actions virtuelles et implicites appelées *Begin* et *End*; Figure 5. L'action *begin* précède toute action du diagramme feuille alors que *end* succède à toutes les actions du diagramme feuille.

Satisfaire la spécification d'un diagramme à action feuille: La sémantique opérationnelle d'un diagramme à action est définie par son exécution dans l'environnement auquel sont rattachés les *ports in* et *inout* et l'observation des *ports out* et *inout*. Durant une exécution, un diagramme à actions *est* dit *satisfait* ou dans un *état satisfaisant* quand:

1. les spécifications des valeurs initiales sont satisfaites sur les ports *in* et *inout,*
2. les actions d'entrées respectent la séquence spécifiée, les contraintes de type *assume* et les valeurs spécifiées par les intervalles de temps d'occurrence.

Lorsqu'une erreur est détectée sur un port ou un signal durant l'exécution du diagramme à actions, celui-ci entre dans un état de non-satisfaction et son exécution est arrêtée. Une erreur d'exécution est produite si la spécification des valeurs initiales des ports n'est pas respectée, si les relations temporelles *assume/commit* ne sont pas satisfaites et si des actions inattendues se produisent sur un port.

Un diagramme est dit complètement satisfait si son exécution ne rencontre pas d'erreur, c'est-à-dire s'il reste constamment dans un état satisfaisant (autre que l'état d'erreur), de l'action *begin* jusqu'à l'action *end.*

La figure suivante est un schéma récapitulatif qui regroupe des caractéristiques d'un diagramme à actions feuille que nous avons évoqué plus tôt à savoir les actions *begin* et *end* ainsi que les étiquettes *Latest* ...

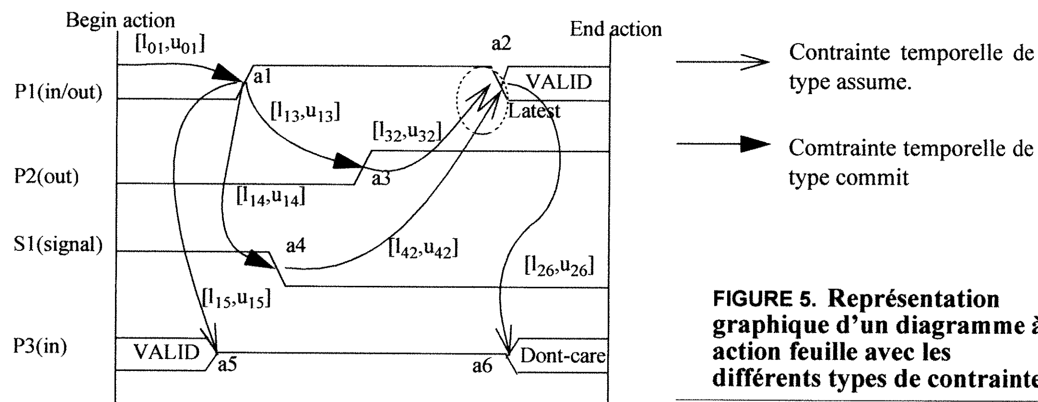


FIGURE 5. Représentation graphique d'un diagramme à action feuille avec les différents types de contraintes

2.2.2. Annotation des diagrammes feuilles [59]

Un diagramme à actions feuille peut être annoté par des paramètres *(entrées, sorties, entrées-sorties)*, des variables locales, des constantes génériques, des prédicats (fonctions booléennes) et des procédures compatibles aux types du langage VHDL.

Les paramètres et les variables locales peuvent être attachés aux événements d'un diagramme feuille. Les étiquettes d'une action ou d'une valeur initiale peuvent en général être des symboles arbitraires. Pour des étiquettes comme VALID, la sémantique est la même. L'étiquette symbolique a l'effet d'une déclaration automatique de variable qui est correspondante au type de la donnée de l'action du diagramme. Dans le cas de la spécification d'un événement en *entrée*, la valeur actuelle du port est attachée à la variable associée. Pour les événements en *sortie*, la valeur de l'action est assignée à la variable une fois que l'action est produite (dans ce cas, l'attachement d'une variable à la spécification de l'événement est obligatoire).

Les prédicats et les procédures ayant comme paramètres d'entrée des variables et/ou les paramètres des diagrammes feuilles peuvent être attachés à n'importe quelle action. Lorsque les événements se produisent, les procédures et les prédicats qui leur sont attachés sont exécutés dans un temps considéré comme *nul*. Ces prédicats ou procédures ne doivent pas contenir des références au temps, au délai ou à la synchronisation (par exemple: *WAIT* en VHDL). Les procédures décrivent la fonction d'un diagramme à actions. Si le prédicat attaché à un événement spécifique retourne *faux* après son évaluation, le diagramme à actions est *désactivé* et entre dans un état de *non satisfaction.*

À chaque événement donné sont attachés au plus une procédure et un prédicat. La sémantique de l'exécution au temps où un événement a lieu (ou plusieurs événements ont lieu au même moment) sont:

1. Mettre à jour toutes les variables associées aux événements d'entrées qui sont exécutés à l'instant même où les événements se produisent.
2. Évaluer tous les prédicats qui sont attachés aux événements dans (1) et aux événements de sortie qui sont supposés être exécutés dans l'instant courant.
3. Exécuter dans un ordre arbitraire toutes les procédures attachées aux événements de (2).
4. Mettre à jour les ports correspondants aux événements de sortie qui sont exécutés dans l'instant courant.

2.2.3. Diagrammes à actions hiérarchiques

Les diagrammes à actions peuvent être composés hiérarchiquement. Un diagramme à action hiérarchique Q est défini par un ensemble de ports *externes*, une liste de diagrammes à actions *fils* $(A_1,...,A_n)$, un opérateur de composition hiérarchique et un port de lien pour chaque A_i avec $i \in [1, ..., n]$. Le lien des ports établit la correspondance entre les ports externes des A_i et les ports *internes* ou *externes* de Q.

Les opérations de composition hiérarchique sont: *la concaténation*, *la boucle*, *la concurrence*, *le choix* et *l'exception*.

Soit Q un diagramme hiérarchique et $L = (A_1, ..., A_n)$ une liste de diagrammes à actions, $Q \neq A_i$.

La *Concaténation (Concatenation)* lie l'exécution de la liste des A_i dans leur ordre d'apparition. A_1 commence quand Q commence; A_{i+1} commence quand A_i finit; Q finit quand A_n finit.

La *Boucle (Loop)* répète l'exécution d'un A_i en un nombre fini ou infini de fois. La boucle est une concaténation particulière qui lie un diagramme à actions à lui même.

La *Concurrence (Parallel)* exécute les A_i en parallèle. A_1, ..., A_n commencent quand Q commence et Q finit son exécution quand tous les A_i ont fini.

Le *Choix (Delayed Choice)* fait une sélection entre les diagrammes à actions fils de la liste L. Ce choix est basé sur l'évaluation des prédicats et/ou les violations des hypothèses sur l'exécution des événements. $A_1,...,A_n$ représentent des comportements alternatives *(branches)* de la composition. $A_1,...,A_n$ commencent lorsque Q commence. Q finit lorsqu'un seul A_i finit. Si tous les A_i rentrent dans un état de *non satisfaction,* alors Q rentre dans un état de *non satisfaction*. Lorsque l'exécution Q commence, des prédicats attachés à tous les A_i de l'opération sont appelés et évalués, et l'opération décide du A_i qui continuera l'exécution selon le critère de choix. Si plusieurs A_i de la liste L sont évalués et acceptés pour continuer l'exécution, alors le choix est *retardé* et les branches qui passent dans un état de non satisfaction sont abandonnées au fur et à mesure que l'exécution progresse. Signalons que toutes les branches qui sont évaluées et acceptées ont le même comportement jusqu'à ce qu'une branche A_i se démarque des autres. Si une branche A_i termine son exécution alors que plusieurs autres *branches*

sont en cours d'exécution alors A_i est un *préfixe* des branches en cours d'exécution, ce qui est une erreur. Q passe alors dans un état de *non satisfaction*.

L'*Exception* contient trois diagrammes à action fils: A_N décrit le comportement normal, A_C le comportement de la condition d'exception et A_H le comportement de l'exception. A_N et A_C commencent lorsque Q commence. Si A_C termine son exécution avant le comportement normal A_N, l'exécution de A_N est arrêtée et l'exécution de A_H commence. Si A_H termine son exécution, alors Q aussi termine son exécution. Si A_N termine complètement son exécution avant A_C, alors A_C est arrêté et Q termine.

2.2.4. Annotation des diagrammes hiérarchiques

Comme les diagrammes à actions feuilles, les diagrammes hiérarchiques peuvent avoir des *paramètres (entrée, sortie, entrée/sortie)*, des *variables locales* compatibles à tout type *VHDL* et des constantes *génériques*. Pour les diagrammes de type choix, un prédicat peut être attaché à chaque branche du diagramme. Les arguments des prédicats peuvent être n'importe quelle variable et constante du diagramme. Les prédicats sont évalués à chaque exécution au début de l'exécution du diagramme choix. Seules les branches dont les prédicats ont été évalués *vrai* sont activées. De façon semblable, un prédicat peut être attaché à un diagramme *Boucle*. Il est évalué au début de chaque exécution du diagramme à actions fils de la boucle. Si la valeur du prédicat est fausse, alors l'exécution de la boucle est complétée. En cas contraire le diagramme boucle est arrêté. Le prédicat de la boucle peut être une variable ou une constante quelconque. Par ailleurs, des procédures ou des prédicats peuvent être attachés à la spécification des événements virtuels *Begin/End* des diagrammes à actions d'un diagramme hiérarchique Q. Ces procédures ou prédicats peuvent prendre leurs valeurs dans un sous-ensemble de l'ensemble des variables et des constantes du diagramme Q.

2.2.5. Description d'un diagramme à actions feuille en langage HAAD

La figure 6 représente un exemple de la spécification d'un diagramme feuille de la figure 5 en langage HAAD. Le langage HAAD est un langage basé sur la structure syntaxique du langage LISP. Toutes les clauses sont constituées d'un emboîtement de paires *(Mot_clé, Liste_des_valeurs)*. Pour une description complète de la grammaire du langage de HAAD, nous vous prions de vous rapporter à l'Appendice A de la référence [59].

Un modèle HAAD, même s'il contient un seul diagramme à actions, débute avec le mot clé *DefBehavior* suivi du nom que l'utilisateur donne au modèle. Il est suivi par une liste de ports. À chaque port est associé un nom, un mode (in, out, inout), un type conformément à la syntaxe de VHDL, et les interprétations des actions sur les ports comme *EVENT* pour les événements ou *MESSAGE* pour les messages. Une spécification HAAD peut être remplacée par un paramètre en utilisant les mots clés appropriés. Toute déclaration de paramètre est constituée par son nom, son mode, son type et ce, de façon analogue à la déclaration des ports. Les paramètres sont passés par référence et leur impact se limite au HAAD dans lequel ils sont déclarés. La déclaration des paramètres est suivie par la déclaration des signaux internes *(mot clé SIGNAL)*, des variables locales *(mot clé VARIABLE)* et des constantes *(mot clé CONSTANT)*.

Ce bref aperçu de la syntaxe du langage HAAD est tout aussi valable pour les diagrammes à actions feuille que les diagrammes de composition hiérarchique. Les diagrammes à actions feuille sont déclarés par le mot clé *LEAF* qui permet d'ouvrir une nouvelle région déclarative. Chaque port est spécifié comme une succession d'une ou de plusieurs actions. Le *mode* (in, out, et inout) des actions au niveau des feuilles leur est légué par la direction des ports. Dans le cas d'un port à direction *inout*, le mode de chaque action de ce port doit être spécifié à chaque fois que l'action est déclarée. La séquence des actions de chaque port est définie par le mot clé *CARRIER_SPEC,* suivie par la spécification de la valeur initiale de chaque port et par la liste de la spécification des actions. Chaque action est identifiée par le mot clé *ACTION_SPEC*. Une action est

spécifiée par son nom suivi de son étiquette qui peut être une constante ou un symbole dénotant une constante, un symbole spécifique *VALID* ou *DONT_CARE*.

```
(DEFBEHAVIOR EXAMPLE_HAAD_FIGURE5
 (PORTS
        (PORT P1 INOUT "std_logic" EVENT)
        (PORT P2 OUT "std_logic" EVENT)
        (PORT P3 IN "std_logic" EVENT))
 (SIGNAL S1 "std_logic" EVENT "1")
 (LEAF
        (CARRIER-SPEC P1 (INITIAL-SPEC (CONSTANT "0"))
               (ACTION-SPEC 'a1 (CONSTANT "1") (DIRECTION OUT))
               (ACTION-SPEC 'a2 (VALID) (DIRECTION IN))
        (CARRIER-SPEC P2 (INITIAL-SPEC (CONSTANT "0"))
               (ACTION-SPEC 'a3 (CONSTANT "1"))
        (CARRIER-SPEC S1 (INITIAL-SPEC (CONSTANT "1"))
               (ACTION-SPEC 'a4 (CONSTANT "0"))
        (CARRIER-SPEC P3 (INITIAL-SPEC (VALID))
               (ACTION-SPEC 'a5 (CONSTANT "Z"))
               (ACTION-SPEC 'a6 (DONT-CARE)))
 (PRECEDENCE 'start-action 'a1 (INTENT COMMIT) (CMIN l01) (CMAX u01))
 (PRECEDENCE 'a1 'a3 (INTENT COMMIT) (CMIN l13) (CMAX u13))
 (PRECEDENCE 'a1 'a4 (INTENT COMMIT) (CMIN l14) (CMAX u14))
 (LATEST
        (PRECEDENCE 'a3 'a2 (INTENT ASSUME) (CMIN l32) (CMAX u32))
        (PRECEDENCE 'a4 'a2 (INTENT ASSUME) (CMIN l42) (CMAX u42)))
 (PRECEDENCE 'a1 'a5 (INTENT ASSUME) (CMIN l15) (CMAX u15))
 (PRECEDENCE 'a2 'a6 (INTENT ASSUME) (CMIN l26) (CMAX u26))))
```

FIGURE 6. Spécification en langage HAAD du diagramme à actions feuille de la figure 5

2.2.6. Diagrammes synchrones

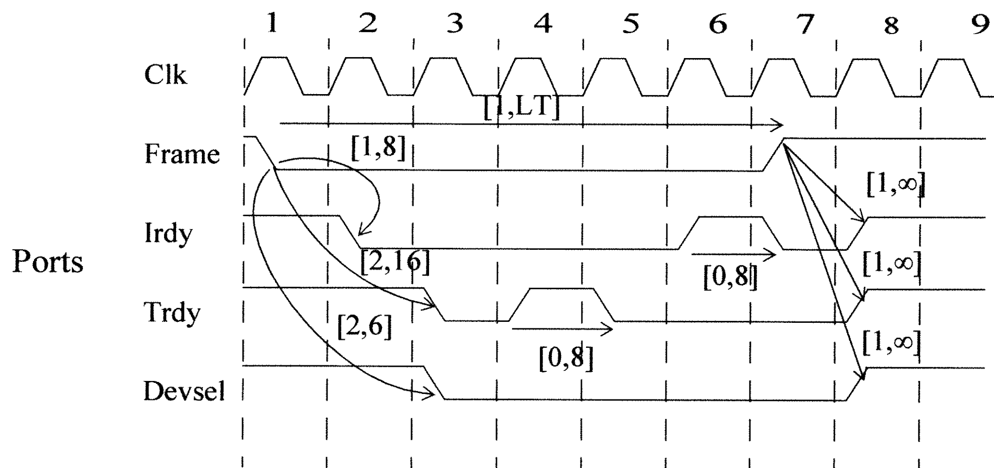
Les diagrammes synchrones ont les mêmes caractéristiques que les diagrammes feuilles à la seule différence qu'une horloge leur est ajoutée, et tout comportement temporel est spécifié en terme de cycles de cette horloge. L'exécution des séquences des actions sur un port est synchronisée par l'horloge.

Deux types de temps sont distingués dans les horloges: le temps *discret* ou *par intervalles* et le temps *continu* ou *réel*. Mathématiquement, le temps tel que nous le vivons quotidiennement nous apparaît continu parce qu'il se déroule de façon ininterrompue. On ne passe pas tout à coup d'une minute à l'autre, en laissant un trou. Le temps, en effet, parcourt tous les points qui séparent un instant t et un instant suivant $t + 1$ augmenté d'une minute. Contrairement au temps continu ou réel, un temps est discret si le temps n'est considéré qu'à certains points discrets du déroulement du temps. Ces points ou instants sont des intervalles de temps bien définis et de longeurs égales, et résultent d'un découpage à part égale du déroulement du temps continu. Chaque intervalle est considéré comme une unité de temps appelée dans notre cas *cycle d'horloge*.

Dans le cas d'un diagramme à actions synchrone, le temps discret veut dire que les événements sont produits et pris en compte seulement à des instants précis qui sont les *fronts actifs d'une horloge (montant ou descendant)*. Les procédures et prédicats sont évalués entre les fronts actifs de deux cycles d'horloge et les sorties sont produites à chaque front actif du cycle du temps indiqué par la spécification. Dans le cas de notre étude, le temps est discrètement continu et chaque graduation est un *entier naturel*; une *unité* étant égale à un *cycle d'horloge.*

La figure 7, prise ici comme exemple de diagramme synchrone, représente les caractéristiques des signaux de contrôle d'une transaction de lecture du bus PCI. Cette figure permet non pas de montrer la sémantique de la transaction de lecture du bus PCI, mais de montrer les spécifications d'un diagramme synchrone. Sur cette figure, l'horloge *clk* est active au front montant. Chaque cycle est délimité par les lignes verti-

cales en pointillés et numéroté par des entiers naturels. La première action du signal *Frame* a lieu au 1er cycle d'horloge et sa dernière au 7ème cycle.



LT est le temps de latence d'une Transaction de Lecture sur le PCI

FIGURE 7. **Exemple de diagramme synchrone: Transaction réduite de lecture sur le bus PCI.**

2.3. Conclusion

La conception d'un système numérique dépend du niveau de clarté de sa spécification. Plusieurs équipes se sont penchées sur le problème de la spécification formelle pour réduire les ambiguïtés et les interprétations subjectives dans les protocoles. La plupart des approches trouvées par ces équipes, aussi efficaces soient elles, ne sont capables de traiter que des aspects particuliers de la spécification. La méthode HAAD proposée dans la référence [59] est la plus complète. Elle prend en compte les aspects fonctionnels du système et traite des protocoles de système quel que soit le degré de leur complexité, par une description de chronogrammes rigoureuse, basée sur une structure hiérarchique assez complète.

Le chapitre suivant présente la méthode de la construction de la machine à états observatrice. Il introduit premièrement la description des machines à états et montre comment ces machines sont utilisées pour construire une machine à état observatrice. Il parle également de la décomposition des diagrammes à actions feuilles en diagrammes plus simples et de l'introduction de signaux spécifiques qui sont utilisés pour coordonner les activités de ces diagrammes.

CHAPITRE 3

Construction des machines observatrices à partir des spécifications HAAD

The understanding of the theory of a routine may be greatly aided by providing, at the time of construction one or two statements concerning the state of the machine at well chosen points... In the extreme form of the theoretical method a watertight mathematical proof is provided for the assertions. In the extreme form of the experimental method the routine is tried out on the machine with a variety of the initial conditions and is pronounced fit if the assertions hold in each case. Both methods have their weaknesses.
Alan Mathison Turing, Ferranti Mark | Programming Manual (1950)

3.1. Introduction

Dans ce chapitre, nous allons considérer que la spécification littérale des protocoles de bus est tansformée en spécification HAAD.

La structure générale d'une machine observatrice après sa réalisation peut être vue comme un réseau de machines à états liées les unes aux autres par des modules de composition hiérarchiques. Pour la construire, nous avons utilisé une approche simple qui peut être regroupée en trois points :

1. Décomposer les diagrammes feuilles de la spécification du protocole de bus en diagrammes plus simples et les lier par des opérations de composition hiérarchique, de telle sorte que la spécification du protocole du bus soit respectée par la composition des diagrammes feuilles qui résultent de la décomposition.
2. Définir pour chaque diagramme feuille qui résulte de la décomposition une machine à états finis.

3. Connecter ces machines à états par des modules logiques qui réalisent les règles de la composition hiérarchique définies lors de la décomposition.

Cette technique en trois points permet de décrire de façon plus concise la machine à états observatrice et d'éviter le problème d'explosion de la description si tous les diagrammes feuilles sont décrits par une seule machine à états.

Dans ce chapitre, nous allons présenter la décomposition des diagrammes feuilles, la transformation des diagrammes résultant de la décomposition en machine à états finis synchrone et temporisée (STFSM) et l'interconnexion de ces machines pour produire la machine observatrice.

3.2. Décomposition des diagrammes feuilles.

Lors de notre étude, nous avons remarqué qu'il est très difficile de transformer un diagramme feuille représentant une fonction d'un système (par exemple, la lecture du bus PCI) en machine STFSM sans se confronter au problème d'explosion de la description. La complexité d'une machine numérique croît alors de façon exponentielle par rapport à la complexité du diagramme feuille. En effet, la machine observatrice doit prendre en considération toutes les combinaisons des entrées dans chaque état. Pour remédier à ce problème, nous proposons de décomposer les diagrammes feuilles en diagrammes de même nature, mais moins complexes. Cette décomposition est basée sur des relations entre les diagrammes feuilles et les diagrammes hiérarchiques, et sur des techniques de décompostion de circuits numériques [5], [7], [4], [2], [6].

Dans la suite de cette section, nous présentons l'approche de base que nous avons utilisée dans la décomposition d'un diagramme feuille en plusieurs sous diagrammes moins complexes. Nous avons basé notre étude sur des cas généraux et avons montré quelques exemples sur un fragment de la spécification de la fonction de lecture sur le

bus PCI. Nous avons utilisé deux méthodes de décomposition: la *décomposition parallèle* et la *décomposition en cascade*. De plus, nous avons défini des règles de façon *ad hoc* pour faciliter la décomposition.

3.2.1. La Décomposition en Parallèle d'un Diagramme Feuille

La décomposition parallèle permet un découpage d'un diagramme feuille f en sous diagrammes dont les exécutions se font dans le même intervalle de temps. Dans le cas de notre étude, nous avons basé cette décomposition parallèle sur le lien que les ports ont les uns avec les autres. Si une action a_p sur un port p est liée par une flèche à une autre action a_q sur un port q, alors les ports p et q sont isolés et vont constituer un sous diagramme de f. Si nous prenons l'exemple du diagramme de la fonction de lecture sur le bus PCI représenté par la figure 7 au chapitre 2, nous pouvons faire une décomposition parallèle si nous regroupons les signaux selon leur dépendance et obtenir l'exemple de la figure 3 au chapitre 2 comme sous diagramme.

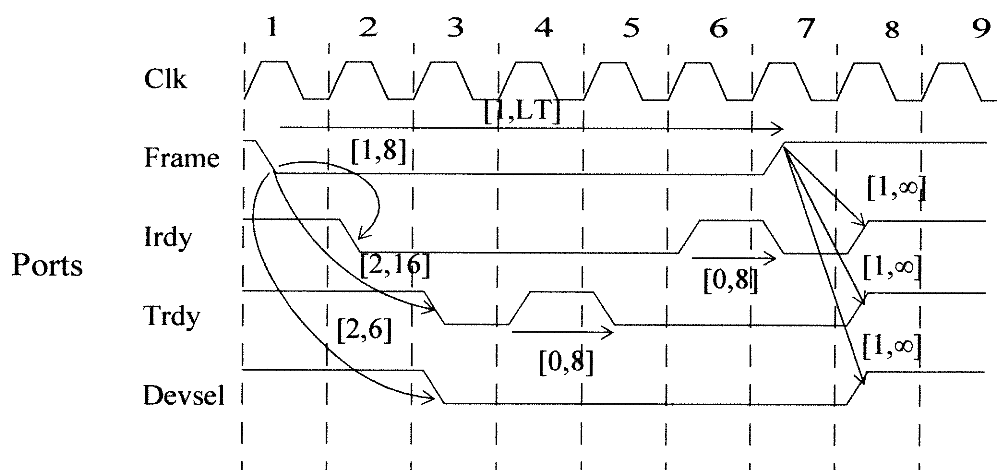


FIGURE 8. Transaction réduite de lecture sur le bus PCI représentant un diagramme feuille *f*

La décomposition parallèle d'un diagramme feuille suit les règles suivantes:

Règle1: Regrouper deux à deux des paires d'actions sur des ports différents liés par une ou plusieurs contraintes pour en faire des diagrammes feuilles.

La règle 1 peut être illustrée par la figure 9. Si nous regardons bien la figure 8 à partir de laquelle nous avons obtenu la figure 9, nous remarquons que les ports *FRAME* et *IRDY* ont deux paires d'actions qui sont liées. Nous pouvons ainsi isoler les comportements des deux ports et en faire un diagramme feuille.

Règle2: Créer des diagrammes feuilles pour tous les ports du diagramme f qui ne sont pas couverts par la règle 1.

La règle 2 permet de créer des diagrammes feuilles pour des ports dont les actions ne sont pas liées avec d'autres actions de ports différents. Dans le cas de la figure 8, nous n'avons pas de port qui n'est pas lié à un autre par une contrainte.

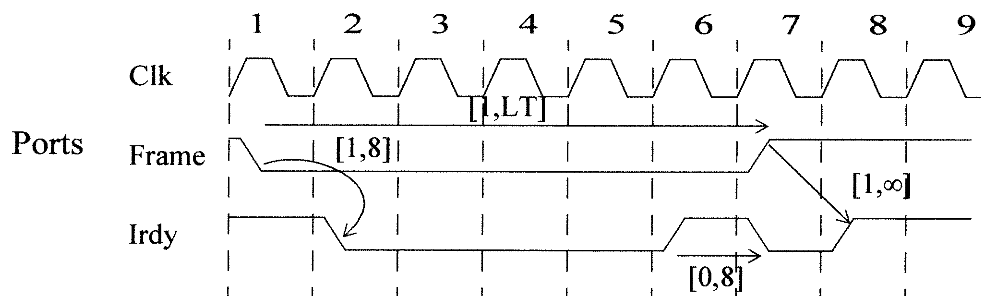


FIGURE 9. Exemple de diagramme feuille résultant de l'application de la règle 1 de la décomposition parallèle sur le diagramme *f*

3.2.2. La Décomposition en cascade d'un diagramme feuille

La décomposition en cascade se fait par rapport au temps chronologique du diagramme feuille. Ce découpage du diagramme principal doit tenir compte de la progression temporelle de celui-ci. Un diagramme feuille obtenu de la décomposition en cascade doit traiter au moins d'une contrainte entre une paire d'actions de deux ports différents ou une paire d'actions d'un même port.

Dans le cas où il y a eu au préalable une décomposition parallèle du diagramme feuille principal, il est plus judicieux d'utiliser ces sous diagrammes pour les décomposer en cascade afin d'obtenir des sous diagrammes encore plus simples. Prenons par exemple la figure 9 précédente qui est le résultat d'une décomposition parallèle. En la décomposant en cascade, on obtient les boîtes de la figure 10. Chaque boîte représente un diagramme feuille plus petit qui peut être représenté plus facilement par une machine à états.

