Vérification des systèmes matériels numériques séquentiels synchrones : application du langage Lustre et de l'outil de vérification Lesar

Bachir Berkane

THESE

présenté par

Bachir BERKANE

pour obtenir le grade de DOCTEUR
de L'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

(arrêté ministériel du 30 mars 1992)

(Spécialité : Signal - Image - Parole)

Vérification des systèmes matériels numériques séquentiels synchrones

Application du langage Lustre et de l'outil de vérification Lesar

---

Date de soutenance : 2 octobre 1992

Composition du jury :	Président	G. Mazaré
	Rapporteurs	F. Anceau
		P. Prinetto
	Examinateurs	G. Thuau
		V. Olive

Thèse préparée au sein du Laboratoire de Génie Informatique

PRESIDENT DE L'INSTITUT
Monsieur Maurice RENAUD

INPG

Année 1991/1992

PROFESSEURS DES UNIVERSITES

BARIBAUD	Michel	ENSERG
BARRAUD	Alain	ENSIEG
BAUDELET	Bernard	ENSPG
BAUDIN	Gérard	UFR PGP
BEAUFILS	Jean-Pierre	ENSIEG/ILL
BOIS	Philippe	ENSHMG
BONNET	Guy	ENSPG
BOUYIER	Gérard	ENSERG
BRISSONNEAU	Pierre	ENSIEG
BRUNET	Yves	CUEFA
CAILLERIE	Denis	ENSHMG
CAVAIGNAC	Jean-François	ENSPG
CHARTIER	Germain	ENSPG
CHENEVIER	Pierre	ENSERG
CHERADAME	Hervé	UFR/PGP
CHERUY	Arlette	ENSIEG
CHOVET	Alain	ENSERG
COGNET	Gérard	ENSHMG
COLINET	Catherine	ENSEEG
COMMAULT	Christian	ENSIEG
CORNUT	Bruno	ENSIEG
COULOMB	Jean-Louis	ENSIEG
CROWLEY	James	ENSIMAG
DALARD	Francis	ENSEEG
DARVE	Félix	ENSHMG
DELLA DORA	Jean	ENSIMAG
DEPEY	Maurice	ENSERG
DEPORTES	Jacques	ENSPG
DEROO	Daniel	ENSEEG
DESRE	Pierre	ENSEEG
DIARD	Jean-Paul	ENSEEG
DOLMAZON	Jean-Marc	ENSERG
DURAND	Francis	ENSEEG
DURAND	Jean-Louis	ENSPG
FAUTRELLE	Yves	ENSHMG
FOGGIA	Albert	ENSIEG
FONLUPT	Jean	ENSIMAG
FOULARD	Claude	ENSIEG
GALERIE	Alain	ENSEEG
GANDINI	Alessandro	UFR/PGP
GAUBERT	Claude	ENSPG
GENTIL	Pierre	ENSERG
GENTIL	Sylviane	ENSIEG
GREVEN	Hélène	CUEFA
GUERIN	Bernard	ENSERG
GUYOT	Pierre	ENSEEG
IVANES	Marcel	ENSIEG
JALLUT	Christian	ENSEEG
JANOT	Marie-Thérese	ENSERG