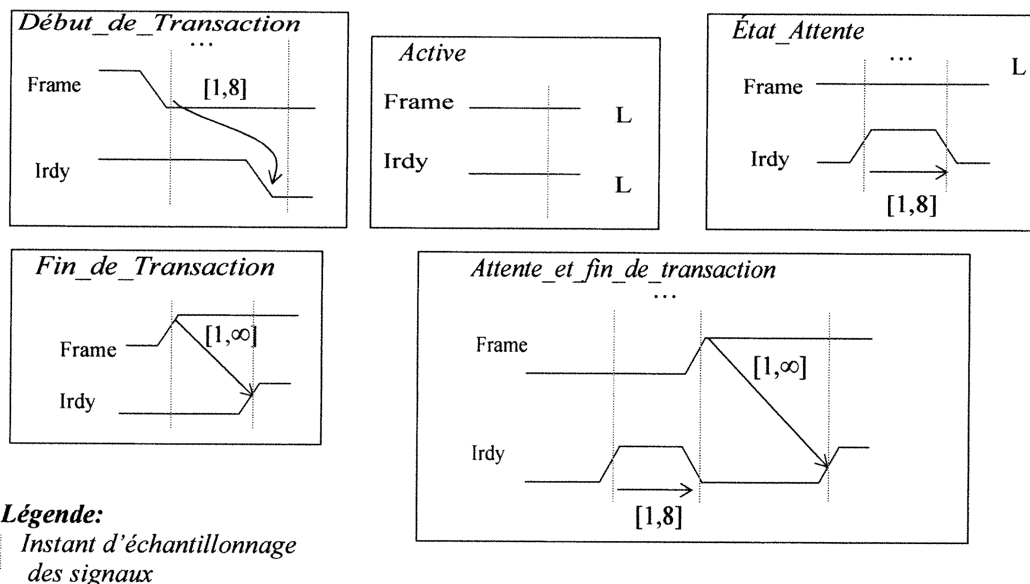


FIGURE 10. Décomposition en cascade du diagramme feuille représenté par la figure 9

Nous présenterons plus tard comment les diagrammes feuilles représentés par les boîtes de la figure 10 sont liés par des opérateurs de composition pour que la figure 10 soit fonctionnellement équivalente à la figure 9. La décomposition en cascade d'un diagramme suit la règle suivante:

Règle3: Décomposer les diagrammes feuilles (résultant d'une décomposition parallèle préalable si possible) en diagrammes feuilles plus petits et décrits comme suit:

i). Créer un diagramme feuille pour une paire d'actions liée par une contrainte sur le même port ou sur des ports différents.(Exemple de la figure 11).

ii). Créer un diagramme feuille pour chaque combinaison logique des ports deux à deux regroupés par la règle 1. Exemple de la boîte "active" dans la figure 10.

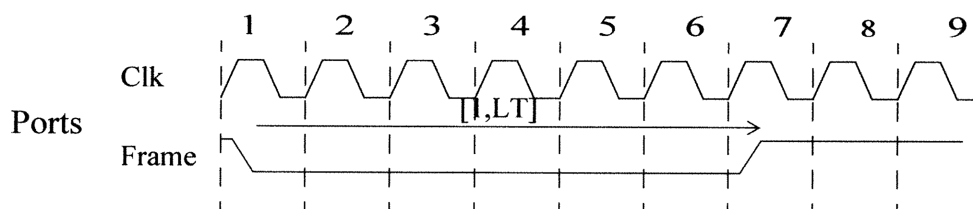


FIGURE 11. Exemple de diagramme feuille résultant de l'application de la règle 3. i) de la décomposition cascade sur le diagramme feuille de la figure 9

Une fois la décomposition faite, une machine STFSM est construite pour chaque fragment de diagramme feuille.

Un problème simple, mais important, de la décomposition est de pouvoir recomposer les machines afin de retrouver le comportement original. À cet effet, nous allons parler de la composition parallèle et de la composition en cascade, et montrer que le choix et

la boucle sont des opérations de composition qui sont déduites des deux précédentes. Nous allons nous baser sur les opérateurs de la méthode HAAD [59].

La composition *parallèle* est représentée par la figure 12 qui suit. Ici, les deux machines M_1 et M_2 qui représentent des fragments de diagramme reçoivent du même vecteur d'entrées $x=(x_0,...,x_{n-1})$ de n éléments les données nécessaires pour traiter les signaux dont les comportements sont vérifiés à partir d'une interface. Ces machines fonctionnent de façon indépendante, mais synchrone à l'horloge du système. Les sorties ou les résultats produits par les machines M_1 et M_2 sont alors composés par une fonction combinatoire C qui génère un résultat booléen indiquant si le fonctionnement observé par les machines est conforme au protocole ou non.

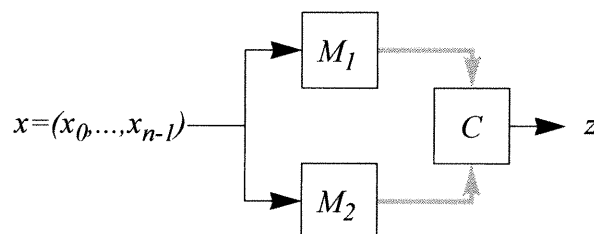


FIGURE 12. Composition parallèle

Dans le cas de la composition *cascade*, comme celui de la composition parallèle, les deux machines M_1 et M_2 reçoivent les mêmes entrées $x=(x_0,...,x_{n-1})$ mais elles ne fonctionnent pas indépendamment. La machine M_2 dépend d'une entrée auxiliaire qui est la sortie de M_1. Dans ce cas, la sortie de la machine M_2 constitue la sortie générale de la machine composée.

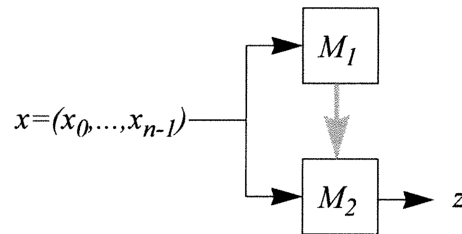


FIGURE 13. **Composition en cascade**

La composition *choix* peut être vue comme une composition cascade suivie d'une composition parallèle, à la seule exception que la fonction de composition C est différente dans les deux cas comme le montrent les opérations de la méthode HAAD.

La composition *boucle* est la composition cascade sur une même machine. Parmi les entrées, nous introduisons une variable *exit* qui permet d'arrêter l'exécution de la boucle.

Nous donnerons plus de détail sur les opérations de composition lorsque nous présenterons les équations de composition dans les sections suivantes.

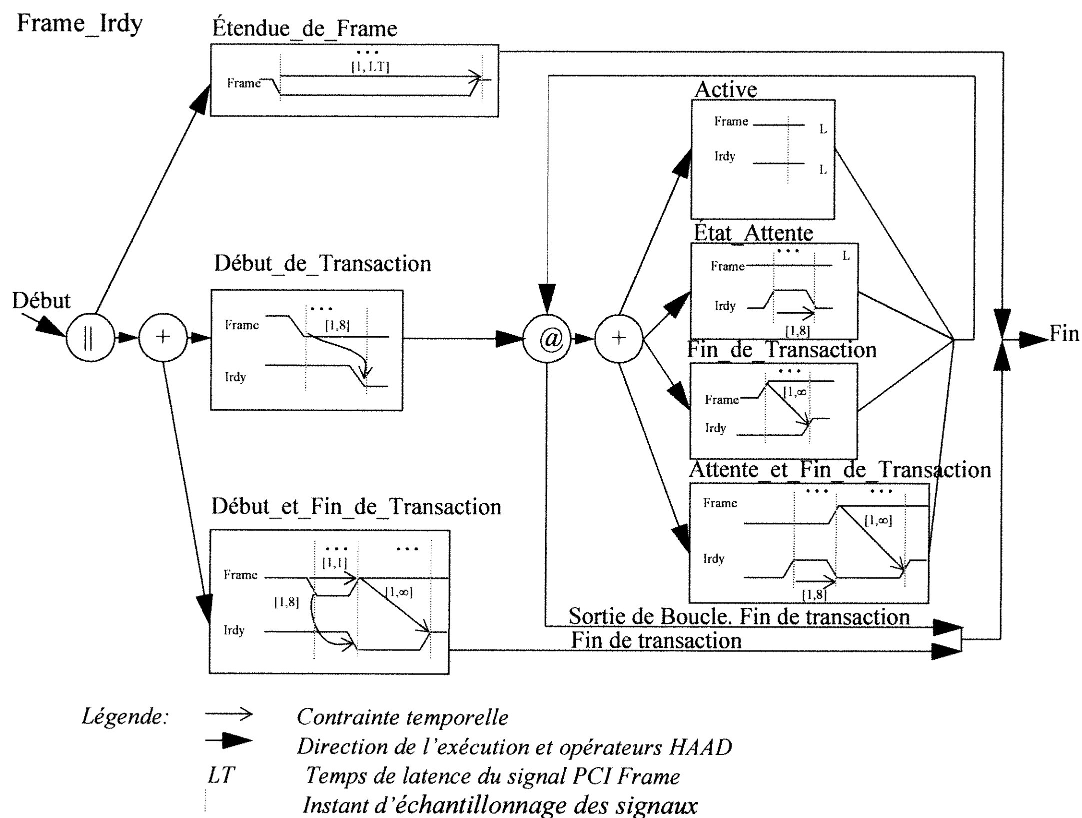


FIGURE 14. Modèle HAAD du fragment Frame_Irdy de la transaction de lecture du bus PCI. Recomposition des fragments pour donner fonctionnellement Frame_Irdy

La figure 14 ci-dessus est un exemple de composition qui permet de reconstituer le fragment de la fonction de lecture sur les signaux Frame et Irdy du bus PCI tel que montré par la figure 9. Nous remarquons que la figure 10 et la figure 11 sont représentées dans la figure 14 en plus d'une boîte, à savoir *Début_et_fin_de_transaction*, qui représente un comportement exceptionnel lorsque la transaction de lecture ne contient qu'une phase de donnée.

Comment fonctionne la figure 14 ?

Au *début* de la composition, nous avons une opération parallèle qui compose la boîte *Étendu_de_Frame* et une opération de choix. L'opération de choix se situe sur les boîtes *Début_de_transaction* et *Début_et_Fin_de_Transaction.* Lorsque le choix est exécuté, une seule des deux boîtes est exécutée du début jusqu'à la fin, dépendamment des valeurs des signaux. Si c'est la boîte *Début_et_Fin_de_Transaction* qui est exécutée jusqu'à la fin, alors la transaction des deux signaux *Frame* et *Irdy* est terminée lorsque cette boîte et la boîte *Étendue_de_Frame* sont exécutées jusqu'à la fin. Dans le cas où c'est la boîte *Début_de_Transaction* qui est exécutée dans l'opération de choix, cette boîte est alors concaténée à une opération de boucle sur une opération de choix. L'opération de choix est exécutée à chaque itération de la boucle et elle porte sur les boîtes *Active*, *État_Attente*, *Fin_de_Transaction* et *Attente_et_Fin_de_Transaction.* Dans ce cas, l'exécution est terminée pour le fragment de la fonction de lecture du bus PCI lorsque l'exécution de la boucle est terminée sans erreur et que l'une des deux boîtes *Attente_et_Fin_de_Transaction* et *Fin_de_Transaction* est la dernière à être exécutée.

Dans la section suivante, nous présentons la machine à états qui permet de décrire les diagrammes feuilles et les compositions de ces diagrammes.

3.3. Machine à états finis synchrone et temporisée (STFSM)

La machine à états finis synchrone et temporisée a été inspirée par les caractéristiques des machines à états finis et des machines temporisées [39] [40]. Dans la proposition de la méthode pour la synthèse des machines à états observatrices à partir des spécifications HAAD, nous nous sommes basés sur la communication entre automates, c'est-à-dire que ces machines sont décrites par un réseau d'automates capables de communiquer les uns avec les autres à travers des interfaces définies par un ensemble d'équations appelé ici *équations de composition.*

La machine observatrice peut être représentée par la machine générique, que nous appelons en anglais *Synchronous Timed Finite State Machine (STFSM),* qui est un modèle général synchrone pour décrire un diagramme feuille. Elle est caractérisée par trois états particuliers appelés états de composition qui lui permettent d'être composée avec d'autres machines.

Une machine STFSM M est formellement définie comme:

$M = (Q, \Sigma, D, S, O, F, \delta, \omega, q_0)$, où

Q est l'ensemble des états de la machine M.

$q_0 \in Q$ l'état initial.

Σ : l'alphabet d'entrée défini par les symboles qui sont encodés par les valeurs des ports et des signaux du bus, par les valeurs de registres qui permettent de vérifier la progression des événements sur le bus et des signaux *ack_in*, *reset*, *go* de synchronisation inter-machines dont nous discuterons plus tard.

D: l'ensemble des compteurs utilisés pour mesurer le temps écoulé entre les actions. Les compteurs $d_i \in D$ sont incrémentés d'une unité à chaque front actif de l'horloge du système.

Lorsque les machines à états sont initialisées à l'état $q_0 \in Q$, les compteurs sont initialisés à 0. Pour chaque transition où le temps est compté, le compteur associé à cette transition est incrémenté jusqu'à ce que la machine passe à l'état suivant. Le compteur est réinitialisé à ce moment là et peut être réutilisé pour une autre transition.

S: l'ensemble des opérations d'initialisation sur les compteurs associés aux transitions de STFSM. Ainsi une écriture $d_i := 1$ veut dire que le compteur d_i est initialisé à 1. Nous parlons ici de réinitialisation à 1 car une transition est faite d'un état à un autre durant un cycle d'horloge. Cette période de transition doit être prise en compte par le compteur de la transition.

O: l'ensemble des symboles de sortie encodés par des ports ou des signaux associés aux états et des signaux *ack_out, end, err* de synchronisation inter-machines.

F: $F \subseteq Q$, est l'ensemble des états finaux.

δ : est la fonction de transition des états.

ω : est la fonction de transition des sorties.

Par exemple, la transition suivante

$$q_1 \xrightarrow{ab=00,\ ack_in\ /\ ack_out} q_2$$

indique que la machine STFSM transite de l'état q_1 vers l'état q_2 lorsque *00* sont les valeurs que portent les signaux *a*, *b* sur le bus. Le signal *ack_in* est actif. La transaction produit la sortie *ack_out.* Une famille de transitions pour plusieurs valeurs de compteurs peut être représentée en ajoutant un prédicat sur la transition représentée par l'arc suivant

$$q_1 \xrightarrow{...d=3,\ d>2\ /\ d:=1\ ...} q_2$$

qui indique que la valeur du compteur *d* est *3* lors de la transition de q_1 vers q_2, que cette transition est valide seulement si la machine reste dans l'état q_1 pendant plus de deux cycles d'horloge. Durant cette transition, le compteur *d* est réinitialisé à 1.

Nous avons parlé des signaux de composition plusieurs fois dans les sous sections précédentes.
En quoi consistent-ils?

Nous avons distingué six signaux de communication. Trois signaux en entrée, à savoir: *go*, *reset*, et *ack_in*, et trois signaux de sorties, à savoir: *end*, *ack_out*, et *error*. Les signaux de composition dépendent des valeurs des autres signaux ou des états dans lesquels se trouvent les STFSM correspondants.

1. *go* est le signal qui donne l'ordre de départ à un STFSM. Chaque STFSM reste dans son état initial tant que son signal *go* n'est pas activé.

2. *reset* est le signal de réinitialisation d'une machine STFSM. Le signal *reset* de chaque machine est activé lorsque celui de la machine à états observatrice est activé.
3. *ack_out* est le signal que la machine STFSM produit lorsqu'elle reçoit le signal *go* et qu'elle quitte l'un de ses états de composition.
4. *ack_in* est le signal que reçoit une machine lorsqu'elle est dans son état final ou d'erreur et qu'elle est composée en cascade avec une autre machine. En effet, c'est le signal de sortie *ack_out* de la machine suivante qui devient le signal *ack_in* de la machine courante. C'est aussi le signal qui permet à la machine de quitter son état final ou son état d'erreur pour retourner soit à son état initial, soit à son premier état intermédiaire, soit à son état d'erreur.
5. *end* est un signal de sortie qui est produit lorsque la machine rentre dans son état final.
6. *error* est aussi un signal de sortie que la machine produit lorsqu'elle est dans son état d'erreur.

La figure 15 est un exemple de machine STFSM qui représente le diagramme feuille de la boîte *Attente_et_Fin_de_Transaction* de la figure 10. Sur cette machine, nous remarquons *{0, 1, 2, 3, 4, 5}* l'ensemble des états, *0* l'état initial, *{4, 5}* l'ensemble des états finaux, $\{01, 00_R, 10, 11, go, ack_in, reset\}$ l'alphabet des entrées, $\{d_1, d_2\}$ l'ensemble des compteurs, les étiquettes portées par les flèches de transition qui constituent les fonctions de transition, $\{d_1{:}=1, d_2{:}=1\}$ l'élément des opérations d'initialisation de compteur(s) et *{err, end, ack_out}* l'ensemble des symboles de sorties. Dans cette figure, le symbole "~" représente une négation. Aussi, sur la figure, nous remarquons deux types de sortie. *ack_out* est une sortie qui dépend des transitions *0* à *1*, *4* à *1*, et *5* à *1*, et *err, end* sont des sorties qui dépendent des états *5* et *4*.

Comment fonctionne la machine STFSM de la figure 15 par rapport au diagramme feuille représenté par la boîte Attente_et_Fin_de_Transaction ?

Au début de l'exécution, la machine reste dans son état initial *0* tant qu'elle n'a pas reçu son signal de départ *go* et ce, quelles ques soient les entrées *Frame* et *Irdy*. Lorsque qu'elle a reçu le signal *go*, si la valeur précédente des signaux *Frame* et *Irdy* est *00* (ce qui est représenté par 00_R), et la valeur actuelle des entreés est *01*, alors la machine transite de l'état *0* à l'état *1*. Et si nous regardons la boîte *Attente_et_Fin_de_Transaction* de la figure 10, nous remarquons que la valeur des signaux en entrée *Frame Irdy* avant le premier échantillonnage est *00*. La valeur des signaux *Frame Irdy* au premier échantillonnage est *01*. Ce qui coïncide avec le comportement de la machine lorsqu'elle est dans son état initial *0*. Si nous revenons à la machine à état, elle active le signal *ack_out* et affecte *1* au compteur d_1. Pour toute combinaison de valeurs des signaux *Frame* et *Irdy* autres que celle représentée sur l'arc de transition, la machine transite de l'état *0* vers l'état *4*. Elle produit toutefois *ack_out* à cause du signal *go* sur la transition et *error* lorsqu'elle est dans l'état *4*.

De l'état *1*, la machine peut rester jusqu'à 7 itérations avec 01 pour la valeur des signaux *Frame Irdy*. Ceci est réprésenté sur le diagramme feuille *Attente_et_Fin_de_Transaction* par la période d'attente [1, 8] après le premier échantillonnage. De l'état *1* toujours, si la machine reçoit comme entrée *10*, elle passe à l'état *3*. Si elle reçoit 11, alors passe alors à l'état *2* car on considère que la dernière phase de donnée n'a pas encore été effectuée, et elle initialise le compteur d_2 à *1*. Dans le cas de toutes autres combinaisons de signaux, ou si la condition du compteur d_1 n'est pas remplie, la machine passe dans l'état *5* et produit le signal *err*.

De l'état *2*, la machine peut rester jusqu'à 6 itérations avec comme valeur de *Frame Irdy*, 11. Ici, le nombre d'itération est relatif à la durée où le signal *Irdy* peut être désactivé durant une phase de donnée. Comme *Irdy* est déjà désactivé depuis deux cycles d'horloge, le temps restant est alors de 6 cycles d'horloge. De l'état 2 toujours, la

machine passe à l'état *3* lorsque la valeur des signaux *Frame* et *Irdy* est *10,* et à l'état *5* pour toutes autres combinaisons.

De l'état *3*, la machine fait le transfert de la dernière phase de donnée. Selon le référence [57], il n'y a pas de durée déterminée pour cette période. Pour représenter ce délai, nous avons décidé d'utiliser le symbole ∞. Ce délai peut être considéré comme un paramètre qui peut être initialisé lors de la configuration du bus et de son vérificateur.

De l'état *3* toujours, la machine passe à l'état *4* avec *11* pour entrées *Frame Irdy*. Ceci est représenté sur le diagramme feuille par le dernier échantillonnage. De l'état *3*, la machine passe à l'état *5* si les entrées *Frame Irdy* sont différentes de *11*.

La machine reste dans l'état *4* tant qu'elle n'aura pas reçu le signal *ack_in* et que la valeur des entrées est 11. Cette description a permis de faire un lien entre les machines à états et les diagrammes feuille.

Dans le modèle que nous avons développé, toutes les machines STFSMs sont basées sur une ossature qui est caractérisée par l'état initial, l'état final et l'état d'erreur, que nous appelons les états de composition. Le nombre d'états intermédiaires entre ces trois états dépend du diagramme que la machine représente. Sur la figure 15, les états de composition sont: l'état *0* qui est l'état initial, l'état *4* qui est l'état final et l'état *5* qui est l'état d'erreur. Ici, nous avons 3 états intermédiaires à savoir: *1, 2* et *3*. Dans la prochaine sous section, nous allons étudier la forme générale du STFSM que nous avons appelé "machine STFSM générique". Nous allons distinguer deux types de machines, soient la machine STFSM à 1 cycle d'horloge et la machine STFSM à plusieurs cycles d'horloge.

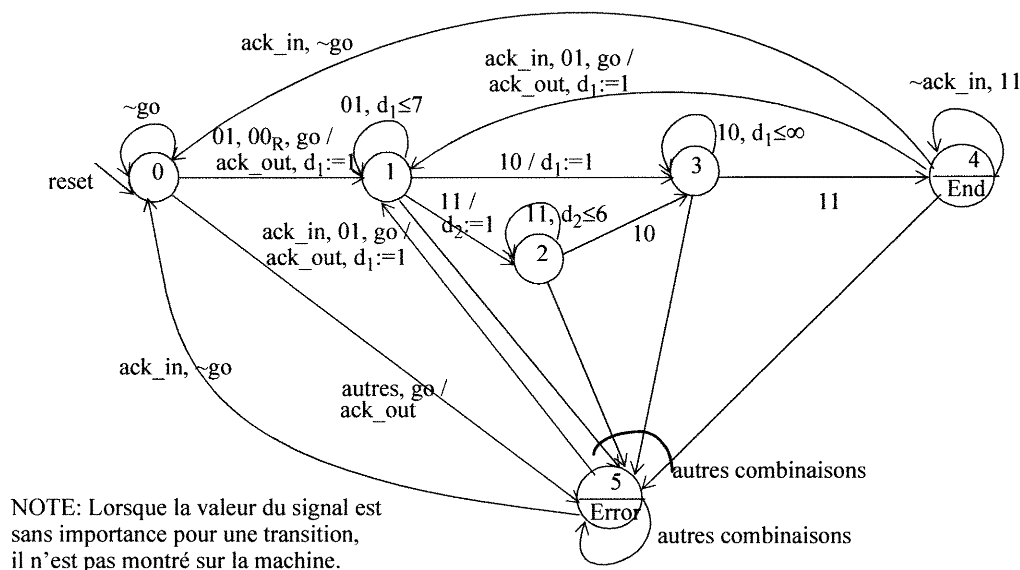


FIGURE 15. Exemple de STFSM: boîte Attente_et_Fin_de_Transaction de la Figure 10

3.4. La machine STFSM générique

La machine STFSM générique est une représentation abstraite de toutes les STFSM particulières. Elle est la forme programmable qui permet de générer les STFSM qui composent la machine à états observatrice. Remarquons que la machine générique a une forme générale caractérisée par son état initial, son état d'erreur et son état final, états que nous appelons *états de composition*. Ce sont ces états dans lesquels une STFSM communique avec une autre. Nous expliquerons dans la section 3.7. en quoi consiste la communication entre les machines STFSM.

Pour des raisons de la composition, voir section 3.5., nous avons distingué deux types de machines génériques. La machine à plusieurs cycles d'horloge et la machine à un cycle d'horloge. La machine générique à plusieurs cycles d'horloge, représentée par la figure 16, donne une forme générique pour les diagrammes feuilles qui durent pendant plusieurs cycles d'horloge (exemple: la boîte *Attente_et_Fin_de_Transition* de la figure 10). Quant à la machine générique à un cycle d'horloge, représentée par la figure

17, elle représente le diagramme feuille à un seul cycle d'horloge (exemple: la boîte *Active* de la figure 10). Cette machine ne contient que les états de composition nécessaires pour la composition avec d'autres machines.

Étant données les figures 16 et 17, les variables p_i sont utilisées pour modéliser les diagrammes particuliers de façon générique, indépendamment de la structure des états intermédiaires. Ces variables permettent de former un modèle générique non-déterministe d'un STFSM pour des fins de vérification du protocole de communication et elles n'apparaissent pas dans une machine qui modélise un diagramme spécifique comme dans le cas de la figure 15.

Sur les figures 16 et 17, les p_i représentent l'ensemble des valeurs des signaux, des compteurs et des opérations de compteurs sur l'arc d'une transition entre deux états. Si nous faisons une association avec la figure 15, p_i peut être associée à l'ensemble $\{01, 00_R, d{:=}1\}$ sur l'arc de transition entre l'état *0* et l'état *1*. Puisque notre but est de montrer comment les signaux de composition communiquent entre eux, ces signaux sont explicitement représentés dans la description des machines.

Sur les figures 16 et 17, *~x* dénote "*non x*" et *"p, q"* dénote "*p et q*".

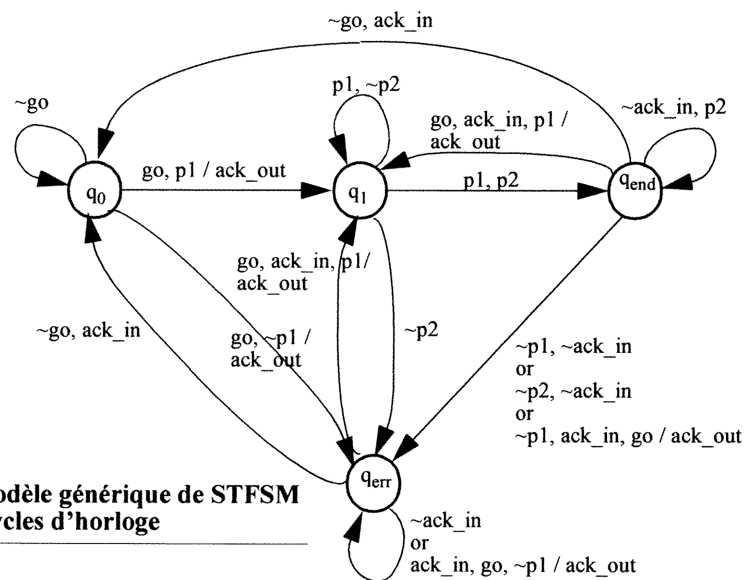


FIGURE 16. Modèle générique de STFSM à plusieurs cycles d'horloge

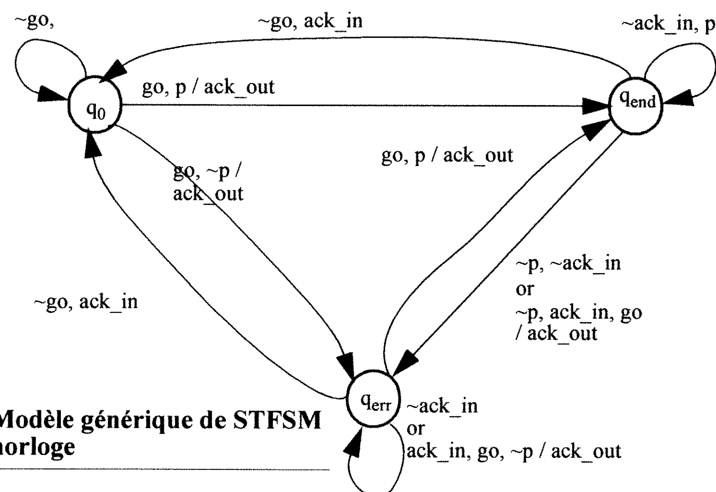


FIGURE 17. Modèle générique de STFSM à 1 cycle d'horloge

Comment fonctionne la machine générique à plusieurs cycles d'horloge?

Lorsque la machine à états observatrice est mise en marche, chaque machine STFSM est mise en marche et reste dans son état initial q_0. À chaque cycle d'horloge, la STFSM boucle sur l'état q_0 jusqu'à ce qu'elle reçoive le signal de démarage *go*. Lorqu'elle a reçu le signal *go*, si la proposition p_1 est vraie, alors elle passe dans l'état

q_1 sinon, elle passe dans l'état q_{err}. Dans les deux cas, le machine produit une sortie *ack_out*. Sur la figure 16, q_1 représente symboliquement tous les états intermédiaires de la machine. Tant que la proposition p_1 reste vraie, la machine reste dans les états intermédiaires jusqu'à ce que les propositions p_1 et p_2 soient vraies; la machine quitte alors q_1 et passe à l'état q_{end}. Dans le cas où les deux propositions ne sont pas simultanément vraies et que la machine quitte l'état q_1, elle passe à l'état q_{err}. Dans l'état q_{end}, tant que la machine ne reçoit pas le signal de composition *ack_in* et que la proposition p_2 est vraie, elle boucle sur le même état. Dans le cas où elle reçoit un signal *ack_in* et si, au même moment, elle reçoit le signal de composition *go* et que la proposition p_1 est vraie alors elle passe dans un état intermédiaire; sinon, si elle reçoit *ack_in* et ne reçoit pas *go,* elle passe alors à l'état initial q_0; sinon, elle passe à l'état q_{err}.

De l'état q_{err}, si la machine reçoit *go*, *ack_in* et que p_1 est vraie, elle passe de cet état au premier état intermédaire q_1 et produit *ack_out*; sinon, si elle reçoit *ack_in* et pas *go*, elle passe à l'état initial q_0; sinon, elle boucle sur l'état q_{err} à chaque cycle d'horloge. Lorsque la machine se retrouve dans n'importe quel état, si elle reçoit le signal *reset,* elle retourne alors à l'état initial q_0.

La machine à 1 cycle d'horloge ne possède pas d'états intermédaires aux états de composition. Dans ce cas, lorsque la machine reçoit *go* et que la proposition *p* est vraie, elle passe à l'état final q_{end} et produit une sortie *ack_out*. Les transitions de q_{end} vers q_0 et vers q_{err} restent les mêmes. De l'état q_{end}, si la machine reçoit *ack_in* et ne reçoit pas *go*, elle passe à l'état initial q_0; sinon, si elle reçoit *go* et *p* est vraie à ce moment là, la machine passe à l'état q_{end} et produit *ack_out*; sinon, elle reste dans son état q_{err}. Si nous faisons une association avec le diagramme feuille *Active* de figure 10, nous remarquons que la valeur des signaux *Frame Irdy* est de *00*. La machine STFSM qui représente le diagramme feuille *Active* ne vérifie que si la valeur des entrées est *00*.

Dès qu'elle reçoit le signal *go* dans son état initial, elle vérifie si les entrées *Frame Irdy* à ce même moment sont *00*. Si oui, la machine transite de l'état initial à l'état final.

Comme nous l'avons déjà dit plus haut, nous avons distingué deux types de machines STFSM génériques pour des raisons de composition. Dans les sections suivantes, nous présenterons des détails sur les raisons qui nous ont conduits à différencier deux types de machines STFSM génériques lorsque nous avons présenté la composition des STFSM en fonction des diagrammes à actions hiérarchiques (HAD).

3.5. Composition des STFSM en fonction des HAD

Les STFSM des diagrammes feuilles issus de la décomposition d'un protocole de bus doivent êtres composées en fonction des opérations hiérarchiques issues aussi de la décomposition. Cette composition se fait de façon ascendante, dans le sens contraire de la décomposition, c'est-à-dire de STFSM élémentaires vers des STFSM composées en faisant une combinaison logique des signaux de composition.

3.5.1. Les équations de coordination des STFSM

Sans perte de généralité, considérons la liste L de deux éléments M_1 et M_2 représentant des STFSM avec des signaux end_i, $error_i$, ack_out_i, en sortie et go_i, ack_in_i, $reset_i$ en entrée avec $i \in \{1,2\}$.

L'opération de cascade: $go_1 = go$; $go_2 = end_1$; $end = end_2$; $reset_1 = reset_2 = reset$; $error = error_1 \vee error_2$; $ack_in_1 = ack_out_2$; $ack_out = ack_out_1$; $ack_in_2 = ack_in$.

Lorsque les deux machines M_1 et M_2 sont composées en cascade en une machine M, l'ordre de départ go_1 de la machine M_1, qui est la première machine qui doit être exécutée, est le même que l'ordre de départ *go* de la composée M. Lorsque la machine

M_1 a complété son exécution et qu'elle arrive dans son état final, la sortie end_1 est produite. La sortie end_1 devient l'ordre de départ go_2 de la machine M_2. La machine M est complètement exécutée et produit *end* lorsque M_2 est complètement exécutée et produit end_2. Lorsque les deux machines M_1 et M_2 sont réinitialisées respectivement par $reset_1$ et $reset_2$, elles reçoivent chacune le signal de réinitialisation *reset* de la machine M. La machine M rentre dans son état d'erreur et produit la sortie *error* lorsque la machine M_1 rentre dans son état $error_1$ ou que M_2 rentre dans son état d'erreur $error_2$. La sortie *ack_out* de la machine M est produite lorsque la sortie ack_out_1 de la machine M_1 est produite. L'entrée ack_in_2 de la machine M_2 est identique à celle de la machine M. L'entrée ack_in_1 de la machine M_1 est produite lorsque la sortie ack_out_2 est produite.

Machine M

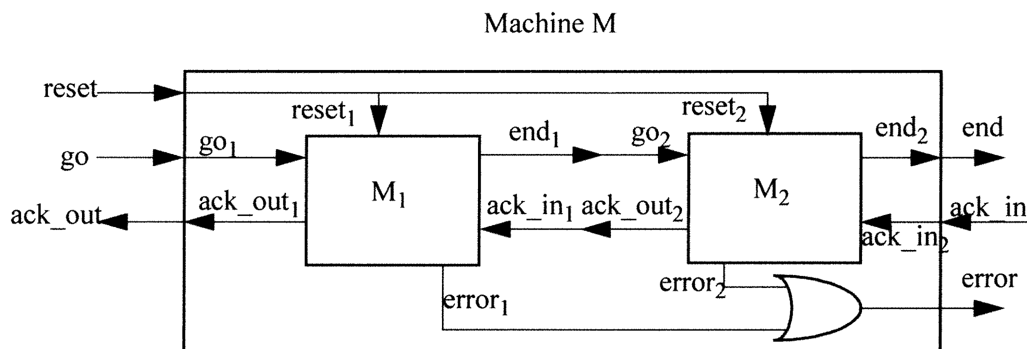


FIGURE 18. Description schématique de la composition en cascade de 2 machines

L'opération de boucle sur la machine M_1 est plusieurs composition en cascade de la même machine M_1. Pour pouvoir arrêter l'exécution de la boucle, nous avons introduit le prédicat *exit.*

$go_1 = (go \vee end_1) \wedge \neg exit;\ end = end_1 \wedge exit;\ reset_1 = reset;\ error = error_1;$

$ack_in_1 = (end_1 \wedge \neg exit) \vee (ack_in \wedge exit) \vee (error \wedge ack_in)$;

$ack_out = go \wedge ack_out_1$.

Le prédicat *exit* est un prédicat sur le nombre de fois qu'un diagramme feuille doit être exécuté dans une boucle. Le prédicat *exit* peut être considéré sur un compteur. Lorsque ce compteur atteint le nombre de fois que le diagramme feuille doit être executé dans la boucle, *exit* est activé.

L'ordre d'exécution go_1 de la machine M_1 est donné chaque fois que la machine reçoit un ordre *go* externe et que *exit* n'est pas vrai ou chaque fois que la machine M_1 arrive dans son état final, produit end_1 et que *exit* n'est pas vrai. L'opération de boucle est terminée lorsque *exit* est vrai et que la machine M_1 est dans son état final. L'opération de boucle est finie lorsque la machine M_1 rentre dans son état d'erreur et que *exit* est vrai. L'opération de boucle produit *ack_out* lorsque l'ordre de l'opération vient de *go* et que ack_out_1 de la machine M_1 est produit. L'exécution de la boucle est erronnée et produit *error* lorsque M_1 tombe dans son état d'erreur et produit $error_1$. La machine M_1 reçoit ack_in_1 si M_1 est dans son état final et que *exit* n'est pas vrai (ceci permet à la machine M_1 d'exécuter la prochaine boucle), ou si elle reçoit *ack_in* de la machine *M* de l'opération de la boucle et que *exit* est vrai au même moment, ou si la machine *M* rentre dans son état d'erreur et qu'elle reçoit *ack_in*.

C'est dans l'opération de la boucle qu'intervient la notion de machine générique à un seul cycle d'horloge. Ici, la machine ne dispose que de trois états, à savoir l'état *initial*, l'état *final* et l'état d'*erreur* (figure 17). Lorsque la machine à deux ou plusieurs cycles d'horloge est dans son état *final* et qu'elle reçoit un signal *go*, elle doit normalement effectuer la première transition valide ou d'erreur conformément à la valeur des entrées reçues (exemple de la transition de q_{end} à q_1 ou à q_{err} de la figure 16) pour garder la composition. Dans le cas d'une machine à un seul cycle d'horloge, un problème se pose quant à cette transition. Lorsque la machine reçoit un signal *go* et

qu'elle se trouve dans son état final, si elle passe dans son état initial, il lui faut un autre signal *go* pour passer de l'état initial à l'état final. Cela prend au moins un cycle d'horloge supplémentaire et engendre la perte de données valides de signaux. Si elle reste dans son état final et si elle est concatenée à une autre machine, elle va toujours générer des signaux *end* qui vont générer des *go* incorrects pour la machine suivante. Pour remédier à ce problème, nous avons créé une structure particulière qui consiste à dupliquer la machine en deux machines identiques qui peuvent être considérées comme des clones de la première machine. Elles sont composées par l'opérateur de choix. Les machines ainsi dupliquées sont exécutées à tour de rôle à chaque cycle d'horloge, lors de l'exécution de la boucle. Cette exécution est basée sur une variable booléenne V qui, à chaque cycle d'horloge, change de valeur. Ceci permet d'exécuter alternativement chacune des machines dupliquées sur la même action mais à des temps différents. Les opérations de composition sont une combinaison des opérations de boucle, que nous venons de décrire, et des opérations de choix dont nous allons parler plus tard dans la suite de cette section. La seule différence entre la composition de choix de deux machines à un cycle d'horloge dans une boucle et de deux machines à plusieurs cycles d'horloge est que les deux branches du choix sont complémentaires, à savoir: $go_1 = go \wedge v$ pour la première branche et partout où go_1 est utilisé dans la machine M_1 de la première branche, et $go_2 = go \wedge \neg v$ pour la seconde branche et pour tous les go_2 de la machine M_2 du choix (voir figure 19). Les autres éléments de la composition de choix restent identiques à l'opération de choix.

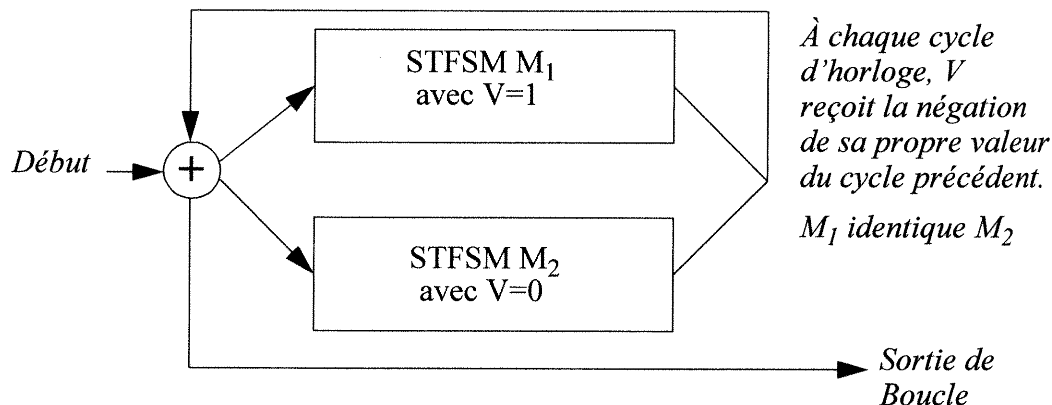


FIGURE 19. Opération de boucle sur un STFSM à 1 cycle d'horloge

L'opération parallèle: $go_1 = go_2 = go$; $end = end_1 \wedge end_2$;

$reset_1 = reset_2 = reset$; $error = error_1 \vee error_2$; $ack_in_1 = ack_in_2 = ack_in$;

$ack_out = ack_out_1 \wedge ack_out_2$.

Considérons une machine M qui est la composée parallèle des deux machines M_1 et M_2. Les deux machines concurrentes reçoivent en même temps l'ordre de départ go. La machine M est correctement executée lorsque M_1 et M_2 sont dans leur état final et que end_1 et end_2 sont respectivement produits par les deux machines. Lorsque la machine M est réinitialisée, les machines M_1 et M_2 le sont aussi. L'exécution de la machine M est erronée si la machine M_1 rentre dans son état d'erreur et produit $error_1$ ou la machine M_2 rentre dans son état d'erreur et produit $error_2$. La machine M produit ack_out lorsque ack_out_1 de la machine M_1 et ack_out_2 de la machine M_2 sont produits. La machine M produit ack_out lorsque ack_in_1 et ack_in_2 des machines M_1 et M_2 sont produits.

L'opération de choix retardé: $go_1 = go_2 = go$; $end = end_1 \vee end_2$;

$error = error_1 \wedge error_2$; $reset_1 = reset_2 = reset$; $ack_out = ack_out_1 \vee ack_out_2$;

$ack_in_1 = ack_in_2 = ack_in$.

Considérons une machine M qui est la composée choix retardé de deux machines M_1 et M_2. Les ordres d'exécution go_1 et go_2 des machines M_1 et M_2 respectivement sont donnés lorsque *go* de la machine M est donné. M est complètement executée lorsque M_1 ou M_2 arrive dans son état final et produit respectivement end_1 ou end_2. La réinitialisation de M par *reset* entraîne la réinitialisation de M_1 et de M_2 par $reset_1$ et $reset_2$. La machine M rentre dans un état d'erreur et produit *error* lorsque M_1 et M_2 rentrent toutes les deux dans leur état d'erreur et produisent $error_1$ et $error_2$. La machine M produit *ack_out* lorsque ack_out_1 de M_1 ou ack_out_2 de M_2 est produit. Lorsque M reçoit son *ack_in*, ack_in_1 et ack_in_2 sont aussi reçus respectivement par M_1 et M_2.

L'ensemble des machines STFSM composées par les opérations de composition représente la machine à états observatrice.

3.6. Conclusion

Dans ce chapitre, nous avons décrit l'approche pour la synthèse de la machine observatrice de bus, afin de déterminer si oui ou non le comportement d'un bus est conforme à sa spécification. La machine observatrice est générée à partir des spécifications HAAD du protocole du bus et est implémentée en RTL synthétisable, c'est-à-dire en Verilog ou en VHDL, comme un réseau de machines à états. Dans le prochain chapitre, nous allons décrire comment vérifier formellement le protocole de composition et sa réalisation par les machines STFSM et les opérateurs de composition.

CHAPITRE 4 *Validité de la méthode de synthèse des EFSM*

Many persons who are not conversant with mathematical studies imagine that because the business of [Babbage's Analytical Engine] is to give its results in numerical notation, the nature of its processes must consequently be arithmetical and numercal, rather than algebrical and analytical. This is an error. The engine can arrange and combine its numeral quantities exactly as if they were letters or any other general symbols; and in fact it might bring out its results in algebrical notation, were provisions made accordingly.
Augusta ADA, Countess of Lovelace (1844)

4.1. Introduction

Ce chapitre porte sur l'utilisation de la méthode de démonstration de modèle pour prouver l'exactitude de la méthode proposée pour la synthèse des machines observatrices à partir des spécifications HAAD. D'abord, nous donnerons un bref aperçu des méthodes formelles et ensuite montrerons l'exactitude du modèle STFSM générique de la machine observatrice et de la composition de ces machines.

4.2. Démonstration de modèle [27] [18] [55] [16] [33]

Le "Model Checking" ou démonstration de modèle nous permet de prouver que les propriétés spécifiées d'un système sont respectées par la réalisation de ce dernier sous forme d'un FSM. Autrement dit, la démonstration de modèle est faite comme une recherche exhaustive de l'espace d'états qui est garantie d'être faite dans un temps fini car le système est fini.

Deux approches sont utilisées en démonstration de modèle (model checking). La première, le "Temporal Model Checking", qui est une technique développée indépendamment dans les années 1980 par Clarke et Emerson d'une part [12] [14] et par Queille et Sifakis [11] d'autre part, est une approche dans laquelle les spécifications sont exprimées en logique temporelle (*TL*) et les objets à vérifier sont modélisés par des systèmes à transitions finis. Des procédures de recherche sont utilisées pour démontrer si le système de transition est un modèle de la spécification. Dans ce cas, le système, aussi représenté par un automate, est comparé à la spécification pour déterminer si son comportement est conforme à celui de la spécification.

La seconde approche appelée "Symbolic Model Checking" sur laquelle a travaillé Kenneth L. McMillan [43] [33], est une technique développée pour la recherche de l'espace d'états basée sur une représentation implicite des états en utilisant les diagrammes de décisions binaires ordonnés réduits (ROBDD), et qui permet de réduire le problème d'explosion d'états en ne représentant pas explicitement les états du modèle étudié. Ce graphe de décision permet de produire des fonctions booléennes sous forme compacte et canonique. Pour appliquer cette technique à la vérification temporelle, McMillan observe que si l'état d'un système est représenté par un vecteur de variables booléennes, alors un ensemble d'états peut être représenté par une fonction booléenne qui retourne *vrai* pour tous les états de l'ensemble. De même, il observe qu'une relation xRy entre les états peut être représentée par une fonction booléenne de deux ensembles de variables, un ensemble représentant x et l'autre représentant y. De là, l'algorithme de démonstration de modèle peut être développé par les ROBDDs pour représenter les ensembles et les relations. En utilisant la démonstration de modèle symbolique, certains systèmes regulièrement structurés avec des nombres d'états très élevés peuvent être automatiquement vérifiés.

Différentes notions de conformance ont été explorées, à savoir: l'inclusion de langages, les ordonnancements des états par affinage et les équivalences observées. Dans la référence [13], Vardi et Wolper ont montré d'autre part comment le problème de la

démonstration de modèle par la logique temporelle peut être remodelé en terme d'inclusion de langages omega.

4.2.1. Démonstration de modèle

Un système à états finis peut être représenté par un graphe de transition à états étiquetés où les étiquettes d'un état sont les valeurs des propositions atomiques (par exemple: les valeurs des bascules et des signaux). Les propriétés concernant le système sont exprimées comme des formules en logique temporelle pour lesquelles les systèmes à transition d'états doivent être un modèle. La démonstration de modèle consiste à parcourir le graphe de transitions du système et à vérifier si le système est le modèle de la propriété.

4.2.2. Computation Tree Logic (CTL) [49] [14]

La logique temporelle exprime l'ordre des événements dans le temps par une spécification des propriétés telle que "*la propriété p aura éventuellement lieu*". Le CTL est une variante de la logique temporelle et de l'arborescence temporelle. Les formules en CTL font référence à l'arbre de calcul dérivé du modèle. Les états du système sont des valeurs enregistrées dans des bascules. Chaque formule de la logique est soit vraie ou fausse pour un état donné. L'exactitude de la formule est évaluée à partir des valeurs de vérité de ces formules de manière récursive jusqu'à ce qu'une proposition atomique soit vraie ou fausse pour un état donné. Une formule est satisfaite pour un système si elle est vraie pour tous les états initiaux du système.

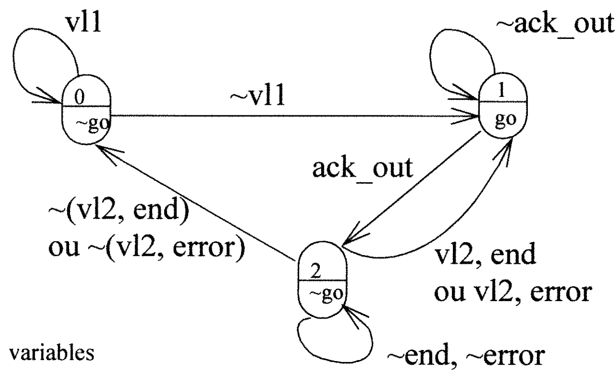
Nous allons présenter dans les sections qui suivent la vérification de la méthode que nous avons proposée pour la synthèse des machines à états observatrices à partir des spécifications HAAD, en utilisant VIS, qui est un outil de démonstration de modèle, et le CTL.

4.3. Validité des STFSM génériques

La vérification de la méthode pour la synthèse des machines à états observatrices se fait en deux étapes, à savoir: la validité du modèle générique du STFSM et celle de la composition des STFSM selon la spécificité de chaque opérateur.

4.3.1. Vérification du modèle STFSM générique

La vérification du modèle STFSM générique a nécessité la création d'un environnement qui commande les signaux d'entrée de la machine générique conformément aux équations de composition. Pour cela, nous avons introduit deux automates. Le premier, figure 20, appelé *générateur du signal go* (Generator) sert à générer le signal *go* qui donne l'ordre de départ d'exécution du modèle générique, reçoit et traite le signal *ack_out* lorsque le modèle générique le produit. Le second, figure 21, est appelé *réflecteur* (Reflector) car il permet de générer un signal *ack_in* suite à la réception du signal *end* ou *error* produit par le modèle générique. Le générateur, le réflecteur et le modèle générique sont connectés comme le montre la figure 22. On vérifie que toutes les transitions spécifiées existent et qu'il n'y a pas d'autres transitions pouvant corrompre le fonctionnement du modèle générique. Par ailleurs, nous avons vérifié que le modèle ne reste pas bloqué dans une boucle infinie, dans un état quelconque (voir Annexe).



Note:
vl1 et vl2 sont des variables booléennes non déterministe qui permettent au générateur de générer de façon non déterministe le signal go pour le modèle générique.

FIGURE 20. Machine à états pour générer le signal go

.

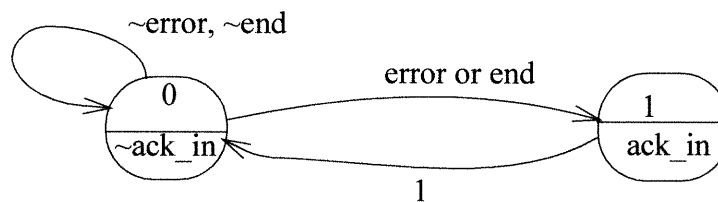


FIGURE 21. Machine à états réflecteur du signal ack_in

Comment fonctionnent le générateur et le réflecteur?

Le *générateur* est une machine à états définie comme suit: *{0, 1, 2}* l'ensemble des états; *{vl1, vl2, ack_out, error, end}* l'alphabet des entrées; *0* l'état initial; *1* l'état final; *go* la sortie vers le modèle générique; l'ensemble des étiquettes sur les arcs constitue la fonction de transition.

Les variables *vl1* et *vl2* sont des variables booléennes non déterministes afin que *go* soit généré de façon aléatoire. Au début de son exécution, la machine *générateur* est dans son état initial *0*. De l'état *0*, tant que la variable *vl1* est vraie, *générateur* boucle. Le signal *go* reste toujours inactif. Si la valeur de *vl1* devient fausse, *générateur* passe

alors de l'état *0* à l'état *1*. De l'état *1*, *générateur* produit *go* et attend *ack_out* de la machine STFSM générique. Tant qu'elle n'a pas reçu *ack_out*, elle va boucler sur l'état *1* et produire *go* jusqu'à ce que *ack_out* soit produit. Si elle reçoit *ack_out*, elle passe de l'état *1* à l'état *2*.

De l'état *2*, si *générateur* ne reçoit pas *end* ou *error* de la part de la machine STFSM générique, il boucle sur *2*. Si il reçoit *end* ou *error* et que la variable *vl2* est vraie au même moment, il retourne à l'état *1*; sinon il va à l'état *0*.

Le fonctionnement du générateur permet d'activer le signal *go* uniquement lorsque le modèle générique est dans un état de composition *(initial, final, erreur)* afin de relancer le modèle générique durant toute la durée de la période de simulation. Les variables *vl1* et *vl2* sont ajoutées afin que l'activation du signal *go* ne soit pas systématique à chaque fois que le modèle se retrouve dans un état de composition. Nous avons proposé cette solution pour la vérification du modèle afin de montrer tous les aspects fonctionnels de celui-ci, c'est-à-dire que le modèle peut boucler dans l'état de composition dans lequel il est aussi longtemps qu'il n'aura pas reçu le signal *go*.

La machine *réflecteur* (reflector) permet de générer le signal *ack_in* qui est supposé être produit lors d'une opération de composition *concatenation*. Ce signal permet à la machine générique de quitter son état d'erreur ou son état final et de retourner dans son état initial ou dans un état intermédiaire. La machine *réflecteur* a deux états *{0, 1}*; son état initial est *0*; son ensemble des entrées est *{error, end}*; son ensemble de sortie est *{ack_in};* son état final est *1*; l'ensemble des étiquettes sur les arcs constitue la fonction de transition.

Au début de l'exécution de la machine *réflecteur*, l'état courant est *0* et *ack_in* est initialisé à 0. Tant que *réflecteur* n'a pas reçu *error* ou *end*, elle boucle sur son état *0*. Si elle reçoit *error* ou *end*, elle passe alors de l'état *0* à l'état *1*. De l'état *1*, elle produit *ack_in* et retourne à l'état *0* à la prochaine transition. C'est la raison pour laquelle l'arc de transition de l'état *0* à l'état *1* est étiqueté 1.

Le but du réflecteur est d'activer le signal *ack_in* à chaque fois que le modèle générique se retrouve dans son état final ou d'erreur. Il permet d'activer pendant un cycle d'horloge le signal *ack_in* qui est un signal en entrée pour le modèle générique. Une fois le signal *ack_in* généré, au prochain cycle d'horloge le reflecteur retoune dans son état initial et désactive le signal *ack_in*. Ce comportement du réflecteur produit pour le modèle générique le signal *ack_in* comme l'aurait fait un autre modèle qui est composé avec le premier, c'est-à-dire que le deuxième modèle produit *ack_out* qui devient *ack_in* pour le premier pendant un cycle d'horloge.

Comme nous l'avons déjà décrit dans le chapitre précédent, une machine STFSM élémentaire et une machine composée ont les mêmes interfaces si nous considérons les signaux de compositions. De ce fait, le modèle générique de la machine observatrice peut être une STFSM élémentaire qui représente la description d'une machine à états, ou la composée de deux STFSM élémentaires, de deux STFSM composées ou d'une STFSM élémentaire et d'une STFSM composée. Aussi, nous avons indiqué dans le chapitre précédent qu'une STFSM générique à un cycle d'horloge est utilisée de la même façon que n'importe quelle STFSM élémentaire dans toutes les opérations de composition, sauf l'opération de boucle.

4.3.2. Prototype de test pour le modèle énérique du STFSM en Verilog

La figure 22 et sa desciption en Verilog représenté dans l'annexe sont considérées comme le prototype de test du modèle STFSM générique et de sa composition avec les machines *générateur* et *réflecteur*. Ce prototype montre l'interconnection des signaux de composition et permet au STFSM d'être testé conformément à sa description. Il fait par ailleurs abstraction du niveau de complexité des systèmes qui peuvent être vérifiés par la machine à états observatrice. Aussi, il porte sur l'encapsulation des opérations de composition et sur le lien entre les opérateurs et la structure des STFSM.

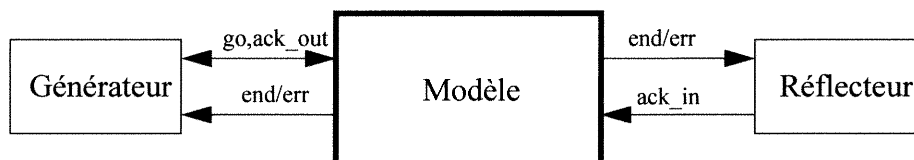


FIGURE 22. Schéma de prototype de test pour le modèle générique du STFSM

La représentation en Verilog de la machine observatrice est constituée de plusieurs modules encapsulés les uns dans les autres. Chaque module représente une machine STFSM élémentaire ou une opération de composition qui est elle-même une encapsulation de plusieurs STFSM composées par des opérations de composition. Les modules dont la forme générale est:

module nom("signaux du système comme entrée", "signaux de composition")

sont généralement déclarés avec deux groupes de paramètres. Le premier est constitué des signaux du système étudié qui sont des entrées de la machine observatrice. Le second est l'ensemble des signaux de composition *go, reset, ack_in, ack_out, end* et *error* qui sont déclarés pour tous les modules. Les signaux *go, reset, ack_in* sont déclarés en entrée et les signaux *end, error et ack_out* sont déclarés en sortie.