JAUSSAUD	Pierre	ENSIEG
JOST	Rémy	ENSPG
JOUBERT	Jean-Claude	ENSPG
JOURDAIN	Geneviève	ENSIEG
KUENY	Jean-Louis	ENSHMG
LACHENAL	Dominique	UFR PGP
LACOUME	Jean-Louis	ENSIEG
LADET	Pierre	ENSIEG
LESIEUR	Marcel	ENSHMG
LESPINARD	Georges	ENSHMG
LIENARD	Joël	ENSIEG
LONGEQUEUE	Jean-Pierre	ENSPG
LORET	Benjamin	ENSHMG
LOUCHET	François	ENSEEG
LUCAZEAU	Guy	ENSEEG
LUX	Augustin	ENSIMAG
MASSE	Philippe	ENSIEG
MASSELOT	Christian	ENSIEG
MAZARE	Guy	ENSIMAG
MICHEL	Gérard	ENSIMAG
MOHR	Roger	ENSIMAG
MOREAU	René	ENSHMG
MORET	Roger	ENSIEG
MOSSIERE	Jacques	ENSIMAG
OBLED	Charles	ENSHMG
OZIL	Patrick	ENSEEG
PANANAKAKIS	Georges	ENSERG
PAULEAU	Yves	ENSEEG
PERRET	Robert	ENSIEG
PIAU	Jean-Michel	ENSHMG
PIC	Etienne	ENSERG
PLATEAU	Brigitte	ENSIMAG
POUPOT	Christian	ENSERG
RAMEAU	Jean-Jacques	ENSEEG
REINISCH	Raymond	ENSPG
RENAUD	Maurice	UFR/PGP
ROBERT	François	ENSIMAG
ROSSIGNOL	Michel	ENSPG
ROYE	Daniel	ENSIEG
SABONNADIERE	Jean-Claude	ENSIEG
SAGUET	Pierre	ENSERG
SAUCIER	Gabriéle	ENSIMAG
SCHLENKER	Claire	ENSPG
SCHLENKER	Michel	ENSPG
SILVY	Jacques	UFR/PGP
SIRIEYS	Pierre	ENSHMG
SOHM	Jean-Claude	ENSEEG
SOLER	Jean-Louis	ENSIMAG
SOUQUET	Jean-Louis	ENSEEG
TICHKIEWITCH	Serge	ENSHMG
TROMPETTE	Philippe	ENSHMG
TRYSTRAM	Denis	ENSGI
VEILLON	Gérard	ENSIMAG
VERJUS	Jean-Pierre	ENSIMAG
VINCENT	Henri	ENSPG
ZADWORNY	François	ENSERG

SITUATION PARTICULIERE

PROFESSEURS D'UNIVERSITE

DETACHEMENT

ENSERG	BLIMAN	Samuel	Mutation	
ENSPG	BLOCH	Daniel	Recteur	21/12/93
ENSIMAG	LATOMBE	Jean-Claude	Detachement	01/05/93
ENSHMG	PIERRARD	Jean-Marie	Disponible	

RETRAITE

ENSEEG	BONNETAIN	Lucien

DIRECTEURS DE RECHERCHE CNRS

ABELLO	Louis
ALDEBERT	Pierre
ALEMANY	Antoine
ALLIBERT	Colette
ALLIBERT	Michel
ANSARA	Ibrahim
ARMAND	Michel
AUDIER	Marc
AUGOYARD	Jean-François
AVIGNON	Michel
BERNARD	Claude
BINDER	Gilbert
BLAISING	Jean-Jacques
BONNET	Roland
BORNARD	Guy
BOUCHERLE	Jean-Xavier
CAILLET	Marcel
CARRE	René
CHASSERY	Jean-Marc
CHATILLON	Christian
CIBERT	Joël
CLERMONT	Jean-Robert
COURTOIS	Bernard
CRIQUI	Patrick
CRISTOLOVEANU	Sorin
DAVID	René
DION	Jean-Michel
DOUSSIERE	Jacques
DRIOLE	Jean
DUCHET	Pierre
DUGARD	Luc
DURAND	Robert
ESCUDIER	Pierre
EUSTATHOPOULOS	Nicolas
FINON	Dominique
FRUCHARD	Robert
GARNIER	Marcel
GIROD	Jacques
GLANGEAUD	François
GUELIN	Pierre
HOPFINGER	Emil
JORRAND	Philippe
JOUD	Jean-Charles
KAMARINOS	Georges
KLEITZ	Michel
KOFMAN	Walter
LACROIX	Claudine
LANDAU	Ioan
LAULHERE	Jean-Pierre
LEGRAND	Michel
LEJEUNE	Gérard
LEPROVOST	Christian
MADAR	Roland
MARTIN	Jean-Marie
MERMET	Jean

MEUNIER	Gérard
MICHEL	Jean-Marie
NAYROLLES	Bernard
PASTUREL	Alain
PEUZIN	Jean-Claude
PHAM	Antoine
PIAU	Monique
PIQUE	Jean-Paul
POINSIGNON	Christiane
PREJEAN	Jean-Jacques
RENOUARD	Dominique
SENATEUR	Jean-Pierre
SIFAKIS	Joseph
SIMON	Jean-Paul
SUERY	Michel
TEODOSIU	Christian
VACHAUD	Georges
VAUCLIN	Michel
WACK	Bernard
YAYARI	Ali-Reza
YONNET	Jean-Paul