Chaque description en Verilog d'une machine à états observatrice est un module principal de la forme:

module main("signaux du système à l'entrée", "signal témoin").

Les signaux d'entrée contiennent l'horloge du système, la machine observatrice étant synchrone avec le système qu'elle observe.

Le *signal témoin (flag)* est celui qui rapporte l'état de la vérification du protocole du bus. Il est la composition des signaux d'erreur des modules qui dépendent de lui hiérarchiquement. Il est vrai s'il y a une erreur et faux sinon.

4.3.3. Spécification et vérification des propriétés pour le modèle générique du STFSM

Les propriétés proposées en formule CTL sont essentiellement liées aux états de composition (*initial, final* et *erreur*), et aux signaux de communication. Par exemple, lorsqu'une machine est dans un état de composition et qu'elle reçoit le signal *go*, elle produit un signal *ack_out* comme le montre la formule suivante:
$AG(Etat = q_i * go \rightarrow ack_out * EX(Etat = q_j))$ où q_i peut-être associé à l'état *0* et q_j à l'état *1* ou *4* dans la figure 15. De même, nous devons vérifier que lorsqu'une machine est dans l'état de composition d'erreur ou final, elle doit produire respectivement le signal *error* ou *end* comme le montre par exemple la formule suivante:
$AG(Etat = q_i \rightarrow EX(Etat = q_j * error))$ où q_i peut-être associé à l'état *1* et q_j à l'état *4*. Toutes les propriétés qui permettent de vérifier l'intégrité des signaux de communication de la machine générique sont décrites en CTL dans l'annexe.

Nous avons utilisé VIS [49] comme outil de vérification et vérifié avec succès toutes les propriétés des modèles STFSM génériques.

4.3.4. Vérification des compositions des STFSM

La composition hiérarchique de deux ou plusieurs STFSM est aussi un STFSM qui peut être considérée comme une boîte noire qui reçoit en entrée les ports et signaux de composition et qui par son interface respecte les équations de composition décrites dans le chapitre 3. Les signaux de composition de la machine STFSM des différentes opérations sont des fonctions logiques des signaux de composition entre des STFSM qui sont composées par cette opération.

Les propriétés sur les compositions de STFSM vérifient que le comportement des interconnexions conserve les caractéristiques de la machine STFSM générique (voir Annexe) et que les équations de composition de chaque opérateur sont respectées comme il est spécifié dans le chapitre 3. Si nous prenons l'exemple de l'opération de cascade entre deux machines M_1 et M_2, la formule à vérifier serait la suivante:

$AG((M_1.Etat = q_0 * M_2.Etat = q_0 * go) \rightarrow ack_out *$

$AX((M_1.Etat = q_1 * M_2.Etat = q_0) + (M_1.Etat = q_{err} * M_2.Etat = q_0 * error)))$.

De façon analogue, nous pouvons déduire une propriété pour vérifier que lorsque M_1 entre dans son état final, M_2 reçoit automatiquement son signal *go* et passe de l'état q_0 à l'état q_1 ou à l'état q_{err} (état d'erreur) avec la sortie *error.* Dans le cas des autres opérateurs hiérarchiques, les mêmes genres de propriétés sont vérifiés. Tous les résultats sont consignés dans l'Annexe.

La préservation des équations de composition permet de prouver que l'interface d'interconnection des STFSM en machine observatrice basée sur les équations du chapitre 3 est le même, quel que soit le type d'opérateur utilisé, c'est-à-dire que l'interface d'interconnection comporte les signaux de composition *go*, *ack_in*, *reset* en entrée et *end*, *error*, *ack_out* en sortie. Un exemple de conservation de propriétés est de prouver qu'à chaque fois qu'il y a le signal *go* et les machines composées sont dans les états appropriés, le signal *ack_out* est obligatoirement produit. Si nous prenons par exemple deux machines M_1 et M_2 composées en une machine *M*, nous pouvons définir les propriétés suivantes:

$AG((M_1.state = q_0) * (M_2.state = q_0) * go \rightarrow ack_out);$

$AG(error * go \rightarrow ack_out); AG(end * go \rightarrow ack_out);$

$AG((M_1.state = q_0) * (M_2.state = q_0) * go \rightarrow AX(A(!ack_out\ U(error + end))));$

$AG(error * go \rightarrow AX(A(!ack_out\ U(error + end))));$

$AG(end * go * ack_in \rightarrow AX(A(!ack_out\ U(error + end))));$

$AG(error * go * ack_in \rightarrow AX(A(!ack_out\ U(error + end))));$

Ces propriétés permettent par ailleurs de vérifier que si une machine est dans l'état d'erreur ou final et qu'elle reçoit les signaux *go* et *ack_in*, alors elle doit se comporter exactement comme lorsqu'elle reçoit le signal *go* dans son état initial.

Pour s'assurer que les propriétés définies sur les machines composées couvrent le comportement désiré pour la synthèse de la machine observatrice, nous avons défini des propriétés pour chaque équation des opérations de composition du chapitre 3. Ainsi, nous avons vérifié par exemple que le signal *ack_out* d'une machine M composée de deux machines M_1 et M_2 par l'opération cascade est le signal ack_out_1 de la machine M_1, alors que le signal *ack_out* d'une machine M composée de deux machines M_1 et M_2 par l'opération parallèle est le signal ack_out_1 de la machine M_1 et ack_out_2 de la machine M_2.

Vu que l'interface d'une machine STFSM générique et d'une machine STFSM composée par une opération est la même, les propriétés que nous avons vérifiées pour les machines composées ont été appliquées à la machine générique avec succès et *vice versa*. Ceci permet de conclure que le modèle que nous avons proposé pour la synthèse de la machine observatrice est conforme à nos attentes.

4.4. Conclusion

Pour prouver la méthode que nous avons proposée pour la synthèse des machines à états observatrices à partir des spécifications HAAD, nous avons utilisé VIS qui est un vérificateur de modèle [28]. Nous avons présenté d'abord la vérification du modèle générique des STFSM, et ensuite la vérification des différentes compositions hiérarchiques. Dans les deux cas, nous avons défini des propriétés en CTL que nous avons vérifiées positivement. Nous avons aussi montré la conservation des propriétés de l'interface entre la machine générique et les machines composées selon les opérations. Ceci nous a donné la confirmation que la méthode proposée pour la composition des machines STFSM satisfait nos attentes. Dans le chapitre suivant, nous présenterons l'application de la synthèse de la machine observatrice aux opérations de lecture et d'écriture du bus PCI en nous basant sur la description de la référence [57].

CHAPITRE 5

Étude du bus PCI: synthèse de EFSM observatrice et les résultats expérimentaux

Idealy, people want to hook together lots of peripherals -and many different kinds of peripherals- with as little muss and fuss as posible.
David James, Apple research scientist, Byte (1994)

5.1. Introduction [57]

L'architecture du bus PCI (Peripheral Component Interconnect) permet de répondre aux besoins du CPU en terme de transfert de données avec les périphériques et les mémoires.

Nous avons choisi le bus PCI comme exemple de la synthèse des machines observatrices à partir des spécifications HAAD parce qu'il est devenu aujourd'hui un standard dans l'industrie, ce qui fait que la documentation est plus abondante [57]. D'autre part, la spécification de PCI est assez hiérarchisée. Chaque fonction est décrite sur un plan fonctionnel et temporel. Ceci constitue une source assez complète d'information et facilite la traduction de la spécification des fonctions en modèle HAAD. Dans la suite de ce chapitre, nous donnerons une brève description du bus PCI, de ses signaux de contrôle et de l'horloge, des contraintes de temps entre les signaux de contrôle, les types d'interruption des agents sources et destinations. Nous décrirons aussi les tran-

sactions de lecture et d'écriture. Nous ne parlons pas de la vérification des adresses et des données dans notre cas parce qu'il existe des méthodes plus appropriées pour les vérifier. Nous parlerons plus des signaux de contrôle et du protocole du bus.

5.2. Opérations sur le bus PCI

Les opérations de transfert sur le PCI sont appelées "*Burst Transfer*". Un "*Burst Transfer*" est constitué d'une phase d'adresse suivie de plusieurs phases de données. Tous les transferts se font entre deux types d'agents: le *Master* ou l'*Initiator (Source)* et un ou plusieurs *Targets (Destinations)*. La source dans une transaction peut devenir la destination dans une autre. Il existe deux groupes d'agents et deux types de bus: les bus et les agents à 32 bits et ceux à 64 bits. Le PCI est un bus intelligent. Avant toute transaction, l'agent source doit recevoir une autorisation de l'arbitre du bus. La connexion entre les agents et l'arbitre du bus est une connexion point à point qui se fait par l'intermédiaire de deux signaux. Le signal *REQ* de l'agent vers l'arbitre, qui permet de faire une demande d'acquisition du bus, et le second *ACK*, de l'arbitre vers l'agent qui est la réponse à la demande *REQ*.

Lors d'une transaction sur le PCI, la source fait une demande auprès de l'arbitre du bus. Si la demande est acceptée, la source envoie sur le bus l'adresse et le type de transaction à accomplir, suivie d'une ou de plusieurs phases de données. L'adresse et le type de transaction, une fois reconnus par la/les destination(s), sont mémorisés par ces dernières dans des registres jusqu'à la fin de la transaction. Toutes les actions sur le bus sont synchrones par rapport au signal *PCI CLK* à une fréquence de 0 à 33 MHz pour la version 2.0 et de 0 à 66 MHz pour la version 2.1. [57]. Les spécifications du bus PCI indiquent que les agents connectés doivent échantillonner leur signaux au front montant de l'horloge. Tous les signaux de tous les agents sont synchrones à l'horloge, sauf le signal *RESET (RST)* qui est asynchrone et dont l'activation est prise en compte par tous les agents à tout moment. Lors d'une transaction, la phase d'adressage dure un cycle d'horloge et elle est suivie d'une ou de plusieurs phases de données

d'un cycle d'horloge chacune. Ce principe de transaction change lorsque deux agents de types différents communiquent. Si une source de 64 bits communique avec une destination de 32 bits, les phases d'adressage et de données se dédoublent. Chaque transaction est toujours suivie d'un état de repos du bus (*Idle State*), qui peut précéder lui-même une transaction ou non. Lorsqu'une destination est sélectionnée lors d'une transaction, un signal appelé *DEVESEL* (*Device select*) est activé et reste actif jusqu'à ce que la destination transfère la dernière phase de données. La durée d'une transaction est définie par la source, qui active le signal *FRAME* qui indique cette durée. Celle-ci peut être prédéfinie et consignée dans un registre ou peut être calculée par un logiciel. Le signal *FRAME* est activé lors de la phase d'adresse et reste actif jusqu'à ce que la source soit prête pour le transfert de la dernière phase de données. Chaque fois qu'une source est active lors d'une transaction, elle active son signal *IRDY.* De même, toutes les fois qu'une destination est active, elle active son signal *TRDY.* Dans les deux cas, lors d'une transaction, les signaux *IRDY* et *TRDY* peuvent être désactivés. Nous donnerons plus de détails sur les signaux de contrôle du bus PCI dans la suite de ce chapitre.

Une transaction est dite terminée ou complétée si toutes les phases de données initiées par la source sont toutes reçues par la destination.

5.3. Les groupes des signaux fonctionnels du bus PCI

Dans cette section, nous présentons les groupes fonctionnels des signaux du bus. Les figures 23 et 24 contiennent les illustrations des signaux qui sont importants et ceux qui sont optionnels pour les agents sources et destinations du bus. Sur le bus PCI, comme nous l'avons déjà annoncé, lors d'une transaction, un agent peut être une source et devenir une destination pour la prochaine transaction. À cet effet, les agents peuvent avoir tous les signaux qui caractérisent les sources comme ceux qui caractérisent les destinations. Un agent doit agir au minimum comme une destination avec une configuration minimale de lecture et écriture.

Les signaux de l'interface sont regroupés en groupes fonctionnels. Bien que nous ayons plusieurs groupes de signaux comme l'indiquent les deux figures suivantes, nous nous intéressons seulement aux signaux de contrôle et à l'horloge CLK.

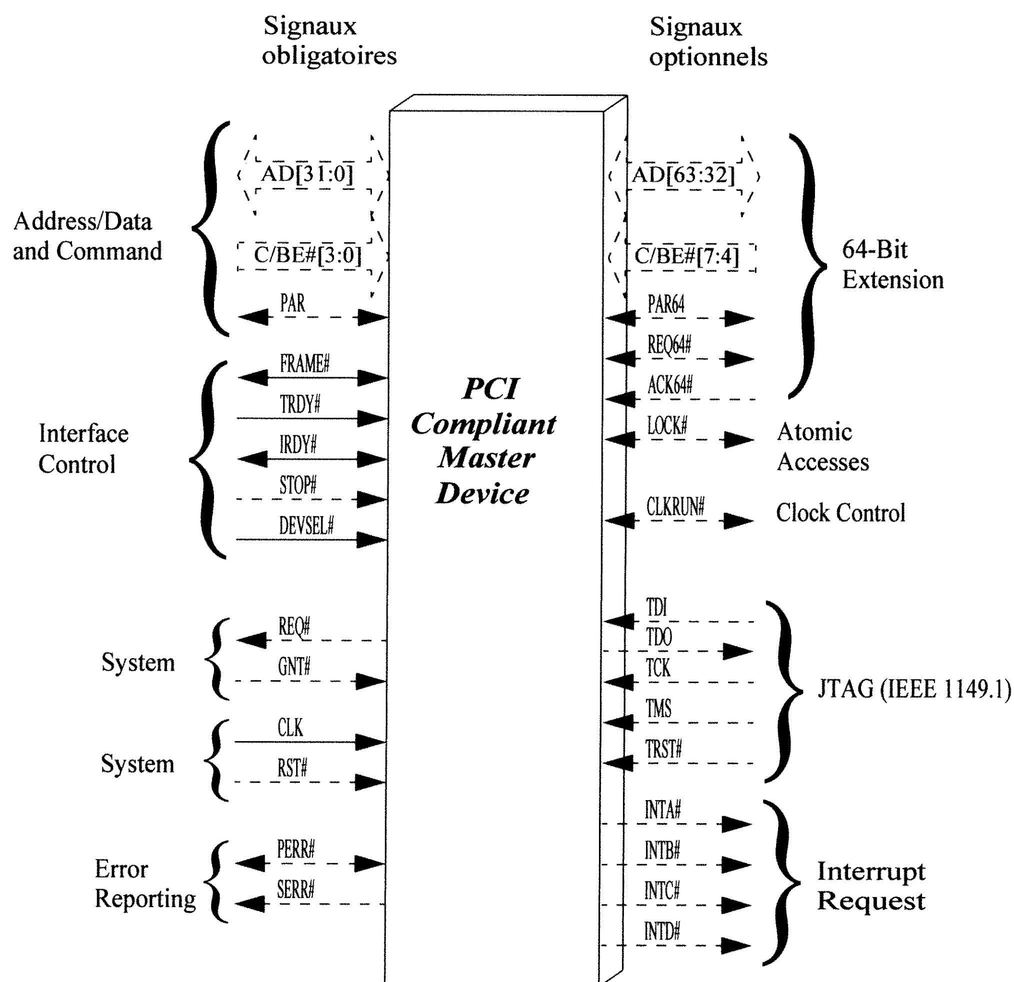


FIGURE 23. Représentation schématique d'un agent source du bus PCI

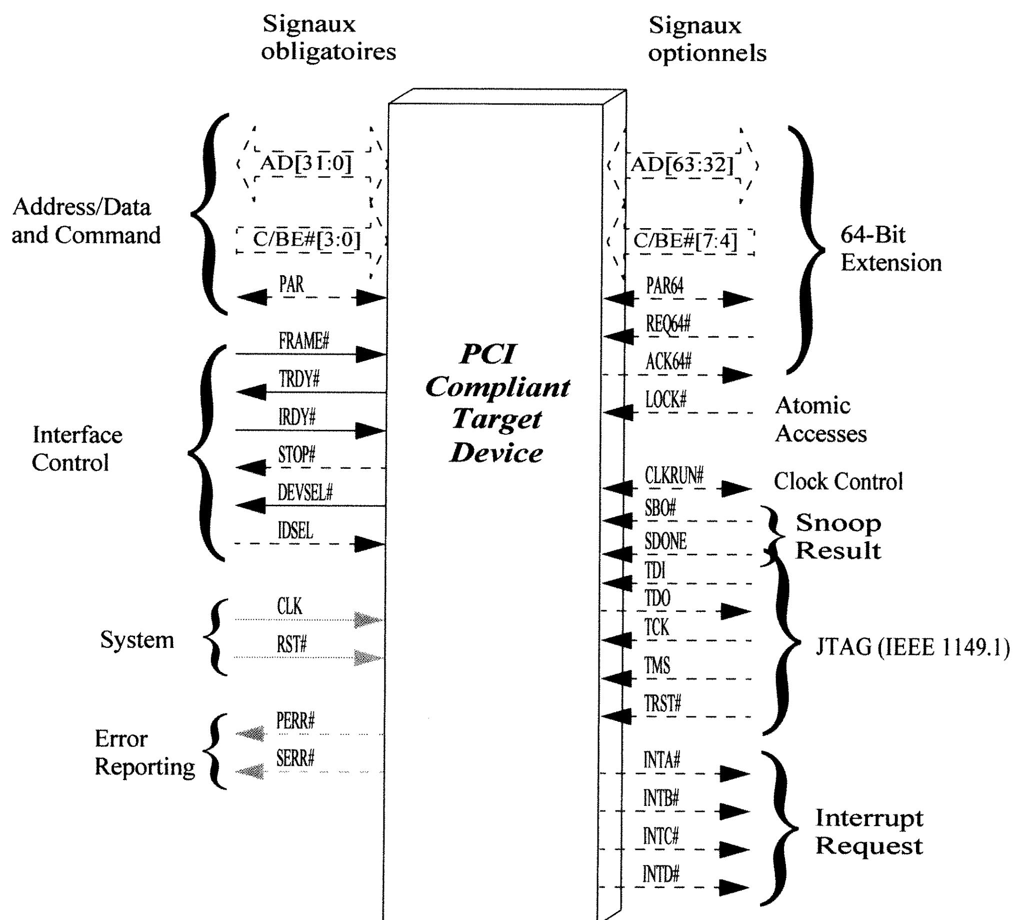


FIGURE 24. Représentation schématique d'un agent destination du bus PCI

5.3.1. Le signal d'horloge (CLK)

Le signal *CLK* est un signal en entrée pour tous les agents connectés au bus PCI. Il est chargé de cadencer toutes les transactions sur le bus y compris l'arbitrage. Tous les agents du bus PCI échantillonnent au front montant de l'horloge *CLK*.

5.3.2. Les signaux de contrôle

Tous les signaux, mise à part l'horloge, sont actifs à l'état 0 (bas).

FRAME est un signal bidirectionnel dont les événements sont toujours conduits par la source de la transaction. Il est un signal d'entrée pour la destination. Il indique le début et la durée d'une transaction. Pour indiquer sa possession du bus pour une éventuelle transaction, la source doit activer *FRAME* (événement de 1 à 0). Le signal *FRAME* reste actif durant toute la période de la transaction. Il ne passe de l'état actif à l'état inactif que lorsque la source est prête pour faire le transfert de la dernière phase de données. Le signal *FRAME* est bidirectionnel parce que tous les agents du bus doivent être capables de détecter (entrée) la fin d'une transaction précédente et prendre ensuite le contôle du bus (sortie).

IRDY (Initiator Ready) est un signal d'entrée/sortie pour la source et d'entrée pour la destination. Ce signal indique, lors d'une écriture, que la source est prête à envoyer des données aux destinations, et lors d'une lecture, qu'elle est prête à recevoir des données de la destination. Le signal *IRDY* est bidirectionnel à la source parce qu'elle doit être capable de lire les activités sur ce signal (entrée) pour savoir quand le bus passe à un état repos (fin de transaction), afin de pouvoir prendre le contrôle du bus (sortie) pour un transfert après avoir au préalable reçu une autorisation de l'arbitre.

TRDY (Target Ready) est un signal d'entrée pour la source et de sortie pour la destination. Les événements de *TRDY* sont conduits pas la destination à laquelle s'adresse la transaction. Le signal *TRDY* est activé lorsque la destination est prête pour participer ou répondre à une phase de données lors d'une transaction.

Un transfert de données est réellement produit lorsque les signaux *IRDY* et *TRDY* sont actifs et au front montant de l'horloge. Entre les phases de données durant la transaction, il y a des états de repos sur le bus appelé *Wait State*. Lors d'une phase de repos, soit *TRDY* ou *IRDY* passe de l'état actif à l'état inactif.

DEVSEL (Device Select) est activé par la destination dès qu'elle décode son adresse. C'est un signal en entrée pour la source et en sortie pour la destination. Si la source initie une transaction vers une destination et que le signal *DEVSEL* n'est pas activé après six cycles d'horloge, la source déduit que l'adresse de la destination n'est pas bonne ou que celle-ci n'est pas en mesure de répondre à la demande. La transaction est alors annulée.

5.4. Comportement des signaux de contrôle sur le PCI lors de la transaction de lecture [57]

Une fois que l'arbitre a accordé l'autorisation de transaction, l'agent doit attendre que le bus passe à un état de repos (*Idle State*), ce qui est représenté par les signaux *FRAME* et *IRDY* inactifs dans les mêmes cycles d'horloge.

Lors de cette description, nous allons nous référer à la figure 25. Chaque cycle d'horloge est numéroté pour une description plus facile. L'intervalle entre deux lignes verticales représente un cycle d'horloge qui commence et se termine au front montant.

Au début de la transaction de lecture, au premier cycle d'horloge, le signal *FRAME* est activé, passant de l'état *1* à l'état *0* (les signaux de commande étant 0 actif) pour signaler qu'une transaction a commencé sur le bus. Le signal *FRAME* reste dans son état actif jusqu'à ce que la source soit prête pour le transfert de la dernière phase de données. À ce premier cycle d'horloge, l'adresse et le type de transaction sont envoyés sur le bus des données/adresses et des commandes. Il s'écoule une période d'inactivité de l'horloge appelée "*horloge morte*", durant laquelle l'adresse et le type de transaction doivent parvenir à la destination afin de solliciter une transaction de sa part. De même, cette période permet d'éviter des collisions sur les signaux qui sont à la fois en entrée pour certains agents et en sortie pour d'autres. Une collision se produit lorsque plusieurs agents utilisent au même moment le même signal en sortie. Durant le premier cycle d'horloge, les signaux *IRDY*, *TRDY* et *DEVSEL* sont inactifs et se préparent

pour l'opération qui vient d'être amorcée. Ces signaux sont maintenus inactifs par les éléments appelés "*keeper resistors*" sur la carte du système sur laquelle est connecté le bus.

Lors du cycle d'horloge 2, la phase d'adressage et de type de transfert est terminée. La destination est prête pour transférer les données à la source qui en fait la demande. La destination doit maintenir son signal désactivé pour avoir le temps de prendre possession du bus d'adresse. Aussi, le cycle 2 est défini comme une période d'horloge morte parce que le bus d'adresse change de direction pour transférer des données de la destination à la source. De plus, la destination doit attendre que la source soit prête pour recevoir ces données. Au deuxième cycle d'horloge, la source active le signal *IRDY* pour indiquer à la destination qu'elle est en état de recevoir des données. La période entre l'activation du signal *FRAME* et celle du signal *IRDY* peut durer au maximum 8 cycles d'horloge. Dans le cas où cette période marginale est dépassée, la transaction est annulée. En activant le signal *IRDY*, si la source ne désactive pas le signal *FRAME*, alors cela signifie qu'il y a plus d'une phase de données à accomplir dans cette transaction; sinon, la source tout en activant le signal *IRDY,* aurait désactivé simultanément le signal *FRAME.*

Au cycle d'horloge 3, la destination active le signal *DEVESEL* pour indiquer qu'elle a reconnu son adresse et qu'elle est prête à participer à la transaction. *DEVSEL* reste actif jusqu'à ce que la transaction soit complétée. Elle active aussi le signal *TRDY*, pour indiquer que la première phase de données est en cours de transfert.

Lorsqu'au quatrième cycle d'horloge la source et la destination ont respectivement les signaux *IRDY* et *TRDY* activés, la première phase de données est lue. Les agents peuvent transférer plusieurs phases de données à la fois si la source l'exige et si la destination est disponible. Ce qui fait que la destination enverra les données les unes à la suite des autres. Cette phase de données est appelée "*back to back*" et peut s'étendre sur plusieurs cycles d'horloge, à condition que les deux agents de la transaction soient capables de se synchroniser et de rester actifs. Il peut arriver que l'un des agents

demande une pause *(Waite State)* lors d'une transaction pour une raison quelconque. Dans les deux cas, le signal *IRDY* ou *TRDY* peut être désactivé pendant 1 à 8 cycles d'horloge. Sur la figure 25, ce phénomène se produit au quatrième et au sixième cycle d'horloge. Des données sont transférées chaque fois que les deux signaux *IRDY* et *TRDY* sont actifs. Lorsque la source de la transaction détecte la dernière phase de donnée, elle désactive le signal *FRAME* avant le transfert de cette donnée pour indiquer que le bus est libre. Une fois que le signal *FRAME* est désactivé, une dernière phase de données se produit, avant que les signaux *IRDY* et *TRDY* ne soient désactivés. La période entre la désactivation du *FRAME* et des autres signaux ci-haut cités dure un cycle d'horloge. Aussi, après la dernière transaction de données, le bus retourne dans une période d'horloge morte afin d'éviter les phénomènes de collision.

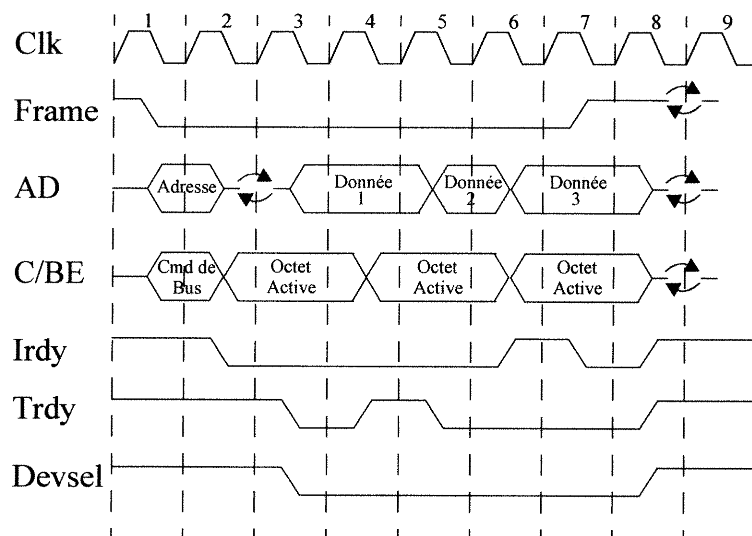


FIGURE 25. Chronogramme de la transaction de lecture sur le bus PCI

5.5. Comportement des signaux de contrôle sur le PCI lors de la transaction d'écriture [57]

Pour la description de la fonction d'écriture du bus PCI, nous nous référons à la figure 26. Nous supposons aussi que l'arbitrage a été fait, que la source est en mesure d'initier l'opération d'écriture et que le bus est retourné dans un état de repos avec les signaux *FRAME* et *IDRY* désactivés. Comme dans l'opération de lecture, nous ne parlons que des signaux de contrôle.

Au premier cycle d'horloge, la source prend possession du bus en activant le signal *FRAME* pour indiquer que l'opération d'écriture a commencé. *FRAME* reste actif jusqu'à ce que la source soit prête pour transférer la dernière phase de données. Au même moment, une adresse valide ainsi que le type d'opération sont transférés respectivement sur le bus d'adresse et sur le bus des commandes. L'adresse et le type de commande sont envoyés sur le bus pour une durée de 1 cycle d'horloge.

Une période d'inactivité de l'horloge, dite période d'horloge morte, est nécessaire pour tous les signaux qui sont conduits par plusieurs agents du bus afin d'éviter des collisons. En ce moment d'inactivité de l'horloge, les signaux *IRDY*, *TRDY* et *DEVSEL* restent ou deviennent inactifs en préparation pour la prochaine transaction sur le bus.

Au deuxième cycle d'horloge, la source ayant toujours possession du bus d'adresse peut commencer par envoyer la donnée. En effet, pour l'opération d'écriture, comme la destination n'a pas besoin de prendre possession du bus d'adresse pour transférer des données, la destination n'a pas besoin de forcer une période d'horloge morte et de maintenir ses signaux *TRDY* et *DEVSEL* désactivés comme dans le cas de l'opération de lecture. La source commence alors le transfert de la première phase de données avec l'activation du signal *IRDY*. La destination, ayant reconnu son adresse et le type d'opération au cycle d'horloge précédent, peut activer au deuxième cycle d'horloge les signaux *TRDY* et *DEVSEL* afin de participer à l'opération. Le signal *DEVSEL* une

fois activé ne peut pas être désactivé jusqu'à la fin de la transaction. Aussi, pendant ce cycle d'horloge, le signal *FRAME* n'est pas désactivé par la source, ce qui veut dire qu'il y a plus d'une phase de données dans cette transaction. Il est par ailleurs possible que la source, après voir pris possession du bus, ne soit pas en mesure de faire le transfert de la première phase de données. Elle peut garder le signal *IRDY* désactivé pour une période de 1 à 8 cycles d'horloge après que le signal *FRAME* ait été activé.

Au troisième cycle d'horloge, une phase de données est transférée, car les signaux *IRDY* et *TRDY* sont actifs. Le signal *FRAME* n'est pas désactivé à ce stade non plus, ce qui indique que la prochaine phase de donnée n'est pas la dernière.

Au quatrième cycle d'horloge, la destination désactive son signal *TRDY*, forçant un "*waite state*" bien que le signal *IRDY* de la source reste actif. Ce "*waite state*" ne peut durer plus de 8 cycles d'horloge.

Au cinquième cycle d'horloge, la source a toujours son signal *IRDY* actif et la destination active aussi son signal *TRDY*. Ce qui veut dire qu'une phase de données peut être transférée à ce stade de la transaction.

Au sixième cycle d'horloge, les deux agents désactivent respectivement leur signal *IRDY* et *TRDY* pour demander un état de pause *(Waite State)* pour des raisons quelconques. Lors de ce "*Waite State*", la source n'est pas prête à transférer sa dernière phase de données. Le signal *FRAME* reste alors actif.

Au septième cycle d'horloge, le "waite state" est terminé pour les deux agents de la transaction. La source étant prête pour le transfert de la dernière phase de données, elle désactive le signal *FRAME*. Les signaux *IRDY* et *TRDY* deviennent actifs pour le transfert de la dernière phase de données.

Au huitième cycle d'horloge, la source a terminé d'envoyer des données sur le bus. Elle désactive son signal *IRDY*. De même, la destination désactive les signaux *TRDY* et *DEVSEL* et retourne le bus dans un état de repos (*Idle State*). Le bus, à ce stade, peut

retourner dans une période d'horloge morte pour permettre que le transfert se termine de façon stable et pour éviter des collisions.

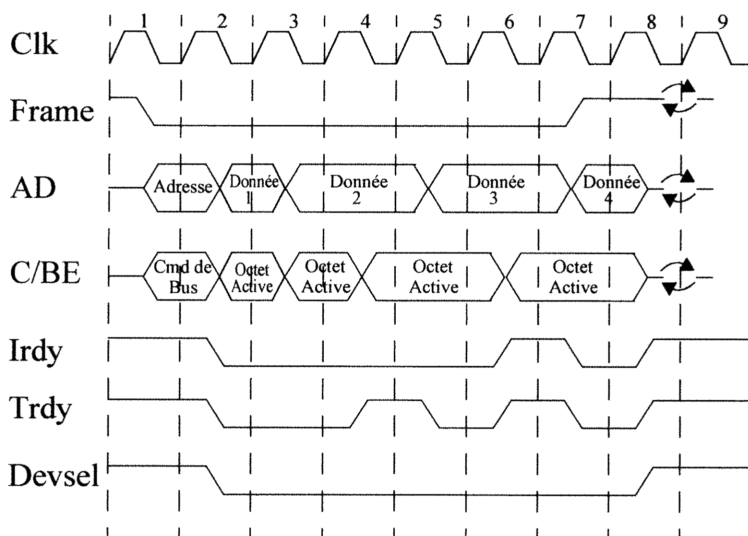


FIGURE 26. **Chronogramme de la transaction d'écriture sur le bus PCI**

5.6. Interruptions prématurées des transactions sur le bus PCI

Pour certaines raisons, il peut arriver que la source ou la destination arrête prématurement une transaction. Cette section définit les circonstances qui nécessitent les arrêts prématurés et les mécanismes utilisés par les agents pour mettre en oeuvre ces arrêts. Nous aborderons premièrement les arrêts prématurés par la source et deuxièment ceux demandés par la destination.

5.6.1. Les Raisons pour lesquelles la source peut interrompre une transaction

La source peut arrêter une transaction pour l'une des trois raisons suivantes:

1. La transaction est complétée normalement. Toute la donnée impliquée dans cette transaction à été transférée vers/de la destination. Ceci n'est pas un arrêt prématuré.

2. Une autre source a reçu l'autorisation de l'arbitre pour un autre transfert avant que l'initiateur de la transaction actuelle ne reçoive son autorisation. La durée de la transaction de la source qui occupe le bus actuellement est tellement longue que le temps de latence de l'initiateur qui vient d'avoir l'autorisation de l'arbitre pour une transaction est dépassé. L'arbitre du bus lui retire tout simplement son accord pour le transfert. Il doit arrêter prématurement son transfert.
3. L'initiateur doit arrêter la transaction prématurement parce qu'il n'y a pas de destination qui répond à cette adresse.

L'arrêt normal et non prématuré d'un transfert sur le bus PCI est décrit dans les transactions de lecture et d'écriture des sections précédentes. Les deux derniers types d'arrêt sont décrits dans les sous sections suivantes.

Nous n'avons pas parlé en détails du cas où l'arbitre retire à une transaction son accord parce que nous ne traitons pas de l'arbitrage. Nous n'avons parlé ici que des signaux de contrôle dans le cas des opérations de lecture et d'écriture.

Lorsqu'une destination de transaction ne reconnait pas son adresse et ne réclame pas le transfert en cours (le signal *DEVSEL* n'est pas activé par la destination dans un délai prédéfini), la source interrompt la transaction pour les raisons suivantes:

1. Il n'y a pas d'agent resident à l'adresse à laquelle la source demande une transaction. Dans ce cas, aucune destination ne réclame la transaction et le signal *DEVSEL* n'est pas activé. La source interrompt la transaction et signale une erreur.
2. Lors d'une transaction, lorsque la source doit diffuser un message spécial par un cycle spécial à plusieurs destinations simultanément, il n'y a pas de destination qui active le signal *DEVSEL* en réponse à la transaction. Lorsque plusieurs destinations reconnaissent leur adresse lors d'un message spécial et activent leur signal *DEVSEL*, la ligne de *DEVSEL* devient un contentieux entre les destinations. La source interrompt aussitôt la transaction. Ceci n'est pas considéré comme une erreur car c'est la façon normale pour la source de terminer un message spécial.

3. Lorsque la source initie une transaction en utilisant une commande reservée, aucune destination ne répond à cette transaction. Aucun signal *DEVSEL* n'est activé. La source interrompt le transfert.

Un autre cas est à considérer lorsqu'une transaction est prématurément interrompue par la source. Lorsqu'une transaction d'une phase de données est initiée et qu'il n'y a pas de réponse de la part d'une destination, la source doit interrompre la transaction. En effet, lorsqu'une transaction est initiée, la source doit permettre aux destinations parmi les agents à décodeur d'adresse rapide de répondre. Si elle n'a pas de réponse dans le premier cycle d'horloge, elle va attendre les agents dont la vitesse du décodeur d'adresse est moyenne. Si parmi ces agents il n'y aucune destination qui réponde à la transaction, la source va attendre la réponse des agents à vitesse de décodage d'adresse lente. Dans le cas où il n'y a toujours pas de réponse, la source attend de voir si le pont d'expansion du bus PCI peut répondre à la transaction. Si la source n'a pas reçu de message du pont d'expansion du bus PCI, elle imterrompt la transaction.

5.6.2. Les raisons pour lesquelles la destination peut interrompre une transaction

En utilisant les signaux *TRDY* et *DEVSEL* en conjonction avec le signal *STOP*, la référence [17] définit trois modes d'interruption de transaction, à savoir:

1. L'activation du signal *STOP* indique à la source que la transaction doit être interrompue. Si les signaux *TRDY* et *DEVSEL* restent actifs pendant que le signal *STOP* est activé, la destination indique à la source qu'elle va transférer la dernière phase de données. Ceci amène la source à interrompre prématurément la transaction au cycle d'horloge suivant. Deux types d'interruption sont à considérer. Lorsque *TRDY* et *STOP* sont activés et que *IRDY* n'est pas actif, le transfert de la dernière phase de données ne peut pas se faire. Lorsque *IRDY* devient actif, la source désactive au même moment le signal *FRAME* pour indiquer la fin de la transaction. La source désactive son signal *IRDY* après le transfert de la dernière phase de données. La destination désactive à son tour ses signaux *TRDY*, *DEVSEL* et *STOP*.

2. Lorsque les signaux *TRDY* et *STOP* sont activés lors d'une transaction par un agent destination et que le signal *IRDY* de la source est aussi actif, la dernière phase de données est transférée en ce cycle d'horloge. Comme la source ne sait pas que c'est la dernière phase de données qu'elle va effectuer, le signal *FRAME* n'est pas désactivé avant le transfert de la donnée. La source ne désactive le signal *FRAME* que lors de la dernière phase de données. Cela implique qu'il reste une phase de données à transférer. La destination ayant activé le signal *STOP* avant la dernière phase de données désactive son signal *TRDY* pour forcer la fin de la transaction. Une phase de données dite "*phase de donnée nulle*" a lieu à cause du fait que *IRDY*, *STOP* et *DEVSEL* sont actifs. Dans une situation comme celle-ci, la destination n'est pas autorisée à désactiver les signaux *DEVSEL* et *STOP* avant que la source ne désactive *FRAME*.

   Plusieurs agents destinations du bus PCI ne supportent que des transactions à une seule phase de données. Lorsque la source initie une transaction de plusieurs phases de données avec les destinations qui n'en supportent qu'une seule ces destinations, après la première phase de données, activent leur signal *STOP* pour interrompre la transaction.

3. Il peut aussi arriver qu'une destination interrompe une transaction en désactivant *TRDY* au moment où *STOP* est activé. Les phases de données qui vont suivre cette étape sont tout simplement ignorées par la destination. La source doit désactiver le signal *FRAME* au prochain cycle d'horloge et terminer la transaction au cycle suivant sans transfert de données. La destination ne désactive ses signaux *TRDY* et *DEVSEL* qu'après que le signal *FRAME* soit désactivé par la source.

Voici les raisons pour lesquelles une destination peut interrompre une transaction:

1. Lorsque la destination détermine que le temps nécessaire pour accomplir le transfert d'une phase de données est supérieur à 8 cycles d'horloge, sauf le cas de la première phase de donnée qui doit respecter la règle des 16 cycles d'horloge, et que la source initie une transaction, la destination fait une demande d'interruption

décrite dans la partie 3 des types d'interruption de transfert par la destination. Ceci permet d'éviter que les agents très lents monopolisent le bus et d'empêcher les autres agents qui font la demande pour un transfert d'être coincés.

2. Lorsque la destination de la transaction ne supporte pas le mode de transfert "*burst*" du bus PCI, et qu'elle détecte que la source veut un transfert de plusieurs phases de données, la destination fait une demande d'interruption 3 des modes d'interruption de transfert.
3. Lorsque l'adresse indiquée par la source lors d'une transaction est une adresse d'une page ou d'un bloc mémoire et que toute l'espace n'est pas adressable pendant la première phase de données de la transaction en cours, la destination doit faire une demande d'interruption 1 ou 2 des modes d'interruption de transfert par la destination, ou comme alternative, faire la demande d'interruption de transfert 3. Cela oblige la source à fragmenter un transfert de plusieurs phases de données en plusieurs transferts d'une phase de données.

Plusieurs autres raisons décrites dans la référence [57] peuvent entraîner une demande d'interruption de la part de la destination, mais nous ne les aborderons pas dans cette sous section, puisque ces raisons se rapportent aux comportements du bus PCI qui ne sont pas traités ici.

5.7. Les contraintes temporelles lors d'une transaction sur le bus PCI

Les contraintes temporelles sur le bus PCI sont multiples. Nous avons: les contraintes temporelles qui concernent l'accès au bus, les contraintes de l'arbitrage, les contraintes de l'aquisition du bus etc ... À titre d'exemple de notre méthode, nous ne parlerons ici que des contraintes qui ont rapport avec les fonctions de lecture et d'écriture.

5.7.1. Prévenir la monopolisation du bus par la source

Pour prévenir la monopolisation du bus, chaque source a un temps maximum, dit temps de latence (*LT*), au delà duquel le bus ne peut pas faire de transaction.
Lorsqu'une transaction est initiée (*FRAME* activé), le registre de *LT* est initialisé à la durée maximale de la transaction. À chaque cycle d'horloge du bus, la valeur du registre est décrementée. Si cette valeur est égale à 0 alors que la transaction n'est pas terminée (*FRAME* désactivé), alors la source droit céder le bus pour un autre arbitrage.

Il existe plusieurs raisons pour lesquelles la source peut continuer la transaction en cours même si le temps de latence *LT* est terminé. Or comme nous n'avons considéré que le cas simplifié des transactions de lecture et d'écriture sur le bus, nous n'avons pas tenu compte de ces raisons dans ce cas d'étude.

Une fois que la source active le signal *FRAME* pour initier une transaction, *IRDY* ne doit pas rester inactif pendant plus de 8 cycles d'horloge.

Une autre règle dit que la source ne doit pas désactiver son signal *IRDY* pendant plus de 8 cycles d'horloge *(Wait State)* lors d'une phase de données.

5.7.2. Prévenir la monopolisation du bus par la destination

Il arrive souvent que des destinations lentes d'une transaction monopolisent le bus. Pour prévenir ce problème, deux cas qui concernent notre étude ont été identifiés.

1. Le temps de latence de la destination est défini comme la période de temps entre le début d'une transaction jusqu'à ce que la destination transfère la première phase de données. Si le temps nécessaire pour compléter cette phase de données est supérieur à 16 cycles d'horloge, la destination fait une demande d'interruption afin de permettre à la source de faire une nouvelle demande de transaction à l'arbitre.
2. Si le signal *TRDY* doit être désactivé lors de la durée d'une phase de données autre que la première, et que la source n'est pas prête pour la dernière phase de données

(*FRAME* désactivé), la destination doit faire une demande d'interruption de transaction si la durée de cet état dépasse 8 cycles d'horloge.

5.8. Modèle HAAD synchrone des transactions de lecture et d'écriture

Dans cette section, nous décrirons de comment, à partir du modèle HAAD synchrone des transactions simplifées de lecture et d'écriture du bus PCI, en arriver aux modèles de la machine STFSM de ces fonctions, et ensuite à la machine observatrice des protocoles de ces transactions. Nous nous sommes penchés dans un premier cas sur la décomposition des diagrammes à action en diagrammes transformables en modèle générique STFSM et pour ensuite construire les STFSM correspondants aux diagrammes feuilles.

La figure 27 montre que les fonctions de lecture et d'écriture peuvent être composées par l'opération de composition *boucle* sur une opération *choix retardé*.

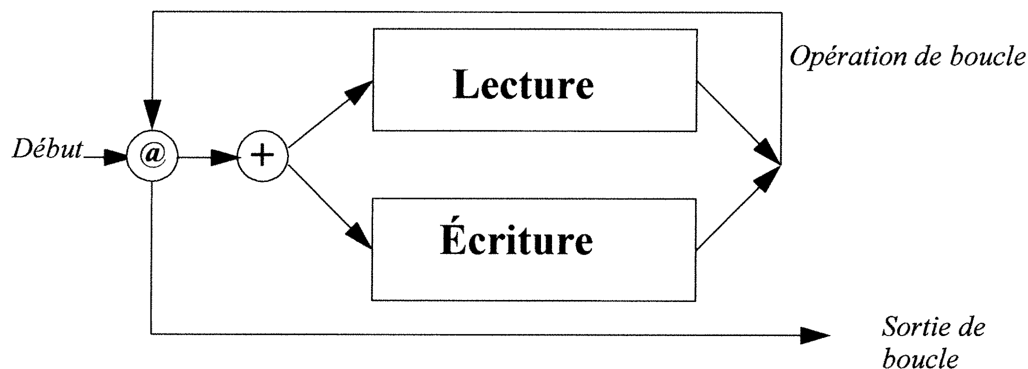


FIGURE 27. Modèle HAAD des transactions de lecture et d'écriture sur le bus PCI

À partir des contraintes temporelles décrites dans la section 5.7, nous avons enrichi les chronogrammes simplifiés des transactions de lecture (figure 25) et d'écriture (figure 26) pour obtenir respectivement la figure 28 et la figure 29.

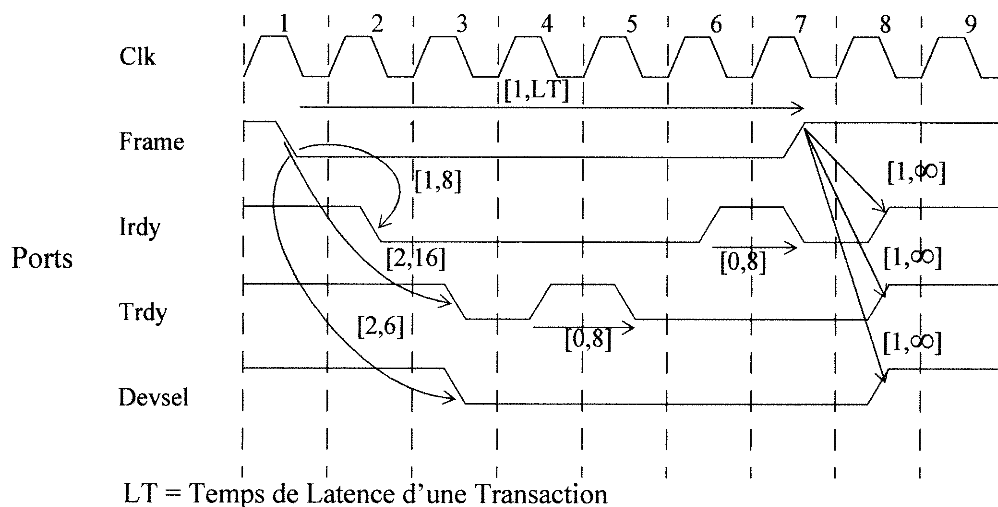


FIGURE 28. Diagramme à action de la transaction de lecture sur le bus PCI

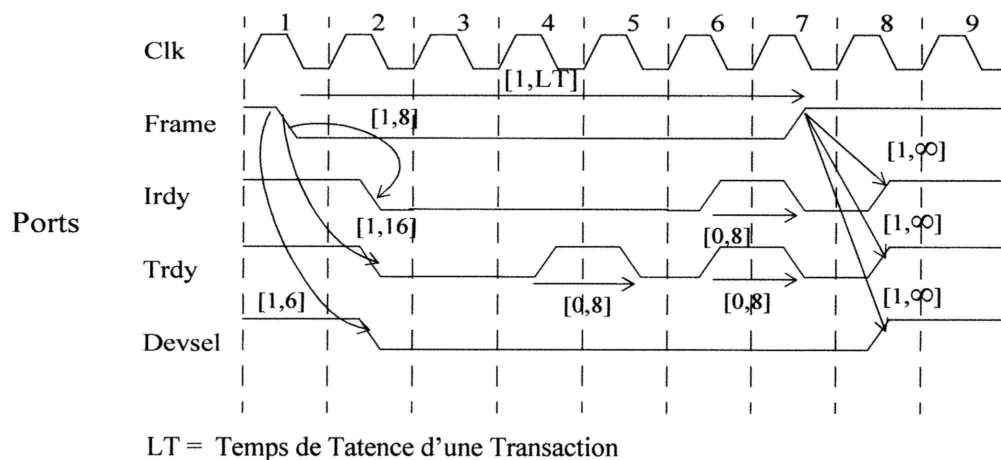


FIGURE 29. Diagramme à action de la transaction d'écriture sur le bus PCI

5.8.1. Traduction de la transaction de lecture en diagrammes feuilles transformables en STFSM

Lorsque nous avons appliqué la méthode de la décomposition parallèle expliquée dans le chapitre 3 au diagramme à actions de la figure 28, nous avons obtenu un ensemble de diagrammes feuilles montrés par les figures 30, 31 et 32. Ici, nous avons regroupé des signaux dont les événements dépendent les uns des autres en diagrammes feuilles.

Frame_Irdy

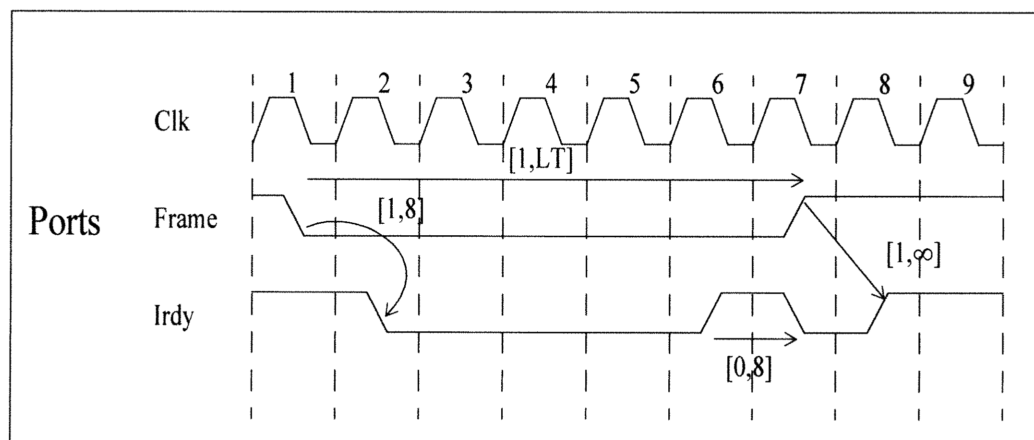


FIGURE 30. Diagramme feuille Frame_Irdy issu de la décomposition parallèle de la figure 28

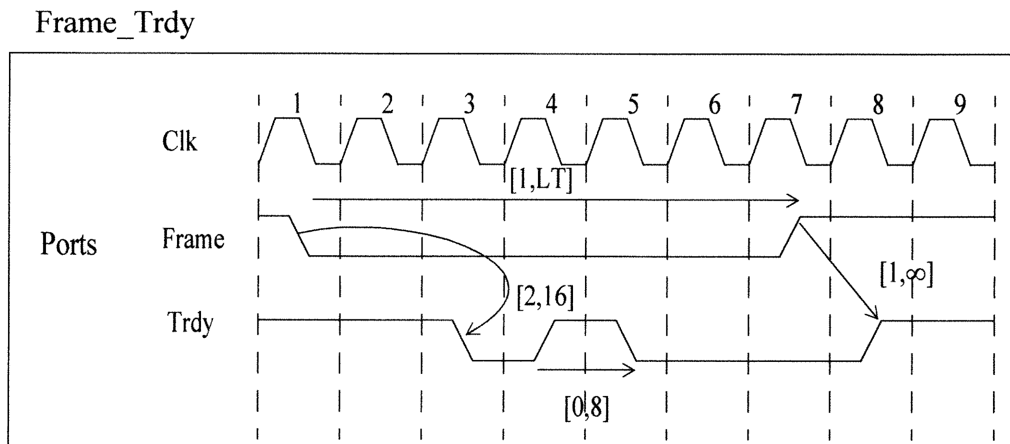


FIGURE 31. Diagramme feuille Frame_Trdy issu de la décomposition parallèle de la figure 28

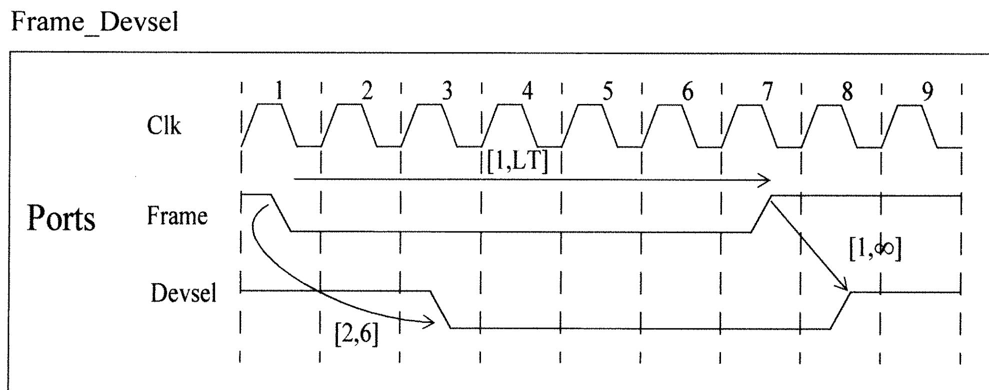


FIGURE 32. Diagramme feuille Frame_Devsel issu de la décomposition parallèle de la figure 28

Les figures issues de la décomposition parallèle peuvent êtres recomposées par l'opération de composition *parallèle* pour obtenir la figure 33.

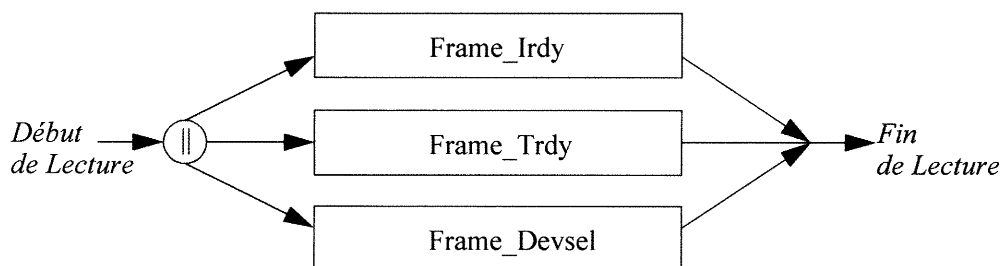


FIGURE 33. Composition parallèle de diagrammes feuilles réduits issus de la décomposition de la figure 28

Les diagrammes feuilles des figures 30 à 32 sont alors décomposés en cascade pour donner des diagrammes feuilles minimaux qui ne contiennent que des fragments de transaction.

Les figures 34 à 36 représentent à la fois la décomposition en cascade des diagrammes feuilles *Frame_Irdy, Frame_Trdy* et *Frame_Devsel* respectivement des figures 30 à 32 en diagrammes feuilles encore plus simples transformables en machine STFSMs. Aussi pour ne pas répéter les mêmes figures plusieurs fois, nous avons montré dans les figures 34 à 36 comment les diagrammes feuilles sont composés par les opérations de composition pour obtenir les mêmes spécifications que celles de la figure 30 à 32 respectivement. Les boîtes des figures représentent ainsi les diagrammes issus de la décomposition cascade.