PERSONNES AYANT OBTENU LE DIPLOME

D'HABILITATION A DIRIGER DES RECHERCHES

BALESTRA	Francis
BALME	Louis
BECKER	Monique
BIGEON	Jean
BINDER	Zdeneck
BOE	Louis-Jean
CANUDAS DE WIT	Carlos
CHOLLET	Jean-Pierre
COEY	Jean-Pierre
CORNUEJOLS	Gerard
COURNIL	Michel
CRASTES DE PAULET	Michel
DALLERY	Yves
DESCOTES-GENON	Bernard
DUGARD	Luc
DURAND	Madeleine
FERRIEUX	Jean-Paul
FEUILLET	René
FREIN	Yannick
GAUTHIER	Jean-Paul
GHIBAUDO	Gérard
GUILLEMOT	Nadine
GUYOT	Alain
HAMAR	Sylviane
HAMAR	Roger
HORAUD	Patrice
JACQUET	Paul
LATOMBE	Claudine
LE HUY	Hoang
LE GORREC	Bernard
LOZANO-LEAL	Rogelio
MACOYSCHI	Mihaïl
MAHEY	Philippe
METAIS	Olivier
MONMUSSON-PICQ	Georgette
MORY	Mathieu
MULLER	Jean
MULLER	Jean-Michel
NGUYEN TRONG	Bernadette
NIEZ	Jean-Jacques
PERRIER	Pascal
PLA	Fernand
RECHENMANN	François
ROGNON	Jean-Pierre
ROUGER	Jean
ROUX	Jean-Claude
TCHUENT	Maurice

PERSONNES AYANT OBTENU LE DIPLOME

DE DOCTEUR D'ETAT INPG

ABDEL-RAZEK	Adel
AKSAS	Haris
ALLA	Hassane
AMER	Ahmed
ANCELLE	Bernard
ANGENIEUX	Gilbert
ATMANI	Hamid
AYEDI	Hassine Feri
A.BADR	Osman
BACHIR	Aziz
BALANZAT	Emmanuel
BALTER	Roland
BARDEL	Robert
BARRAL	Gérard
BAUDON	Yves
BAUSSAND	Patrick
BEAUX	Jacques
BEGUINOT	Jean
BELLISSENT née FUNEZ	Marie-Claire
BELLON	Catherine
BEN RAIS	Abdejettah
BERGER-SABBATEL	Gilles
BERNACHE-ASSOLANT	Didier
BEROYAL	Abderrahmane
BERTHOD	Jacques
BILLARD	Dominique
BLANC épouse FOULETIER	Mireille
BOCHU	Bernard
BOJO	Gilles
BOKSENBAUM	Claude
BOLOPION	Alain
BONNARD	Bernard
BORRIONE	Dominique
BOUCHACOURT	Michel
BRINI	Jean
BRION	Bernard
CAIRE	Jean-Pierre
CAMEL	Denis
CAPERAN	Philippe
CAPLAIN	Michel
CAPOLINO	Gérard
CASPI	Paul
CHAN-TUNG	Nam
CHASSANDE	Jean-Pierre
CHATAIN	Dominique
CHEHIKIAN	Alain
CHIRAMELLA	Yves
CHILO	Jean
CHUPIN	Jean-Claude
COLONNA	Jean-François
COMITI	Jacques
CORDET	Christian
COUDURIER	Lucien