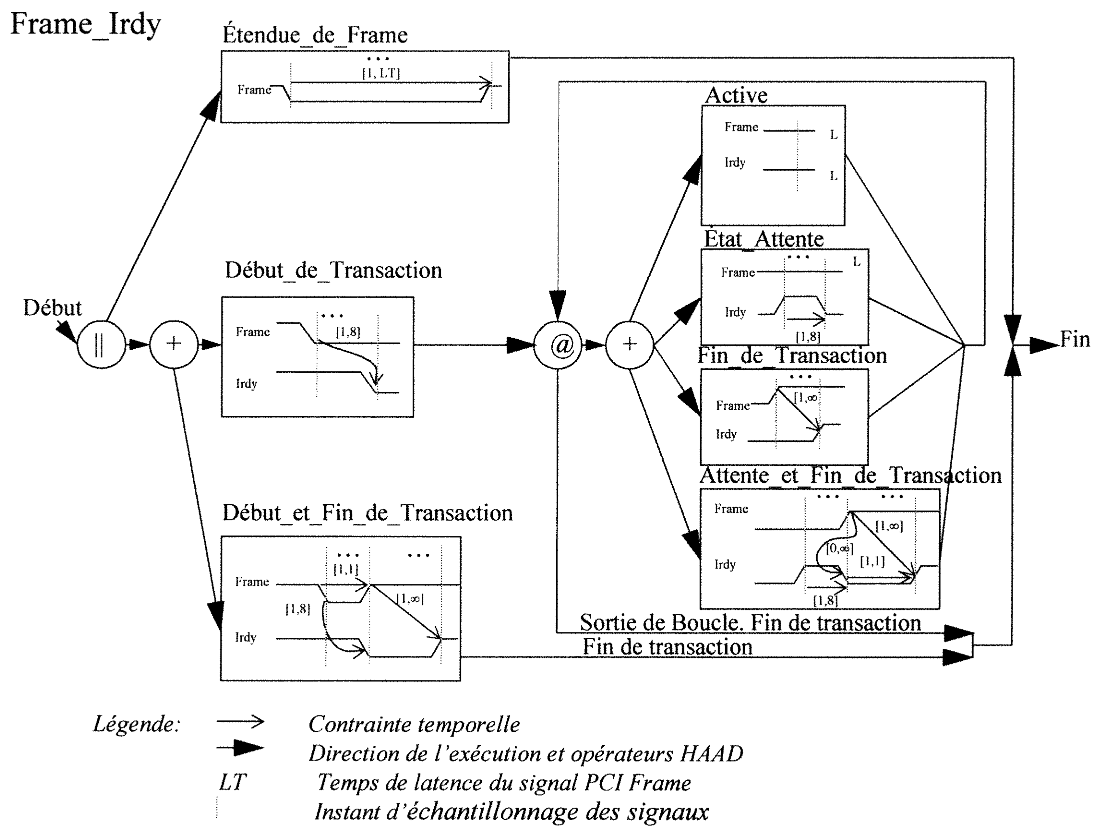


FIGURE 34. Modèle HAAD du fragment Frame_Irdy de la transaction de lecture du bus PCI

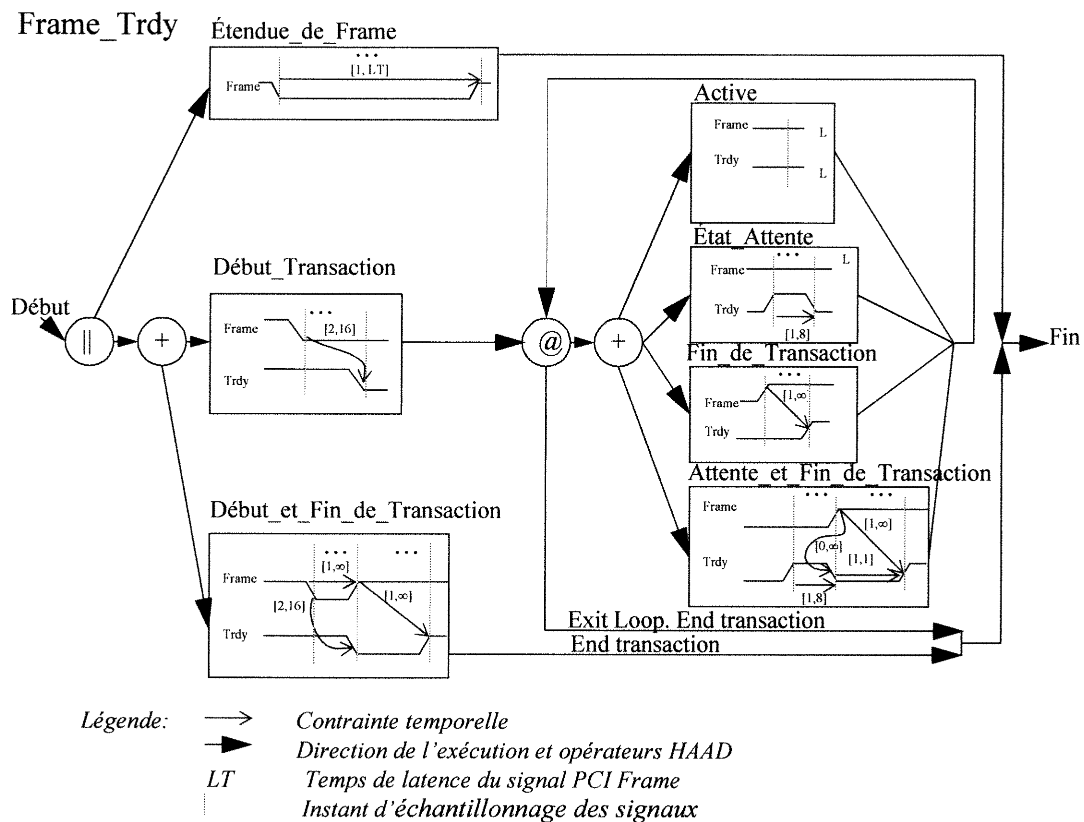


FIGURE 35. Modèle HAAD du fragment Frame_Trdy de la transaction de lecture du bus PCI

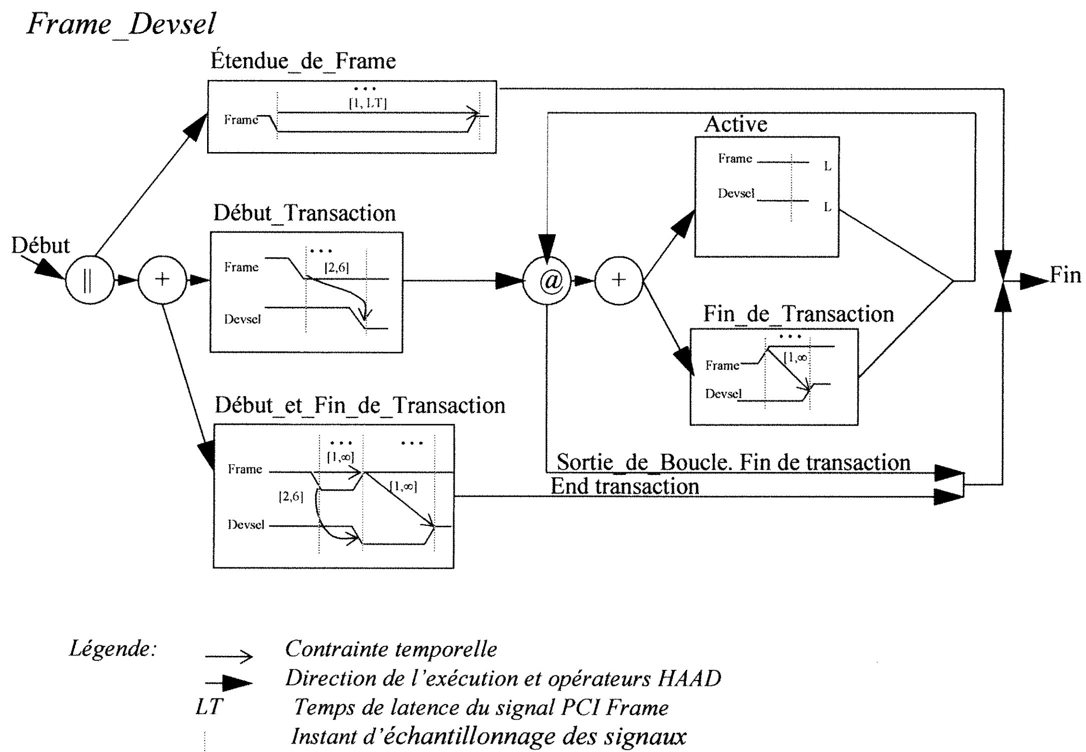
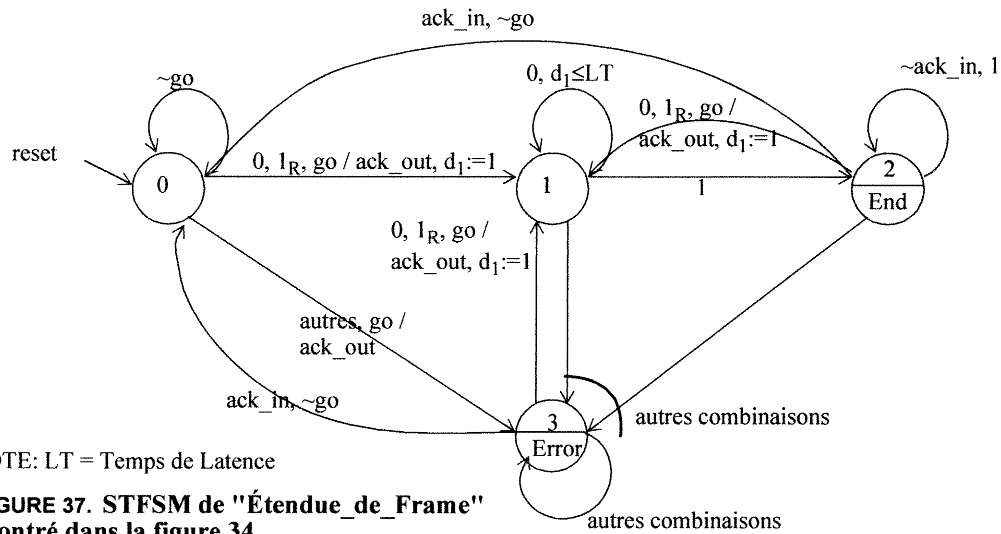


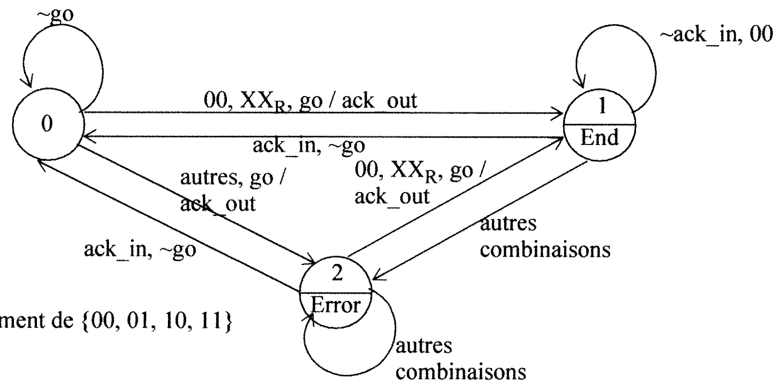
FIGURE 36. Modèle HAAD du fragment Frame_Devsel de la transaction de lecture du bus PCI

Après la décomposition en cascade des diagrammes à actions réduits issus de la décomposition parallèle, nous avons transformé tous les diagrammes résultants en machines STFSMs comme le montre les figures 37 à 43.



NOTE: LT = Temps de Latence

FIGURE 37. STFSM de "Étendue_de_Frame" montré dans la figure 34



NOTE: XX est élément de {00, 01, 10, 11}

FIGURE 38. STFSM "Active" montré dans la figure 34

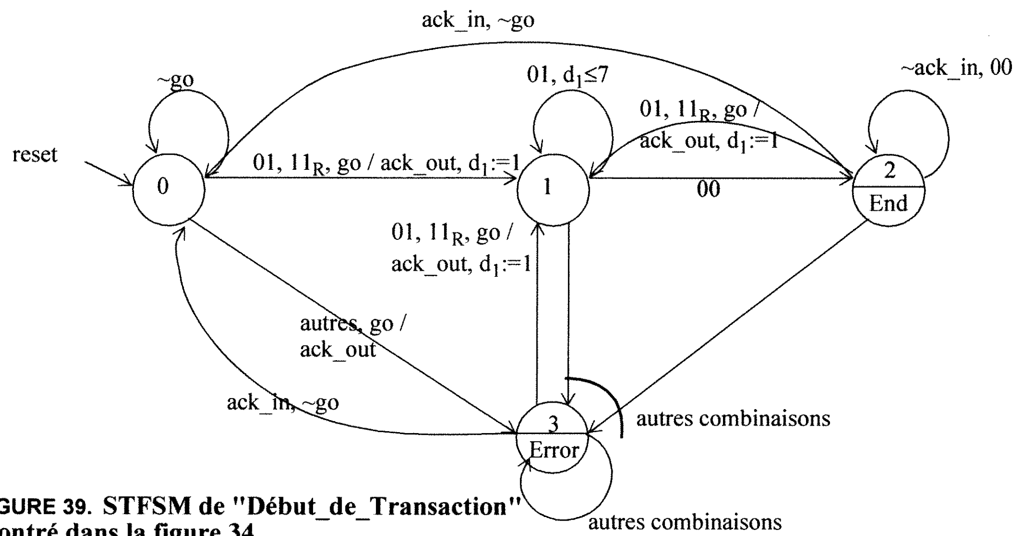
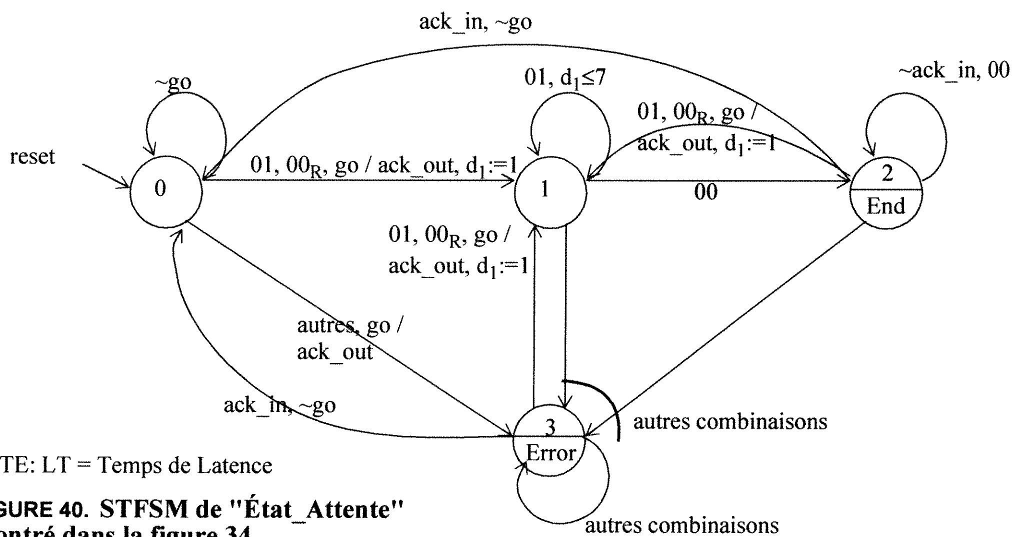
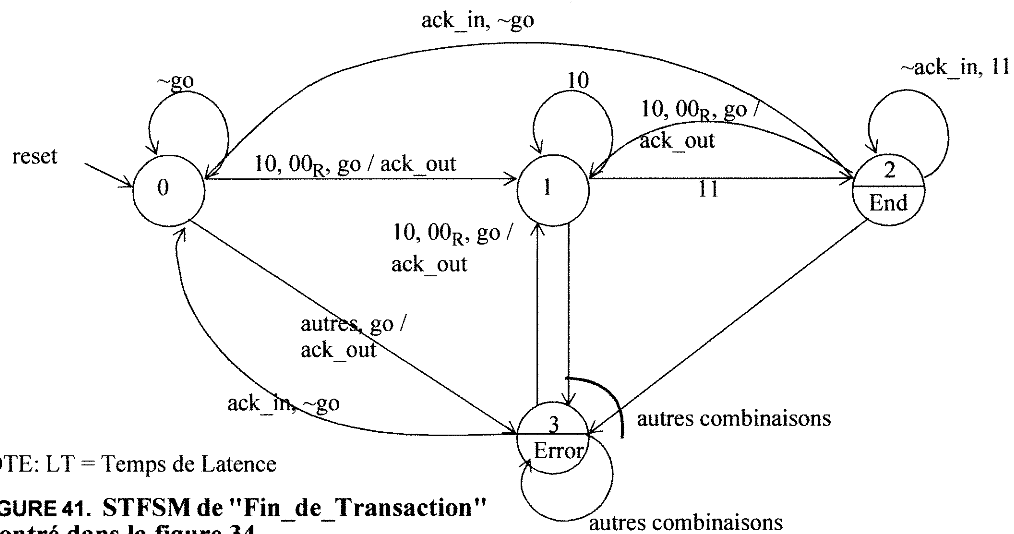


FIGURE 39. STFSM de "Début_de_Transaction" montré dans la figure 34



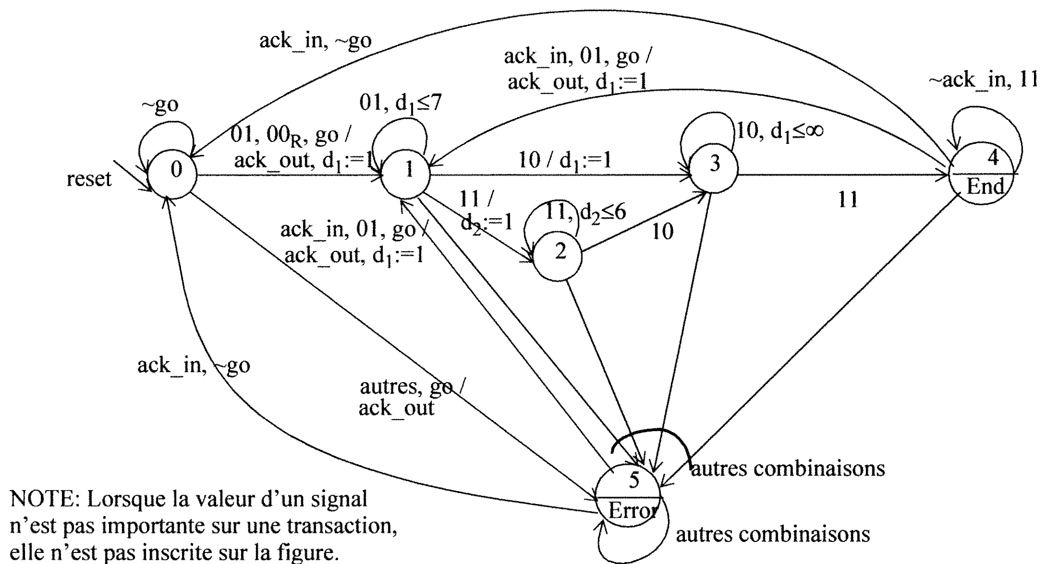
NOTE: LT = Temps de Latence

FIGURE 40. STFSM de "État_Attente" montré dans la figure 34



NOTE: LT = Temps de Latence

FIGURE 41. STFSM de "Fin_de_Transaction" montré dans la figure 34



NOTE: Lorsque la valeur d'un signal n'est pas importante sur une transaction, elle n'est pas inscrite sur la figure.

FIGURE 42. STFSM de "Attente_et_Fin_de_Transaction" montré dans la figure 34

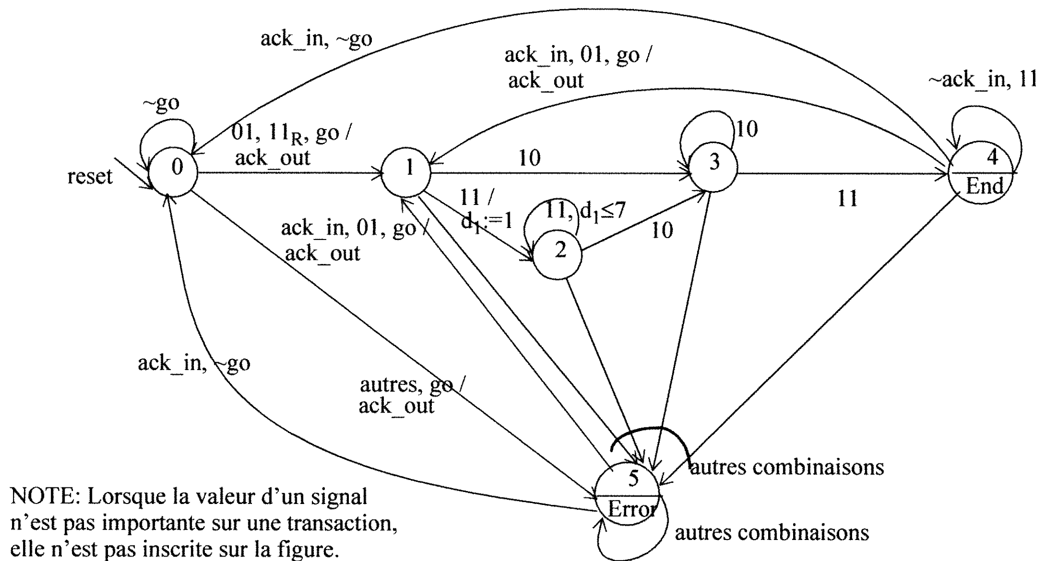


FIGURE 43. STFSM de "Début_et_Fin_de_Transaction" montré dans la figure 34

De façon analogue au diagramme *Frame_Irdy*, nous avons transformé les diagrammes à actions réduits *Frame_Trdy* et *Frame_Devsel* des figures 35 et 36 en machines STFSM. Nous avons aussi remarqué qu'il existe des diagrammes à actions réduits issus des décompositions en cascade qui se ressemblent et ont les mêmes caractéristiques. Pour éviter la redondance, nous avons réutilisé certaines machines STFSMs comme "*Active*", "*Attente_et_Fin_de_Transaction*", "*Étendue_de_Frame*" et "*Fin_de_Transaction*".

5.8.2. Traduction de la transaction d'écriture en diagrammes feuilles transformables en STFSM

La traduction de la transaction d'écriture est identique à celle de la traduction de la transaction de lecture. Seules les contraintes temporelles sont différentes. Nous avons décomposé parallèlement le diagramme à action de la transaction d'écriture (figure 29) en diagrammes réduits pour donner les diagrammes représentés par les figures 44 à 46.

Frame_Irdy

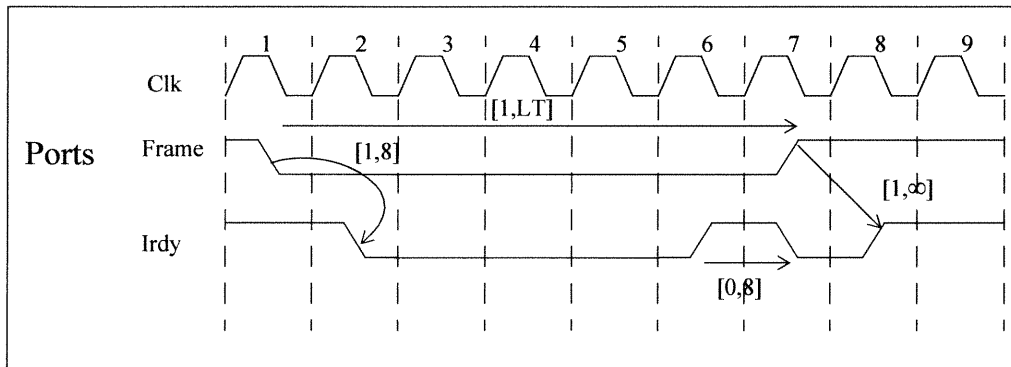


FIGURE 44. Diagramme feuille Frame_Irdy issu de la décomposition parallèle de la figure 29

Frame_Trdy

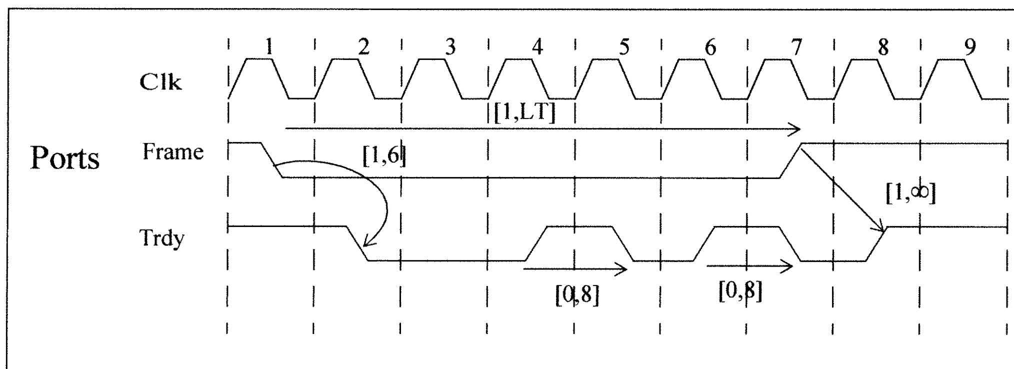


FIGURE 45. Diagramme feuille Frame_Trdy issu de la décomposition parallèle de la figure 29

Frame_Devsel

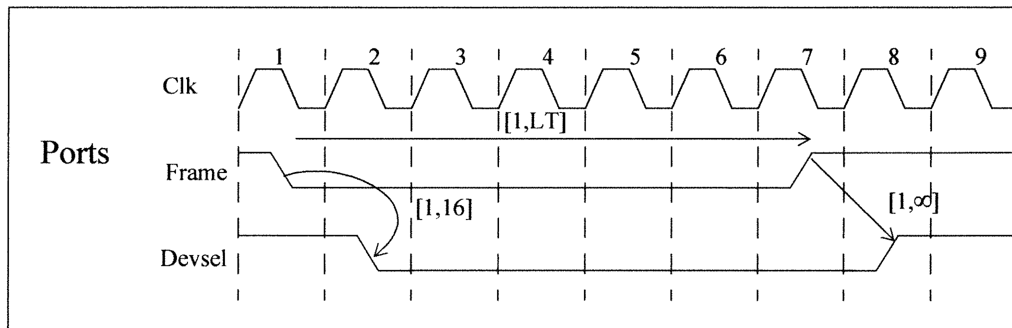


FIGURE 46. Diagramme feuille Frame_Devsel issu de la décomposition parallèle de la figure 29

Comme dans le cas de la décomposition parallèle de la fonction de lecture, les diagrammes feuilles issus de la décomposition peuvent être recomposés et donner la figure 47 suivante:

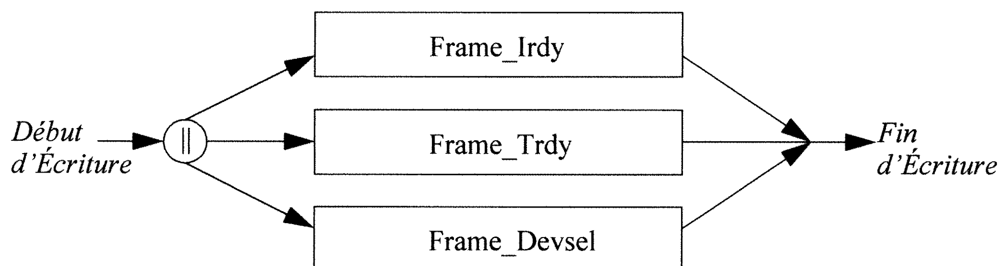


FIGURE 47. Composition parallèle de diagrammes feuilles réduits issus de la décomposition de la figure 25

Tous les diagrammes feuilles des figures 44 à 46 peuvent être décomposés en cascade comme dans le cas de la transaction de lecture. Nous avons obtenu les figures 48 à 50 suivantes:

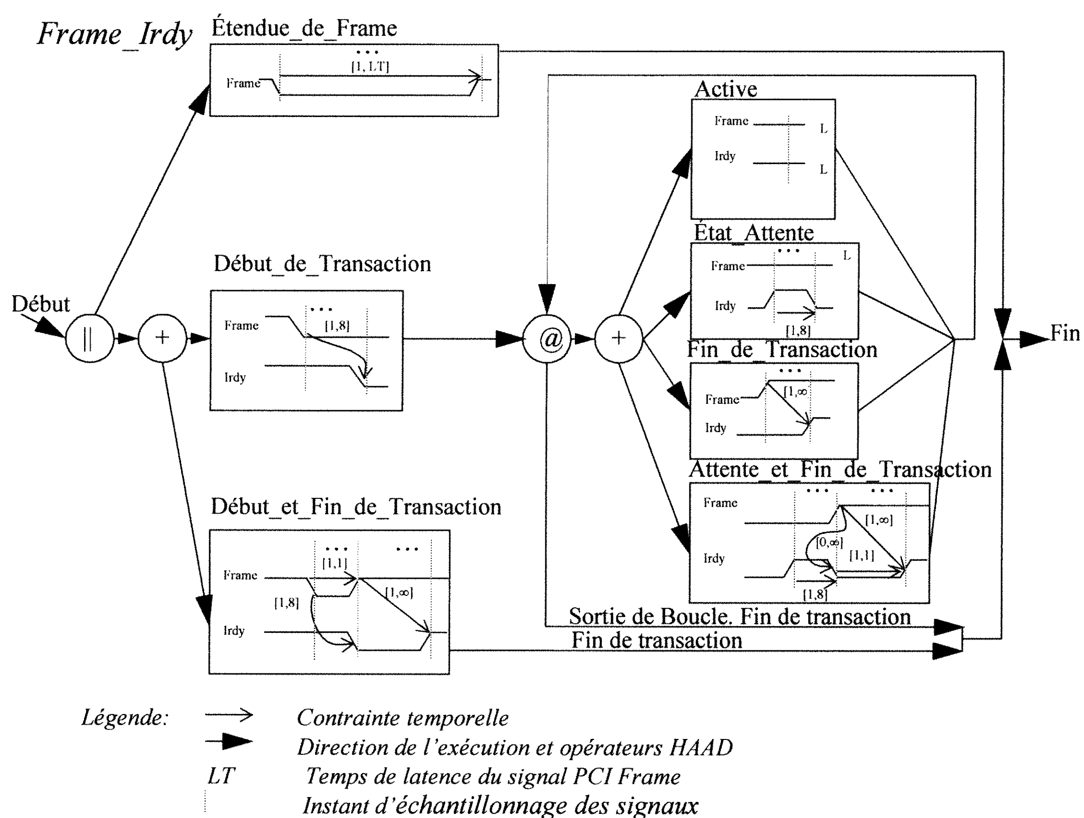
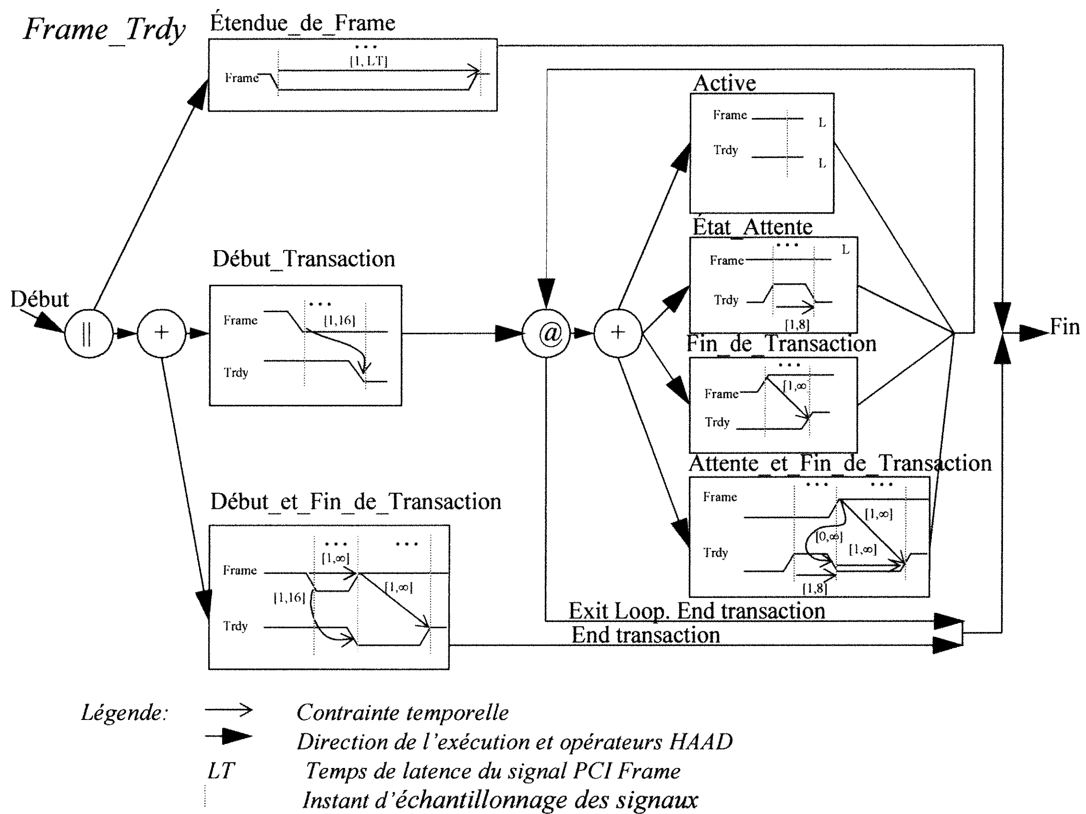


FIGURE 48. Modèle HAAD du fragment Frame_Irdy de la transaction d'écriture du bus PCI



Légende: → *Contrainte temporelle*
➔ *Direction de l'exécution et opérateurs HAAD*
LT *Temps de latence du signal PCI Frame*
Instant d'échantillonnage des signaux

FIGURE 49. Modèle HAAD du fragment Frame_Trdy de la transaction d'écriture du bus PCI

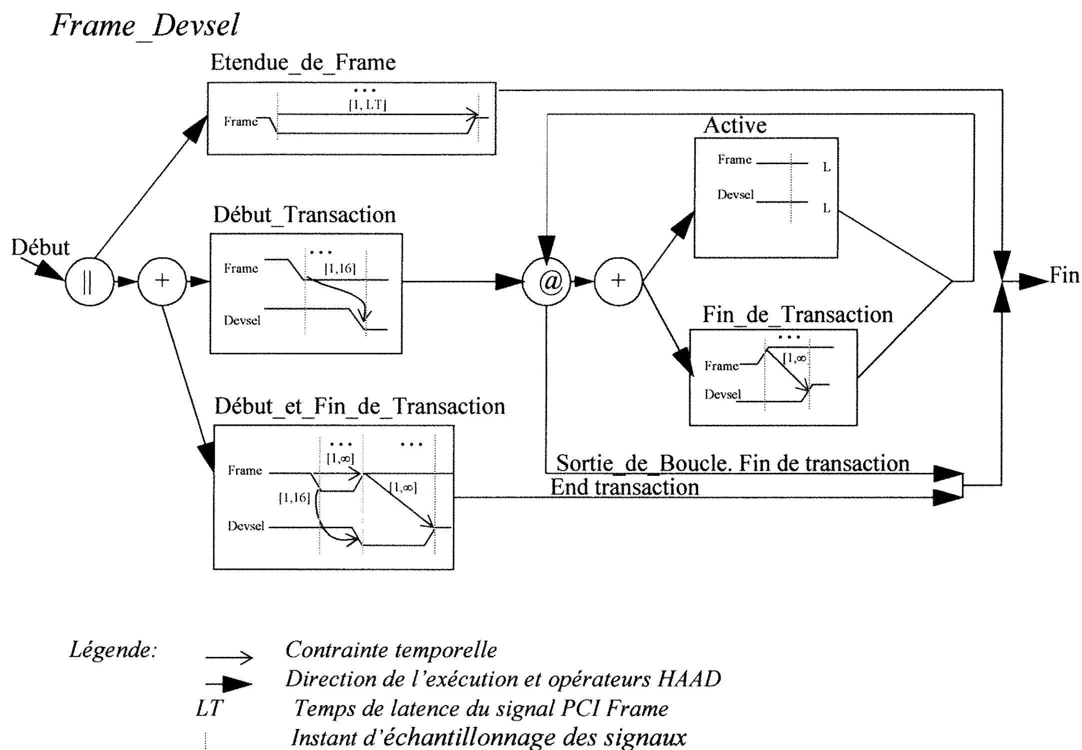


FIGURE 50. Modèle HAAD du fragment Frame_Devsel de la transaction d'écriture du bus PCI

Une fois la décomposition en cascade de la transaction d'écriture terminée, les diagrammes feuilles conformes au modèle STFSM sont transformés en machines à états. Nous pouvons remarquer que les diagrammes feuilles issus de la décomposition en cascade de *Frame_Irdy* de la transaction de lecture sont identiques à ceux de la transaction d'écriture. Dans les autres cas, la seule différence entre les autres figures sont les contraintes temporelles qui concernent juste les débuts de transaction. La suite des transactions est identique. De façon optimale, nous n'avons pas besoin de reproduire les machines à états des diagrammes feuilles issus de la décomposition en cascade de la transaction d'écriture. Elles sont identiques aux machines à états de la transaction de lecture.

Une fois que les décompositions en cascade sont faites sur les sous fonctions issues des décompositions parallèles des fonctions de lecture et d'écriture du bus PCI, nous pouvons créer des machines STFSMs pour chaque diagramme LAD de la spécification de HAAD de ces fonctions. De ces machines, nous pouvons ainsi générer une machine observatrice des fonctions de lecture et d'écriture en traduisant toutes les machines STFSMs en langage Verilog et en créant un module pour chaque opération de composition en une machine STFSM composée comme nous l'avons montré dans le chapitre 4 lors de la vérification de la méthode pour la synthèse de la machine observatrice à partir des spécifications HAAD.

5.9. Conclusion

Nous avons présenté dans ce chapitre un exemple pour soutenir la méthode que nous avons proposée dans le chapitre 3 pour la synthèse de machines observatrices à partir des spécifications HAAD. Cet exemple nous permet d'illustrer que cette méthode non seulement est valide comme le montrent les étapes de la vérification du modèle générique et des opérations de composition dans le chapitre 4, mais qu'elle peut aussi s'appliquer à un cas concret.

Le chapitre suivant nous permettra de conclure ce projet. Nous y résumons nos travaux accomplis, montrons comment nos travaux peuvent être intégrés dans les projets sur la vérification des systèmes numériques et proposons nos plans pour les travaux futurs.

CHAPITRE 6

Conclusion - synthèse des EFSM observatrices à partir des HAAD

Il n'y a aucune raison pour qu'un écrivain qui a peu de métier n'arrive pas à finir un livre.
Françoise Mallet-Joris

Extrait d'un Entretien avec Claude Servan-Schreiber - Mars 1976

Nous avons remarqué qu'avec le développement des systèmes électroniques et la capacité des circuits pouvant atteindre des dizaines de millions de portes logiques, les méthodes traditionnelles de vérification ont perdu de leur efficacité. Nous avons encore en mémoire l'erreur du Pentium de Intel sur le FDIV (Floating point DIVision) qui a coûté à la compagnie de très grandes sommes en perte. La simulation qui, si elle est faite de manière exhaustive, peut répondre aux attentes des concepteurs en terme de vérification, n'a actuellement ni les moyens temporels ni les moyens matériels de trouver toutes les combinaisons possibles d'un circuit de plusieurs dizaines de millions de portes et de simuler tous ses comportements. Les méthodes formelles sont considérées comme la solution pour pallier les insuffisances de la simulation avec le développement de plusieurs méthodes basées sur des approches mathématiques. Malheureusement, elles aussi ne sont pas capables de traiter les systèmes complexes pour le moment. De ce fait, plusieurs méthodes alternatives et outils sont en cours d'étude afin de pouvoir combler les insuffisances de la simulation et pouvoir rendre transparentes aux utilisateurs les rigueurs des méthodes formelles. C'est dans cet ordre

d'idée que nous avons proposé la synthèse des machines observatrices des protocoles de bus. Notre but est de trouver une méthode qui permette de générer automatiquement un observateur des protocoles à partir de spécifications HAAD. Cette méthode peut être utilisée en simulation, émulation et en méthodes formelles, tout en permettant d'éviter les problèmes classiques de la simulation et les limites des méthodes formelles.

6.1. Travaux accomplis

Dans ce projet, nous avons travaillé sur la synthèse des machines à états observatrices à partir des spécifications HAAD. Le but était de définir une méthode pour la génération de machines observatrices qui vont jouer le rôle d'observateurs des transactions sur un bus (ne participant pas à la transaction du bus) et de signaler toute violation des protocoles de spécification de ces bus. La machine observatrice synthétisée peut être utilisée par simulation ou par méthodes formelles, dès le début du développement du bus jusqu'à son intégration, ou en émulation. La synthèse et l'utilisation de la machine observatrice peuvent être faites dès que la spécification du protocole du bus est terminée et à n'importe quel moment de son cycle de développement. Nous avons utilisé des formalismes HAAD pour la spécification des protocoles de bus. À cet effet, nous avons traduit les spécifications des protocoles du bus en spécifications hiérarchiques HAAD. Cette traduction nous permet non seulement de faire une subdivision des protocoles en spécifications moins complexes, mais aussi de pouvoir construire assez facilement des machines à états synchrones temporisées STFSM correspondantes.

La machine observatrice est générée à partir des spécifications HAAD des protocoles du bus et est implémentée au niveau RTL synthétisable, soit en Verilog ou en VHDL, comme réseau coordonné de machines à états communicants.

Pour prouver la validité de la méthode que nous avons proposé pour la synthèse des machines observatrices à partir des spécifications HAAD, nous avons utilisé VIS qui est un vérificateur basé sur la démonstration de modèle. Nous avons présenté d'abord la vérification du modèle générique des STFSM et ensuite la vérification des différentes compositions hiérarchiques. Dans les deux cas, nous avons défini des propriétés en CTL basées d'une part sur les caractéristiques de la machine générique et d'autre part, sur les équations des opérations de composition des STFSM. Les résultats de la vérification de la méthode pour la synthèse des machines observatrices nous permettent de confirmer que la méthode proposée pour la synthèse des machines observatrices à partir des spécifications HAAD satisfait nos attentes.

Nous avons utilisé les spécifications de protocole de lecture et d'écriture du bus PCI pour illustrer notre méthode. Nous avons analysé étape par étape les spécifications du protocole, produit leur modèle HAAD et déduit la machine observatrice en Verilog et en VHDL au niveau RTL. Nous avons ensuite défini un environnement de test pour la machine en VHDL, et simulé son comportement par la méthode classique de simulation en utilisant *ModelSim* de *Mentor Graphics*. Une fois encore les résultats de ces travaux ont confirmé la validité de la solution proposée pour la synthèse de la machine observatrice.

6.2. Travaux à venir

La méthode pour la synthèse de la machine observatrice est une méthode entièrement générique qui peut être automatisée. L'automatisation consistera à créer un compilateur (analyseur syntaxique et sémantique) du modèle HAAD de la spécification du système étudié et un générateur de code HDL (Verilog ou VHDL) qui permettra de produire un code source synthétisable de la machine observatrice pour une spécification HAAD au niveau RTL. Nous pensons qu'un investissement dans cette perspective va permettre un prototypage et une correction rapide de spécification d'un système numérique en étude. Dans cette étude, nous n'avons pas traité tous les aspects

HAAD pour la synthèse de la machine observatrice. Nous n'avons pas parlé, par exemple, de l'utilisation des procédures et des variables, des minimums et des maximums de contraintes ... Les travaux à venir permettront d'étendre la méthode proposée pour la synthèse des machines observatrices à tous les aspects de HAAD. Aussi, la machine observatrice peut être améliorée afin de produire des résultats beaucoup plus explicites avec des messages spécifiques lorsque le protocole du bus étudié est violé.

Références

[1] J. Hartmanis, "Symbolic Analysis of a Decomposition of Information Processisng Machine", Inform. Control, 3, Juin 1960, pp 154-178.

[2] M. Yeoli, "The Cascade Decomposition of Sequential Machines", IRE Trans. Electronic Computers, EC-10, Avril 1961, pp 587-592.

[3] A. Gill , "Introduction to the Theory of Finite-State Machines", McGraw-Hill Book Co., New-York, 1962.

[4] K. B. Krohn et J. L. Rhodes, "Algebraic Theory of Machines", Proc Symp. on Math. Theory of Automata, Polytechnic Press, Brooklyn, N. Y., 1962.

[5] J. Hartmanis, "Loop-Free Structure of Séquential Machines", Inform. and Control, 5, Mars 1962, pp 25-43.

[6] H. P. Zeiger, "Loop-Free Synthesis of Tinite-State Machines", MIT Ph.D. Thesis, Department of Electrical Engineering, Cambridge, Mass., Septembre 1964.

[7] J. Hartmanis et R. E. Strearns, "Algebraic Structure Theory of Sequential Machines", Prentice-Hall, Englewood Cliffs, N. J., 1966.

[8] U. Montanari, "Networks of Constraints: Fundamental Properties and Applications to Picture Processing", Information Science, Vol. 7, no. 2, 1974, pp 95-132.

[9] R. Marlett, "EBT: A Comprehensive Test Generation Technique for Highly Sequential Circuits", Proc. 15th Design Automation Conf., Juin 1978, pp 332-339.

[10] P. Rony, "Interfacing Fundamentals: Timing Diagram Conventions", Computer Design, Janvier 1980, pp 152-153.

[11] J. Quielle et J. Sifakis, "Specification and Verification of Concurrent Systems in CESAR", In Proceedings of the 5th International Symposium in Programming. 1981.

[12] E. M. Clarke et E. A. Emerson, "Synthesis of Synchronization Skeletons for Branching Time Temporal Logic", In Logic of Programs: Workshop, Yorktown Heights, N. Y., Mai 1981.

[13] M. Y. Vardi et P. Wolper, "Automata-Theoretic Techniques for Modal Logics of Programs", Journal of computer and System Science, Vol. 32, no. 2, avril 1986, pp 183-221.

[14] E. M. Clarke, E. A. Emerson et A. P. Sistla, "Automatic Verification of Finite-State Concurrent Systems using Temporal Logic Specifications", ACM Trans. Program. Long. Syst. 8, 2 Avril 1986, pp 244-263.

[15] T. P. Kelsey et K. K. Saluja, "Fast Test Generation for Sequential Circuits", Proc. Int'l Computer-Aided Design, Novembre 1986, pp 354-357.

[16] E. M. Clarke et O. Grumberg, "Research on Automatic Vérification of Finite-state Concurrent Systems", Annual Review of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1987, pp 2:269-290.

[17] "The VHDL Hardware Description Language", IEEE Standard 1076, 1987.

[18] A. Cohn, "A Proof of Correctness of the VIPER Microprocessors: The First Level", VLSI Specification, Verification and Synthesis, 1988.

[19] G. Borriello, "A New Interface Specification Methodology and its Application to Transducer Synthesis", PhD Thesis, University of California at Berkeley, 1988.

[20] P. Camurati et P. Prinetto. "Formal Verification of Hardware Correctness: Introduction and Survey of Current Research", Computer, Juin 1988, pp 8-19.

[21] H-K. T. Ma, S. Devadas, A. R. Newton et A. Sangiovanni-Vincentelli, "Test Generation for Sequential Circuits", IEEE Trans. on Computer-Aided Design, Octobre 1988, pp 1081-1093.

[22] W. Hunt Jr., "Microprocessor Design Verification", Journal of Automated Reasoning 5, 1989, pp 429-460.

[23] E. M. Clarke, E. D. Long et K. L. McMillan, "Compositional Model-Checking", Computer Society Press, Los Alamitos, California, 1989.

[24] K. T. Cheng et V. D. Agrawal, "Unified Methods for VLSI Simulation and Test Generation", Kluwer Academic Publishers, 1989.

[25] O. Coudert, C. Berthet et J. C. Madre, "Verification of Sequential Machines Based on Symbolic Execution", Proc of the Workshop on Automatic Vérification Methods for Finite State Systems, Juin 1989.

[26] R. E. Bryant, "Symbolic Simulation - Technique and Applications", DAC 1990, pp 517-521.

[27] J. R. Burch, E. M. Clarke, K. L. McMillan et D. L. Dill, "Sequential Circuit Verification using Symbolic Model Checking", Proc. of the 27th Design Automation Conf. (Orlando), Juin 1990, pp 46-51.

[28] Y. Wang, "Real-Time Behaviour of Asynchronous Agents", J.C.M. Baeten and J.W. Klop, editors, Proc. of the Conference on Theories of Concurrency: Unification and Extension, CONUR'90, volume 458 of Lecture Notes in Computer Science, Amsterdam, The Netherlands, 27-30 Août 1990, Springer-Verlag, pp 502-520.

[29] L. Jozwiak et J. Kolsteren, "An Efficient Method for the Sequential General Decomposition of Sequential Machines", Microprocessing & Microprogramming, Vol.31, 1991, pp 657-664.

[30] I. Pomeranz et S. M. Reddy, "Test Generation for Synchronous Sequential Circuits Using Multiple Observation Times", Fault-Tolerant Computing Symp., Juin 1991.

[31] A. Arnold, "Systèmes de Transitions finis et Sémantique des Processus Communicants", Masson, 1992.

[32] V. J. Guttag et J. J. Horning, "Larch: Languages and Tools for Formal Specification", editors, Springer-Verlag, 1993.

[33] K. L. McMillan, "Symbolic Model-Checking", Kluwer Academic Publishers, 1993.

[34] O. Lhomme, "Consistency Techniques for Numeric CSPs", Proceedings of the 13th International Joint Conference on Artificial Intelligence, 1993.

[35] S. T. Chanson et J Zhu, "A Unified Approach to Protocol Test Sequence Generation", Proceedings IEEE INFOCOM, San Francisco, Mars 1993, pp 106-114.

[36] K. Rath et S.D. Johnson, "Toward a Basis for Protocol Specification and Process Decomposition", Proc. IFIP Conf. on HDL (CHDL'93), Ottawa, Avril 1993, pp 260-281.

[37] B. Selic et G. Gullekson, "Real-time Object-Oriented Modeling", John Wiley, 1994.

[38] T. Ben Ismail, M. Abid et A. Jerraya, "COSMOS a Codesign Approach for communicating Systems", CODES'94, 1994.

[39] J.M. Autebert , "Théorie des langages et des automates." Masson, 1994.

[40] R. Alur et D. Dill, "A Theory of Timed Automata. Theoretical Computer Science", 1994, pp 126:183-235.

[41] H. Wong-Toi et D. L. Dill, "Approximations for Verifying Timing Properties", T. Rus et C. Rattray, editors, World Scientific Publishing, Theories and Experiences for Real-Time System Development, 1994, pp 177-204.

[42] J. Jaffar et M. Maher, "Constraint Logic Programming: A Survey", Journal of Logic Programming, Vol. 19/20, 1994, pp 503-581.

[43] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan et D. L. Dill, "Symbolic Model Checking from Sequential Circuit Verification", IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems, vol. 13, No. 4, Avril 1994.

[44] E. Cerny et B. Berkane, “Vérification des Chronogrammes Hérarchiques à l’Aide de CCS + contraintes”, Actes du Colloques BMW-94, Méthodes Mathématiques pour la Synthèse des Systèmes Informatiques, ACFAS’94, Montréal, Canada, Mai 1994, pp 211-224.

[45] J. R. Burch et D. L. Dill, "Automatic Verification of Pipelined Microprocessor Control," International Conference on Computer-Aided Verification, LNCS 818, Springer-Verlag, Juin 1994, pp 60-80.

[46] J. Joyce, G. Birtwistle et M. Gordon, "Proving a Computer Correct in Higher Order Logic", TR-100, Computer Lab., University of Cambridge, 1995.

[47] R. B. Jones, D. L. Dill et J. R. Burch, "Efficient Validity Checking for Processor Verification", Int’l Conference on Computer-Aided Design, 1995, pp 2-6.

[48] A. Silburt, I. Perryman, J. Bergeron, S. Nichols, M. Dufresne et G. Ward, "Accelerating Concurrent Hardware Design with Behavioural Modelling and System Simulation", Desisgn Automation Conference, 1995.

[49] R. Brayton et. al., "VIS: A System for Verification and Synthesis", Technical Report UCB/ERL M95, Electronics Research, 1995.

[50] C.-J. H. Seger et R. E. Bryant, "Formal Verification by Symbolic Evaluation of Partially-Ordered Trajectories", Formal Methods in System Design, Vol. 6, no. 2, Mars 1995, pp. 147-190.

[51] T. Ramalingom et A. Das, K. Thulairaman, "A Unified Test Case Generation Methode for EFSM Model Using Context Independent Unique Sequences", International Workshop on Protocol Test Systems, Evry, France, Septembre 1995.

[52] P. J. Ashenden, "The Designer’s Guide to VHDL", Morgan Koufmann, 1996.

[53] J. X. Su, D. L. Dill et C. Barrett, "Automatic Generation of Invariants in Processor Verification", FMCAD'96, Novembre 1996, pp 377-388.

[54] C. W. Barrett, D. L. Dill et J. R. Levitt, "Validity Checking for Combinations of Theories with Equality", FMCAD'96, Novembre 1996, pp 187-201.

[55] T. Kropf, "Formal Hardware Verification: Methods and Systems in Comparison", Lecture Notes in Computer Science. Eds.: G. Goos, J. Hartmanis, J.van Leeuwen. Vol. 1287, XII, 1997.

[56] C. Delgado Kloos et W. Damm (eds.), "Practical Methods for Hardware Design, Research Reports", Esprit, Project 6128, FORMAT, vol. 1, Springer-Verlag, 1997.

[57] T. Shanley et D. Anderson, "PCI System Architecture", Third Edition, MindShare, Inc., 1997.

[58] K. Delgado Kloos et W. Damm (eds.), "Practical Methods for Hardware Design, Research Reports", Esprit, Project 6128, FORMAT, Vol. 1, Springer-Verlag, 1997.

[59] E. Cerny, B. Berkane, P. Girodias et K. Khordoc, "Hierarchical Annotated Action Diagrams: An Interface-Oriented Specification and Verification Method", Kluwer Academic Publishers, 1998.

[60] M. G. Arnold, "Verilog Digital Computer Design: Algorithms into Hardware", Prentice Hall, 1999.

[61] M. G. Arnold, "Verilog Digital Computer Design - Algorithms into Hardware", Prentice Hall, 1999.

[62] F. Hessel, P. Coste, Ph. Le Marrec, N. E. Zergainoh, J. M. Daveau et A. A. JERRAYA, "Communication Interface Synthesis for Heterogeneous Specifications", RSP'99, Clearwater, USA, Juin 1999.