COUTAZ	Jean-Louis
CREUTIN	Jean-Dominique
DAO	Trongtich
DARONDEAU	Philippe
DAVID	Bertrand
DE LA SEN	Manuel
DELACHAUME	Jean-Claude
DENAT	André
DESCHIZEAUX née CHERUY	Marie-Noëlle
DIJON	Jean
DOREMUS	Pierre
DUPEUX	Michel
EL ADHAM	Karim
EL OMAR	Fovaz
EL-HENNAWY	Adel
ETAY	Jacqueline
FABRE épouse MAXIMOVITCH	Suzanne
FAURE-BONTE épouse MARET	Mireille
FAVIER	Denis
FAVIER	Jean-Jacques
FELIACHI	Movloud
FERYAL	Haj Hassan
FLANDRIN	Patrick
FOREST	Bernard
FORESTIER	Michel
FOSTER	Panayotis
FRANC	Jean-Pierre
GADELLE	Patrice
GARDAN	Yvon
GENIN	Jacques
GERYASON	Georges
GILORMINI	Pierre
GINOUX	Jean-Louis
GOUMIRI	Louis
GROC	Bernard
GROSJEAN	André
GUEDON	Jean-Yves
GUERIN	Jean-Claude
GUESSOUS	Anas
GUIBOUD-RIBAUD	Serge
HALBWACHS	Nicolas
HAMMOURI	Hassan
HEDEIROS SILIVEIRA	Hamilton
HERAULT	Jeanny
HONER	Claude
HUECKEL	Tomasz
IGNAT	Michel
ILIADIS	Athananios
JANIN	Gérard
JERRAYA	Ahmed Amine
JUTTEN	Christian
KAHIL	Hassan
KHUONGQUANG	Dong
KILLIS	Andreas
KONE	Ali
LABEAU	Michel
LACAZE	Alain
LACROIX	Jean-Claude
LANG	Jean-Claude
LATHUILLERE	Chantal

LATY	Pierre
LAUGIER	Christian
LE CADRE	Jean-Pierre
LE GARDEYR	René
LE NEST	Jean-François
LE THIESSE	Jean-Claude
LEMAIGNAN	Clement
LEMUET	Daniel
LEVEQUE	Jean-Luc
LONDICHE	Henry
L'HERITIER	Philippe
MAGNIN	Thierry
MAISON	François
MAMWI	Abdullah
MANTEL épouse SIEBERT	Elisabeth
MARCON	Guy
MARTINEZ	Francis
MARTIN-GARIN	Lionel
MASSE	Dominique
MAZER	Emmanuel
MERCKEL	Gérard
MEUNIER	Jean
MILI	Ali
MOALLA	Mohamed
MODE	Jean-Michel
MONLLOR	Christian
MONTELLA	Claude
MORET	Frédéric
MRAYATI	Mohammed
M'SAAD	Mohammed
M'SIRDI	Kouider Nace
NEPOMIASTCHY	Pierre
NGUYEN	Trong Khoi
NGUYEN-XUAN-DANG	Michel
ORANIER	Bernard
ORTEGA MARTINEZ	Roméo
PAIDASSI	Serge
PASSERONE	Alberto
PEGON	Pierre
PIJOLAT	Christophe
POGGI	Yves
POIGNET	Jean-Claude
PONS	Michel
POU	Tong Eck
RAFINEJAD	Paiviz
RAGAIE	Harie Fikri
RAHAL	Salah
RAMA SEABRA SANTOS	Fernando
RAYAINE	Denis
RAZBAN-HAGHIGHI	Tchanguiz
RAZZOUK	Micham
REGAZZONI	Gilles
RIQUET	Jean-Pierre
ROBACH	Chantal
ROBERT	Yves
ROGEZ	Jacques
ROHMER	Jean
ROUSSEL	Claude
SAAD	Abdallah
SAAD	Youcef

SABRY	Mohamed Nabi
SALON	Marie-Christine
SAUBAT épouse MARCUS	Bernadette
SCHMITT	Jean-Hubert
SCHOELLKOPF	Jean-Pierre
SCHOLL	Michel
SCHOLL	Pierre-Claude
SCHOULER	Edmond
SCHWARTZ	Jean-Luc
SEGUIN	Jean
SIWY	Jacques
SKALLI	Abdellatif
SKALLI HOUSSEYNI	Abdelali
SOUCHON	Alain
SUETRY	Jean
TALLAJ	Nizar
TEDJAR	Farouk
TEDJINI	Smail
TEYSSANDIER	Francis
THEVENODFOSSE	Pascale
TMAR	Mohamed
TRIOLLIER	Michel
TUFFELIT	Denis
TZIRITAS	Georges
VALLIN	Didier
VELAZCO	Raoul
VERDILLON	André
VERMANDE	Alain
VIKTOROVITCH	Pierre
VITRANT	Guy
WEISS	François
YAZAMI	Rachid

A ma famille et,

A certains

Remerciements

Je tiens d'abord à formuler ma profonde reconnaissance à Mme **G. Thuau** , mon directeur de thèse, pour son suivi de mes travaux de recherche et pour son aide dans la rédaction de ma thèse.

Je remercie aussi

M. **G. Mazaré**, de m'avoir fait l'honneur de présider le jury.

MM. **J.C Madre** et **N. Halbwachs**, pour leurs précieuses remarques et suggestions.

MM. **F. Anceau** et **P. Prinetto**, d'avoir accepté d'être rapporteurs de cette thèse.

M. **P. Caspi**, pour nos passionnantes discussions.

M. **V. Olive** qui a bien voulu s'intéresser à ce travail en acceptant de participer à ce jury.

M. **J. Sifakis**, pour m'avoir accepté dans son équipe.

Sommaire

Introduction Générale

Problématique

La vérification fonctionnelle d'un système matériel (un circuit, une carte, un ensemble de cartes communicantes) consiste à s'assurer que le système réalise un certain nombre de fonctionnalités spécifiées. Cette vérification est un moment crucial lors du processus de conception. En effet, un produit défectueux nécessite la remise en cause des phases de son développement. Plus une erreur est détectée tard dans le processus de conception, plus sa correction est coûteuse. Il est donc nécessaire de détecter toutes les erreurs avant la fabrication : *valider au plus tôt dans le processus de conception.*

La technique de vérification fonctionnelle la plus utilisée de nos jours est la simulation. La simulation d'un système matériel séquentiel sera exhaustive si elle est faite pour toutes les séquences d'entrées possibles du système. C'est un problème qui se heurte à l'explosion combinatoire du nombre de cas à simuler. Ainsi, la simulation exhaustive n'est pratiquement jamais réalisable. C'est pourquoi d'autres méthodes de validation, comme la *vérification formelle*, se développent notamment en milieu industriel [Lev 91, CCBM 91, DM 92] pour assurer le zéro-défaut. La vérification formelle vise à montrer, *au sens mathématique du terme*, la bonne fonctionnalité d'un système matériel en démontrant que la description du système est exempte d'erreurs.

Dans le domaine de la vérification des systèmes matériels combinatoires, des outils performants ont été réalisés. Nous citons en particulier l'outil PRIAM [Mad 90] qui a été intégré dans le système de CAO (conception assistée par ordinateur) de BULL. Ce système de vérification est

fondé sur une technique d'exécution symbolique des programmes décrivant les systèmes combinatoires. Ces programmes ont d'abord été écrits dans un langage de description du matériel appelé LDS, puis dans le langage VHDL [VHDL 87]. Outre la détection d'erreurs de conception, PRIAM permet de réaliser un diagnostic.

Cependant, dans le domaine des systèmes matériels séquentiels (systèmes ayant la fonction mémoire) des efforts restent encore à faire pour la réalisation de systèmes pouvant être réellement intégrés dans une chaîne de CAO. Ceci malgré de nombreuses recherches qui ont mené à la réalisation d'outils de vérification automatique. Parmi ces systèmes, nous citons en particulier l'outil SIAM réalisé au centre de recherche BULL [Cou 91] et l'outil COSPAN réalisé aux laboratoires de AT&T Bell [HK 89, GK 91, CDCT 92]. Les auteurs de ces systèmes en abordant les techniques de base de la vérification ne se sont pas préoccupés des problèmes pratiques qu'un concepteur de systèmes matériels aura à affronter pour exprimer et résoudre les problèmes de vérification. En particulier, ils ne se sont pas intéressés au langage de description des fonctionnalités à vérifier.

Objectif de l'étude

Notre domaine d'étude concerne les systèmes matériels numériques séquentiels. Parmi ces systèmes, on distingue les systèmes synchrones et les systèmes asynchrones. En conception, l'approche synchrone est souvent adoptée pour s'affranchir du problème de stabilité qui peut engendrer le non déterminisme des systèmes conçus.

L'étude réalisée dans ce document porte sur la définition d'une approche de vérification permettant d'exprimer et de résoudre les problèmes de vérification des systèmes matériels séquentiels synchrones d'une manière simple. Nous proposons une approche unifiée pour la description et la vérification fonctionnelle de ces systèmes, en nous attachant aux problèmes réels qu'un concepteur aura à résoudre. Cette approche est fondée sur le modèle de machines d'états finis. Une application du langage Lustre et de l'outil de vérification Lesar sera étudiée pour valider l'approche et mettre en relief un réel environnement de vérification des systèmes matériels synchrones qui pourra être intégré dans une chaîne de CAO.