Annexe

```
// Description en Verilog de la connection du générateur (generator), de la machine générique (model) et
// du reflecteur (reflector) representee par la figure 21. Nous avons introduit des préfixes pour nommer les
// états des machines à états afin de faire la différence entre les états du modèle générique (s0), ceux du
// réflecteur (r0) et ceux du générateur (g0).

typedef enum {s0, s1, s_end, s_err} model_state;
typedef enum {g0, g1, g2} generator_state;
typedef enum {r0, r1} reflector_state;

// Module principal. Il est considéré comme une boîte noire dans laquelle tous les comportements du
// modèle générique de la machine observatrice, sont produits.

module main(clk);

 input clk;
 wire  go, ack_in, ack_out, Sys_end, Sys_err, clk, init;

 generator generator(clk, go, Sys_err, Sys_end, ack_out);
 model model(clk, Sys_end, Sys_err, ack_in, ack_out, go, init);
 reflector reflector(clk, Sys_err, Sys_end, ack_in);

endmodule

// Définition de la machine générateur (generator).

module generator(clk,go,Sys_err,Sys_end,ack_out);

 output go;
 input clk, Sys_end, Sys_err, ack_out;
 wire vl1, vl2;
 wire go, Sys_end, Sys_err, ack_out, clk;

 generator_state reg state;

initial
 begin
  state=g0;
 end

 assign vl1 = $NDset(0,1);
 assign vl2 = $NDset(0,1);

 assign go = (state == g1)?1:0;

always @(posedge clk)
 begin
  case (state)
   g0:
    begin
     if (vl1==1'b1)
      state=g0;
     else
      state=g1;
    end
```

```
      g1:
        begin
          if (ack_out==1'b1) state=g2;
          else state=g1;
        end
      g2:
        begin
          if (vl2==1'b1 && (Sys_end==1'b1  || Sys_err==1'b1)) state=g1;
          else if (vl2==1'b0 && (Sys_end==1'b1  || Sys_err==1'b1)) state=g0;
               else state=g2;
        end
    endcase
  end
endmodule

// Description du modèle générique (model) STFSM.

module model(clk, Sys_end, Sys_err, ack_in, ack_out, go);

  input clk;
  input go, ack_in;
  output ack_out, Sys_end, Sys_err;
  wire p1, p2;
  wire go, ack_in, ack_out, Sys_end, Sys_err, clk;

model_state reg state;

initial
  begin
    state=s0;
  end

  assign p1 = $ND(0,1);
  assign p2 = $ND(0,1);

 assign Sys_end = (state == s_end)?1:0;
 assign Sys_err = (state == s_err)?1:0;
 assign ack_out = ((state == s0 && go == 1'b1) || (state == s_err && go == 1'b1) || (state == s_end && go
== 1'b1))?1:0;

always @(posedge clk)
  begin
    case(state)
      s0: begin
          if (go == 1'b0)
            state = s0;
          else
            if (p1 == 1'b1)
              state = s1;
            else
              state = s_err;
          end

      s1: begin
```

```
      if (p1 == 1'b1 && p2 == 1'b0)
        state=s1;
      else
        if (p2==1'b1)
          state=s_end;
        else
          state=s_err;
      end

    s_end: begin
        if (go==1'b0 && ack_in==1'b1)
          state=s0;
        else
          if (go==1'b1 && ack_in==1'b1 && p1==1'b1)
            state=s1;
          else
            if (ack_in==1'b0 && p2==1'b1)
              state=s_end;
            else
              state=s_err;
      end

    s_err: begin
        if (go==1'b0 && ack_in==1'b1)
          state=s0;
        else
          if (go==1'b1 && ack_in==1'b1 && p1==1'b1)
            state=s1;
          else
            if (go==1'b1 && ack_in==1'b1 && p1==1'b0)
              state=s_err;
            else
              state=s_err;
      end
  endcase
 end
endmodule

// Description de la machine reflecteur "reflector".

module reflector(clk, Sys_err, Sys_end, ack_in);

 input clk, Sys_err, Sys_end;
 output ack_in;
 wire clk, Sys_err, Sys_end, ack_in;

 reflector_state reg state;

initial
 begin
  state=r0;
 end

 assign ack_in = (state == r1)?1:0;
```

```
always @(posedge clk)
 begin
  case (state)
  r0: if (Sys_end==1'b0 && Sys_err==1'b0) state=r0;
     else
     begin
      state=r1;
     end

  r1: begin
      state=r0;
     end
  endcase
 end
endmodule

#==========================================================================
# Propriétés CTL permettant de vérifier que le modele
# generique de la machine STFSM respecte les équations
# de la concaténation.
#==========================================================================
# Propriétés pour vérifier l'accessibilité de tous les états des machines
# generator, model et reflector.

AG(EF(generator.state = g0));
AG(EF(generator.state = g1));
AG(EF(generator.state = g2));

AG(EF(model.state = s0));
AG(EF(model.state = s1));
AG(EF(model.state = s_end));
AG(EF(model.state = s_err));

AG(EF(reflector.state = r0));
AG(EF(reflector.state = r1));

# Propriétés pour vérifier si les comportements de la machine générateur sont conformes aux
# spécifications et si toutes les transitions sont faisables.

AG((generator.state = g0) -> EX(generator.state = g0));
AG((generator.state = g0) -> EX((generator.state = g1) * (go = 1)));
AG(((generator.state = g1) * (ack_out = 0)) -> AX((generator.state = g1) * (go = 1)));
AG(((generator.state = g1) * (ack_out = 1)) -> AX((generator.state = g2) * (go = 0)));
AG(((generator.state = g2) * (Sys_err = 0) * (Sys_end = 0)) -> AX((generator.state = g2) * (go = 0)));
AG(((generator.state = g2) * ((Sys_err = 1) + (Sys_end = 1))) -> EX((generator.state = g0) * (go = 0)));
AG(((generator.state = g2) * ((Sys_err = 1) + (Sys_end = 1))) -> EX((generator.state = g1) * (go = 1)));

# Propriétés pour vérifier si les comportements de la machine générique STFSM
# sont conformes aux spécifications et si toutes les transitions sont
# faisables

AG(((model.state = s0) * (go = 0)) -> EX((model.state = s0) * (Sys_end = 0) * (Sys_err = 0)) * ack_out =
0);
```

```
AG(((model.state = s0) * (go = 1)) -> EX((model.state = s1) * (Sys_end = 0) * (Sys_err = 0)) * ack_out =
1);
AG((model.state = s0 * go = 1) -> EX((model.state = s_err) * (Sys_end = 0) * (Sys_err = 1)) * ack_out =
1);
AG((model.state = s1) -> EX((model.state = s1) * (Sys_end = 0) * (Sys_err = 0)) * ack_out = 0);
AG((model.state = s1) -> EX((model.state = s_end) * (Sys_end = 1) * (Sys_err = 0)) * ack_out = 0);
AG((model.state = s1) -> EX((model.state = s_err) * (Sys_end = 0) * (Sys_err = 1)) * ack_out = 0);
AG(((model.state = s_end) * (ack_in = 0)) -> EX((model.state = s_end) * (Sys_end = 1) * (Sys_err = 0)) *
ack_out = 0);
AG(((model.state = s_end) * (ack_in = 1) * (go = 1)) -> EX((model.state = s1) * (Sys_end = 0) * (Sys_err =
0)) * ack_out = 1);
AG(((model.state = s_end) * (ack_in = 1) * (go = 1)) -> EX(((model.state = s1) + (model.state = s_err)) *
(Sys_end = 0)) * ack_out = 1);
AG(((model.state = s_end) * (ack_in = 1)  * (go = 0)) -> EX((model.state = s0) * (Sys_end = 0) * (Sys_err
= 0)) * ack_out = 0);
AG(((model.state = s_end) * (ack_in = 0)) -> EX((model.state = s_err) * (Sys_end = 0) * (Sys_err = 1)) *
ack_out = 0);
AG(((model.state = s_end) * (ack_in = 0)) -> EX((model.state = s_err) * (Sys_end = 0) * (Sys_err = 1)) *
ack_out = 0);
AG(((model.state = s_end) * (ack_in = 1) * (go = 1)) -> EX((model.state = s_err) * (Sys_end = 0) *
(Sys_err = 1)) * ack_out = 1);
AG(((model.state = s_err) * (ack_in = 0)) -> EX((model.state = s_err) * (Sys_end = 0) * (Sys_err = 1)) *
ack_out = 0);
AG(((model.state = s_err) * (ack_in = 1) * (go = 1)) -> EX((model.state = s_err) * (Sys_end = 0) * (Sys_err
= 1)) * ack_out = 1);
AG(((model.state = s_err) * (ack_in = 1) * (go = 1)) -> EX((model.state = s1) * (Sys_end = 0) * (Sys_err =
0)) * ack_out = 1);
AG(((model.state = s_err) * (ack_in = 1) * (go = 0)) -> EX((model.state = s0) * (Sys_end = 0) * (Sys_err =
0)) * ack_out = 0);

# Propriétés pour vérifier si les comportements de la machine réflecteur sont conformes aux
# spécifications et si toutes ses transitions sont réalisables.

AG(((reflector.state = r0) * (Sys_err = 0) * (Sys_end = 0)) -> EX((reflector.state = r0) * (ack_in = 0)));
AG(((reflector.state = r0) * ((Sys_err = 1) + (Sys_end = 1))) -> EX((reflector.state = r1) * (ack_in = 1)));
AG((reflector.state = r1) -> AX(reflector.state = r0));

# Propriétés pour vérifier si la composition des machines à états respecte la
# spécification des équations de l'opération de composition cascade.

AG(((generator.state = g1) * (model.state = s0)) -> EX((generator.state = g2) * (model.state = s1)));
AG(((reflector.state = r1) * ((model.state = s_err) + (model.state = s_end))) -> EX((reflector.state = r0) *
((model.state = s1) + (model.state = s0))));

==============================================================================
# Resultat de la simulation par VIS du prototype de la machine à états
# observatrice représentée par la figure 19.
==============================================================================
vis> read_verilog concatenationprotocol.v
concatenationprotocol.v
vis> init_verify
vis> compute_reach -v 1
Computing reachable states using the iwls95 image computation method.
Printing Information about Image method: IWLS95
```

```
    Threshold Value of Bdd Size For Creating Clusters = 5000
         (Use "set image_cluster_size value " to set this to desired value)
    Verbosity = 0
         (Use "set image_verbosity value " to set this to desired value)
    W1 =  6 W2 = 1 W3 = 1 W4 = 2
         (Use "set image_W? value " to set these to desired values)
Shared size of transition relation for forward image computation is      123 BDD nodes (1    components)
*********************************
Reachability analysis results:
FSM depth =              8
reachable states =        81
BDD size =              32
analysis time =        0.02
reachability analysis =   complete

vis> simulate -n 20
# vis release 1.3 (compiled 20-May-99 at 9:47 PM)
# Network: main
# Simulation vectors have been randomly generated

.inputs  generator.vl1 generator.vl2 model.p1 model.p2
.latches generator.Next_state generator.state model.Next_state model.state reflector.Next_state
reflector.state
.outputs
.initial g0 g0 s0 s0 r0 r0

.start_vectors

# generator.vl1 generator.vl2 model.p1 model.p2 ; generator.Next_state generator.state model.Next_state
model.state reflector.Next_state reflector.state ;

0 0 1 0 ; g0 g0 s0    s0    r0 r0 ;
0 0 1 0 ; g1 g0 s0    s0    r0 r0 ;
0 1 1 1 ; g1 g1 s0    s0    r0 r0 ;
0 1 1 0 ; g2 g1 s1    s0    r0 r0 ;
1 1 0 1 ; g2 g2 s1    s1    r0 r0 ;
1 0 0 1 ; g2 g2 s_end s1    r0 r0 ;
0 1 0 0 ; g2 g2 s_end s_end r0 r0 ;
0 1 1 0 ; g1 g2 s_err s_end r1 r0 ;
1 1 1 0 ; g2 g1 s1    s_err r0 r1 ;
1 1 1 1 ; g2 g2 s1    s1    r0 r0 ;
1 1 0 1 ; g2 g2 s_end s1    r0 r0 ;
1 0 1 1 ; g2 g2 s_end s_end r0 r0 ;
0 1 0 1 ; g0 g2 s_end s_end r1 r0 ;
0 1 1 0 ; g1 g0 s_end s_end r1 r1 ;
1 1 0 0 ; g1 g1 s0    s_end r0 r1 ;
1 0 0 0 ; g2 g1 s_err s0    r0 r0 ;
0 1 1 0 ; g2 g2 s_err s_err r0 r0 ;
1 1 1 0 ; g1 g2 s_err s_err r1 r0 ;
0 1 1 0 ; g1 g1 s_err s_err r1 r1 ;
0 0 1 1 ; g1 g1 s_err s_err r1 r1 ;
# Final State : g1 g1 s_err s_err r1 r1

vis> model_check concatenationprotocol.ctl
```

```
# MC: formula passed --- AG(EF(generator.state = g0));
# MC: formula passed --- AG(EF(generator.state = g1));
# MC: formula passed --- AG(EF(generator.state = g2));

# MC: formula passed --- AG(EF(model.state = s0));
# MC: formula passed --- AG(EF(model.state = s1));
# MC: formula passed --- AG(EF(model.state = s_end));
# MC: formula passed --- AG(EF(model.state = s_err));

# MC: formula passed --- AG(EF(reflector.state = r0));
# MC: formula passed --- AG(EF(reflector.state = r1));

# Propriétés pour vérifier si les comportements de la machine générateur sont conformes aux
# spécifications et si toutes les transitions sont faisables.

# MC: formula passed --- AG((generator.state = g0) -> EX(generator.state = g0));
# MC: formula passed --- AG((generator.state = g0) -> EX((generator.state = g1) * (go = 1)));
# MC: formula passed --- AG(((generator.state = g1) * (ack_out = 0)) -> AX((generator.state = g1) * (go =
1)));
# MC: formula passed --- AG(((generator.state = g1) * (ack_out = 1)) -> AX((generator.state = g2) * (go =
0)));
# MC: formula passed --- AG(((generator.state = g2) * (Sys_err = 0) * (Sys_end = 0)) ->
AX((generator.state = g2) * (go = 0)));
# MC: formula passed --- AG(((generator.state = g2) * ((Sys_err = 1) + (Sys_end = 1))) ->
EX((generator.state = g0) * (go = 0)));
# MC: formula passed --- AG(((generator.state = g2) * ((Sys_err = 1) + (Sys_end = 1))) ->
EX((generator.state = g1) * (go = 1)));

# Propriétés pour vérifier si les comportements de la machine générique STFSM
# sont conformes aux spécifications et si toutes les transitions sont
# faisables

# MC: formula passed --- AG(((model.state = s0) * (go = 0)) -> EX((model.state = s0) * (Sys_end = 0) *
(Sys_err = 0)) * ack_out = 0);
# MC: formula passed --- AG(((model.state = s0) * (go = 1)) -> EX((model.state = s1) * (Sys_end = 0) *
(Sys_err = 0)) * ack_out = 1);
# MC: formula passed --- AG((model.state = s0 * go = 1) -> EX((model.state = s_err) * (Sys_end = 0) *
(Sys_err = 1)) * ack_out = 1);
# MC: formula passed --- AG((model.state = s1) -> EX((model.state = s1) * (Sys_end = 0) * (Sys_err = 0))
* ack_out = 0);
# MC: formula passed --- AG((model.state = s1) -> EX((model.state = s_end) * (Sys_end = 1) * (Sys_err =
0)) * ack_out = 0);
# MC: formula passed --- AG((model.state = s1) -> EX((model.state = s_err) * (Sys_end = 0) * (Sys_err =
1)) * ack_out = 0);
# MC: formula passed --- AG(((model.state = s_end) * (ack_in = 0)) -> EX((model.state = s_end) *
(Sys_end = 1) * (Sys_err = 0)) * ack_out = 0);
# MC: formula passed --- AG(((model.state = s_end) * (ack_in = 1) * (go = 1)) -> EX((model.state = s1) *
(Sys_end = 0) * (Sys_err = 0)) * ack_out = 1);
# MC: formula passed --- AG(((model.state = s_end) * (ack_in = 1) * (go = 1)) -> EX(((model.state = s1) +
(model.state = s_err)) * (Sys_end = 0)) * ack_out = 1);
# MC: formula passed --- AG(((model.state = s_end) * (ack_in = 1)  * (go = 0)) -> EX((model.state = s0) *
(Sys_end = 0) * (Sys_err = 0)) * ack_out = 0);
```

```
# MC: formula passed --- AG(((model.state = s_end) * (ack_in = 0)) -> EX((model.state = s_err) *
(Sys_end = 0) * (Sys_err = 1)) * ack_out = 0);
# MC: formula passed --- AG(((model.state = s_end) * (ack_in = 0)) -> EX((model.state = s_err) *
(Sys_end = 0) * (Sys_err = 1)) * ack_out = 0);
# MC: formula passed --- AG(((model.state = s_end) * (ack_in = 1) * (go = 1)) -> EX((model.state = s_err)
* (Sys_end = 0) * (Sys_err = 1)) * ack_out = 1);
# MC: formula passed --- AG(((model.state = s_err) * (ack_in = 0)) -> EX((model.state = s_err) * (Sys_end
= 0) * (Sys_err = 1)) * ack_out = 0);
# MC: formula passed --- AG(((model.state = s_err) * (ack_in = 1) * (go = 1)) -> EX((model.state = s_err) *
(Sys_end = 0) * (Sys_err = 1)) * ack_out = 1);
# MC: formula passed --- AG(((model.state = s_err) * (ack_in = 1) * (go = 1)) -> EX((model.state = s1) *
(Sys_end = 0) * (Sys_err = 0)) * ack_out = 1);
# MC: formula passed --- AG(((model.state = s_err) * (ack_in = 1) * (go = 0)) -> EX((model.state = s0) *
(Sys_end = 0) * (Sys_err = 0)) * ack_out = 0);

# Propriétés pour vérifier si les comportements de la machine réflecteur sont conformes aux
# spécifications et si toutes ses transitions sont réalisables.

# MC: formula passed --- AG(((reflector.state = r0) * (Sys_err = 0) * (Sys_end = 0)) -> EX((reflector.state
= r0) * (ack_in = 0)));
# MC: formula passed --- AG(((reflector.state = r0) * ((Sys_err = 1) + (Sys_end = 1))) ->
EX((reflector.state = r1) * (ack_in = 1)));
# MC: formula passed --- AG((reflector.state = r1) -> AX(reflector.state = r0));

# Propriétés pour vérifier si la composition des machines à états respecte la
# spécification des équations de l'opération de composition cascade.

# MC: formula passed --- AG(((generator.state = g1) * (model.state = s0)) -> EX((generator.state = g2) *
(model.state = s1)));
# MC: formula passed --- AG(((reflector.state = r1) * ((model.state = s_err) + (model.state = s_end))) ->
EX((reflector.state = r0) * ((model.state = s1) + (model.state = s0))));


//==========================================================================
// Code source de l'operation de composition en cascade.Nous avons instancié le
// modele generique deux fois. Ici nous mettons l'accent sur la composition elle même.
// Nous utilisons la description des machines à états reflecteur et générateur
// pour composer avec la machine résultante de la composition en cascade.
//==========================================================================

module main(clk);

  input clk;
  wire  go, ack_in, ack_out, Sys_end, Sys_err, init, clk;

  generator generator(clk, go, Sys_err, Sys_end, ack_out);
  cascade cascade(clk, Sys_end, Sys_err, ack_in, ack_out, go);
  reflector reflector(clk, Sys_err, Sys_end, ack_in);

endmodule

// Description de l'opération de concatenation.

module cascade(clk, Sys_end, Sys_err, ack_in, ack_out, go);
```

```
  wire clk, Sys_end, Sys_err, ack_in, ack_out, go;
  input clk, ack_in, go;
  output Sys_end, ack_out, Sys_err;

  wire Sys_end1, Sys_err1, ack_in1, ack_out1, go1;
  wire Sys_end2, Sys_err2, ack_in2, ack_out2, go2;

  assign go1 = go;
  assign go2 = (Sys_end1 == 1)?1:0;
  assign ack_in2 = ack_in;
  assign ack_in1 = ack_out2;
  assign ack_out = ack_out1;
  assign Sys_end = Sys_end2;
  assign Sys_err = Sys_err1 == 1 || Sys_err2 == 1;

model model1(clk, Sys_end1, Sys_err1, ack_in1, ack_out1, go1);
model model2(clk, Sys_end2, Sys_err2, ack_in2, ack_out2, go2);

endmodule

#==========================================================================
# Description des propriétés CTL pour la composition cascade
# Puisque nous avons vérifié le comportement général des transitions sur la
# machine générique, nous vérifions ici uniquement si les équation sur les
# signaux de composition sont respectées.
#==========================================================================

AG(ack_in2 = ack_in);
AG(ack_in1 = ack_out2);
AG(ack_out = ack_out1);
AG(go1 = go);
AG(go2 = Sys_end1);
AG(Sys_err = Sys_err1 + Sys_err2);
AG(Sys_end = Sys_end2);

//=========================================================================
// Code source de l'operation de composition parallèle.Nous avons instancié le
// modele generique deux fois. Ici nous mettons l'accent sur la composition elle même.
// Nous utilisons la description des machines à états reflecteur et générateur
// pour composer avec la machine résultante de la composition parallèle.
//=========================================================================

module main(clk);

  input clk;
  wire  go, ack_in, ack_out, Sys_end, Sys_err, clk;

  generator generator(clk, go, Sys_err, Sys_end, ack_out);
  parallele parallele(clk, go, Sys_err, Sys_end, ack_in, ack_out);
  reflector reflector(clk, Sys_err, Sys_end, ack_in);

endmodule
```

```
// Description de la composition parallèle de deux machines STFSM.

module parallele(clk, go, Sys_err, Sys_end, ack_in, ack_out);

  input clk;
  input go, ack_in;
  output ack_out, Sys_end, Sys_err;
  wire go, ack_in, ack_out, Sys_end, Sys_err, clk;
  wire Sys_end1, Sys_err1, ack_in1, ack_out1, go1;
  wire Sys_end2, Sys_err2, ack_in2, ack_out2, go2;

  assign go1 = (go == 1)?1:0;
  assign go2 = (go == 1)?1:0;
  assign ack_in2 = ack_in;
  assign ack_in1 = ack_in;
  assign ack_out = (ack_out1 == 1 && ack_out2 == 1)?1:0;
  assign Sys_end = Sys_end1 && Sys_end2;
  assign Sys_err = Sys_err1 || Sys_err2;
  assign init = init1 && init2;

  model model1(clk, Sys_end1, Sys_err1, ack_in1, ack_out1, go1);
  model model2(clk, Sys_end2, Sys_err2, ack_in2, ack_out2, go2);

endmodule

#===========================================================================
# Description des propriétés CTL pour la composition parallèle
# Définition identique à celle de l'opération cascade
#===========================================================================

AG(ack_in2 = ack_in);
AG(ack_in1 = ack_in);
AG(ack_out = ack_out1 * ack_out2);
AG(go1 = go);
AG(go2 = go);
AG(Sys_err = Sys_err1 + Sys_err2);
AG(Sys_end = Sys_end1 * Sys_end2);

//==========================================================================
// Code source de l'operation de composition de choix.Nous avons instancié le
// modele generique deux fois. Ici nous mettons l'accent sur la composition elle même.
// Nous utilisons la description des machines à états reflecteur et générateur
// pour composer avec la machine résultante de la composition choix.
//==========================================================================

module main(clk);

  input clk;
  wire  go, ack_in, ack_out, Sys_end, Sys_err, clk;

  generator generator(clk, go, Sys_err, Sys_end, ack_out);
  choix choix(clk, go, Sys_err, Sys_end, ack_in, ack_out);
  reflector reflector(clk, Sys_err, Sys_end, ack_in);
```

```
endmodule

// Description de la composition choix de deux machines STFSM.

module choix(clk, go, Sys_err, Sys_end, ack_in, ack_out);

  input clk;
  input go, ack_in;
  output ack_out, Sys_end, Sys_err;
  wire go, ack_in, ack_out, Sys_end, Sys_err, clk;
  wire Sys_end1, Sys_err1, ack_in1, ack_out1, go1;
  wire Sys_end2, Sys_err2, ack_in2, ack_out2, go2;

  assign go1 = (go == 1)?1:0;
  assign go2 = (go == 1)?1:0;
  assign ack_in2 = ack_in;
  assign ack_in1 = ack_in;
  assign ack_out = (ack_out1 == 1 || ack_out2 == 1)?1:0;
  assign Sys_end = Sys_end1 || Sys_end2;
  assign Sys_err = Sys_err1 && Sys_err2;

  model model1(clk, Sys_end1, Sys_err1, ack_in1, ack_out1, go1);
  model model2(clk, Sys_end2, Sys_err2, ack_in2, ack_out2, go2);

endmodule

#===========================================================================
# Description des propriétés CTL pour la composition choix. Définition identique à celle de l'opération
# cascade
#===========================================================================

AG(ack_in2 = ack_in);
AG(ack_in1 = ack_in);
AG(ack_out = ack_out1 + ack_out2);
AG(go1 = go);
AG(go2 = go);
AG(Sys_err = Sys_err1 * Sys_err2);
AG(Sys_end = Sys_end1 + Sys_end2);

//==========================================================================
// Code source de l'operation de composition de boucle.Nous avons instancié le
// modele generique deux fois. Ici nous mettons l'accent sur la composition elle même.
// Nous utilisons la description des machines à états reflecteur et générateur
// pour composer avec la machine résultante de la composition boucle.
//==========================================================================

module main(clk);

  input clk;
  wire  go, ack_in, ack_out, Sys_end, Sys_err, clk, exit;

  generator generator(clk, go, Sys_err, Sys_end, ack_out);
  boucle boucle(clk, Sys_end, Sys_err, ack_in, ack_out, go, exit);
  reflector reflector(clk, Sys_err, Sys_end, ack_in);
```

```
endmodule

// Description de la composition boucle d'une machine STFSM.

module boucle(clk, Sys_end, Sys_err, ack_in, ack_out, go, exit);

 wire clk, Sys_end, Sys_err, ack_in, ack_out, go, exit;

 input clk, ack_in, go, exit;
 output Sys_end, Sys_err, ack_out;

 wire  Sys_end1, Sys_err1, ack_in1, ack_out1, go1, exit;

 assign go1 = ((go == 1'b1 || Sys_end1 == 1'b1) && exit == 1'b0)?1:0;
 assign ack_out = (ack_out1 == 1'b1 && go == 1'b1)?1:0;
 assign Sys_end = (Sys_end1 == 1'b1 && exit == 1'b1)?1:0;
 assign Sys_err = (Sys_err1 == 1'b1)?1:0;
 assign ack_in1 = ((Sys_end1 == 1'b1 && exit == 1'b0) || (ack_in == 1'b1 && exit == 1'b1) || (Sys_err ==
1'b1 && ack_in == 1'b0))?1:0;

 model model1(clk, Sys_end1, Sys_err1, ack_in1, ack_out1, go1);

endmodule

#===========================================================================
# Description des propriétés CTL pour la composition choix
# Définition identique à celle de l'opération cascade
#===========================================================================

AG(ack_in1 = (Sys_end1 * !exit) + (ack_in * exit) + (Sys_err * ack_in));
AG(ack_out = ack_out1 + ack_out2);
AG(go1 = (go + Sys_end1) * !exit);
AG(Sys_err = Sys_err1);
AG(Sys_end = Sys_end1 * exit);

//==========================================================================
// Description de la machine génératrice à 1 cycle d'horloge. Nous l'appelons
// ici "model1cycle". Elle est composée avec les machines générateur et
// réflecteur afin de pouvoir vérifier les transitions des états et les
// sorties de cette machine.
//==========================================================================

module model1cycle(clk, Sys_end, Sys_err, ack_in, ack_out, go);

 input clk;
 input go, ack_in;
 output ack_out, Sys_end, Sys_err;
 wire p1, p2;
 wire go, ack_in, ack_out, Sys_end, Sys_err, clk;

model_state reg state;

initial
```

```
   begin
     state=s0;
   end

   assign p = $ND(0,1);

  assign Sys_end = (state == s_end)?1:0;
  assign Sys_err = (state == s_err)?1:0;
  assign ack_out = ((state == s0 && go == 1'b1) || (state == s_err && go == 1'b1) || (state == s_end && go
== 1'b1))?1:0;

 always @(posedge clk)
   begin
     case(state)
       s0: begin
           if (go == 1'b0)
             state = s0;
           else
             if (p == 1'b1)
               state = s_end;
             else
               state = s_err;
           end

       s_end: begin
               if (go==1'b0 && ack_in==1'b1)
                 state=s0;
               else
                 if (ack_in==1'b0 && p==1'b1)
                     state=s_end;
                 else
                     state=s_err;
             end

       s_err: begin
               if (go==1'b0 && ack_in==1'b1)
                 state=s0;
               else
                 if (go==1'b1 && ack_in==1'b1 && p==1'b1)
                   state=s_end;
                 else
                   state=s_err;
             end
     endcase
  end
endmodule

//==========================================================================
// Description de la composée de deux machines à 1 cycle d'horloge
//==========================================================================
module model1cyclecomp(clk, go, Sys_err, Sys_end, ack_in, ack_out);

  input clk;
  input go, ack_in;
```

```
 output ack_out, Sys_end, Sys_err;
 wire go, ack_in, ack_out, Sys_end, Sys_err, clk;
 wire Sys_end1, Sys_err1, ack_in1, ack_out1, go1;
 wire Sys_end2, Sys_err2, ack_in2, ack_out2, go2;
 wire v;   // Variable qui permet de rendre alternativement go1 et go2 actif

 initial
  v = 0;
 always @(posedge clk)
  begin
   assign v = !v;
  end

 assign go1 = go && v;
 assign go2 = go && !v
 assign ack_in2 = ack_in;
 assign ack_in1 = ack_in;
 assign ack_out = (ack_out1 == 1 || ack_out2 == 1)?1:0;
 assign Sys_end = Sys_end1 || Sys_end2;
 assign Sys_err = Sys_err1 || Sys_err2;

 model1cycle model1cycle1(clk, Sys_end1, Sys_err1, ack_in1, ack_out1, go1);
 model1cycle model1cycle2(clk, Sys_end2, Sys_err2, ack_in2, ack_out2, go2);

endmodule
```