Plan de lecture

Cette thèse est composée de trois parties. La première partie présente les caractéristiques des systèmes matériels séquentiels synchrones et fait un état de l'art de la vérification formelle de ces systèmes. Les principales caractéristiques des systèmes matériels numériques séquentiels synchrones sont données dans le chapitre 1. Dans le chapitre 2, les différentes approches adoptées par différentes équipes de recherche pour la description et la vérification des systèmes matériels sont présentées et discutées.

La validation fonctionnelle d'un système matériel consiste à vérifier le système vis-à-vis de son fonctionnement attendu. Il existe deux façons de spécifier ce fonctionnement attendu. D'une part, la spécification peut être donnée sous forme d'une description fonctionnelle complète : pour toute séquence d'entrée valide, quelle séquence le système doit-il fournir en sortie? D'autre part, l'expression de cette spécification peut être donnée sous forme d'un ensemble de propriétés temporelles critiques (le caractère critique d'une propriété est liée à la sévérité de sa perte). Par exemple, un arbitre de bus ne doit pas affecter le bus à deux unités de traitement en même temps. Ces deux façons de spécifier les systèmes matériels ont donné lieu à deux problèmes de vérification. Le premier problème consiste à vérifier la description d'un système matériel par rapport à une description fonctionnelle complète, ce que nous désignerons par *vérification de conformité*. Le deuxième problème consiste à vérifier que le système matériel satisfait des propriétés cruciales de fonctionnement. Nous désignerons ce deuxième problème par *vérification de propriétés*.

La deuxième partie de ce document présente les fondements théoriques d'une approche de vérification unifiée pour résoudre les deux problèmes de vérification de conformité et de propriétés. Nous montrons dans cette partie que tout problème de vérification se ramène à définir une machine de vérification d'états finis sur laquelle la vérification sera réalisée par simple parcours du graphe d'états et de transitions. Nous analysons d'abord dans le chapitre 3 les propriétés temporelles relatives aux comportements des systèmes matériels. Les chapitres 4 et 5 abordent respectivement les problèmes de vérification de propriétés et de conformité. Finalement, le problème général de la vérification d'un système matériel sous un environnement de fonctionnement est étudié dans le chapitre 6.

Dans la troisième partie de ce document, nous étudions l'application du langage Lustre et de l'outil de vérification Lesar. Le chapitre 7 présente le langage Lustre. Le chapitre 8 montre que tout programme Lustre opérant sur des flots de booléens dénote une machine d'états finis. Ceci va nous permettre de définir, dans le chapitre 9, un cadre unifié pour la description et la vérification des systèmes matériels séquentiels synchrones. La vérification est alors réalisée automatiquement par l'outil de preuve Lesar présenté dans le chapitre 10. Enfin, dans le chapitre 11, nous présentons diverses méthodes permettant d'aider à réaliser un diagnostic.

Nous concluons ce travail en mettant en relief un réel environnement de vérification fonctionnelle des systèmes matériels séquentiels synchrones pouvant être intégré dans une chaîne de CAO. Cet environnement sera fondé sur un même langage de description des systèmes matériels et de spécification des fonctionnalités à vérifier.

Partie I

Les systèmes matériels numériques séquentiels synchrones : caractéristiques et validation

Chapitre 1

Les systèmes matériels séquentiels synchrones : SMSS

Un système matériel séquentiel, à la différence d'un système combinatoire, est un système dont l'état dépend, outre de ses entrées présentes, des combinaisons précédentes de ces entrées. Il s'agit d'un système possédant une fonction de mémoire. L'additionneur série est un exemple de circuit séquentiel. En effet, le résultat de l'addition de deux bits à l'instant présent dépend de la retenue précédente. L'état d'un système séquentiel est caractérisé par un ensemble de variables nommées variables d'états correspondant aux différentes mémoires de ce système. Les entrées possibles d'un système séquentiel et les sorties résultantes sont en nombre fini. L'observation du système séquentiel permet de définir les ensembles d'entrées E et de sorties S, du fait qu'ils sont externes à celui-ci. Par contre, l'ensemble des états du système ne peut pas être défini par simple observation. Notons qu'un système combinatoire pur peut être considéré comme un système séquentiel possédant un état unique.

On distingue deux catégories de systèmes séquentiels. (1) Les *systèmes asynchrones* dont les sorties évoluent à chaque changement des variables d'entrées (autrement dit le séquencement et le temps sont indépendants). (2) Les *systèmes synchrones* pour lesquels l'évolution des sorties et de l'état est conditionnée par une commande spécifique appelée horloge. Le signal d'horloge définit une dépendance entre le séquencement et le temps. Comme il a été montré [MC 80], ce signal d'horloge a deux buts différents ; d'une part *logique* : définition des instants privilégiés de transition, d'autre part *physique* : la durée entre deux pas d'horloge représente le retard de propagation de l'information. Dans un système séquentiel, le résultat présent est utilisé pour établir le résultat suivant (cas de l'additionneur série). Par conséquent, un certain nombre de signaux doivent être rebouclés à travers des cellules de mémorisation.

Lorsque ce rebouclage est asynchrone, il peut engendrer des oscillations en sortie, c'est-à-dire un comportement non déterministe. La synchronisation devient alors nécessaire, elle assure un comportement bien défini (*déterministe*), et aura pour conséquence une simplification des processus de conception et de vérification.

Dans ce chapitre, nous donnons tout d'abord les caractéristiques d'un système matériel séquentiel synchrone ou SMSS. Ensuite, les propriétés des systèmes résultant d'une interconnexion d'autres systèmes séquentiels sont présentées.

1.1 Caractéristiques d'un SMSS

Nous donnons d'abord, dans cette section, le modèle mathématique caractérisant un système matériel séquentiel dont les éléments de mémorisation évoluent sur une horloge globale. Puis, les principales techniques de représentation d'un système séquentiel sont rappelées.

1.1.1 Modèle mathématique

Tout système matériel séquentiel ayant *m* lignes d'entrées, *p* lignes de sorties et *n* éléments de mémorisation synchrones peut être caractérisé par un 6-uplet $(Q, E, S, \delta, \lambda, q_0)$ [Boo 67, Koh 70, Sze 68] où :

- Q est l'ensemble fini d'états du système. Chaque état est défini par une configuration donnée des *n* éléments de mémorisation. Le nombre d'états de l'ensemble Q est donc borné par 2^n.

- q_0 est l'état initial du système, défini par les valeurs initiales des *n* éléments de mémorisation.

- E est l'ensemble fini des 2^m valeurs d'entrée possibles, appelées symboles d'entrée. L'ensemble des séquences d'entrée d'un vocabulaire E est appelé *langage d'entrée* noté L_e.

- S est l'ensemble fini des 2^p valeurs de sortie possibles, appelées symboles de sortie. L'ensemble des séquences de sortie d'un vocabulaire S est appelé *langage de sortie* noté L_s.

- δ est la *fonction* de transition, définie de $E \times Q$ dans Q.

• λ est la *fonction* de sortie. Dans le cas où les sorties d'un système dépendent des entrées présentes et de son état présent, la fonction λ est définie de E × Q dans S. Le modèle est appelé *modèle de Mealy* . Si les sorties du système dépendent uniquement de son état présent (λ est définie de Q dans S), le modèle est alors un *modèle de Moore*. Les figures 1.1a et 1.1b illustrent les caractéristiques principales de ces deux modèles. Notons que le passage d'un modèle à l'autre est possible. Le passage d'une représentation de Moore vers une représentation de Mealy se fait sans aucune augmentation du nombre d'états. Alors que le nombre d'états de la machine de Moore équivalente à une machine de Mealy est exponentiel en nombre de lignes de sortie.

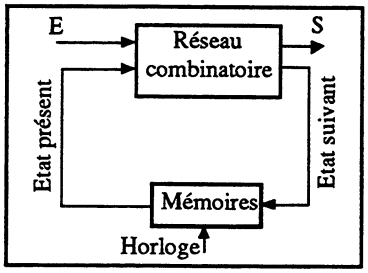


Figure 1.1a. Modèle de Mealy

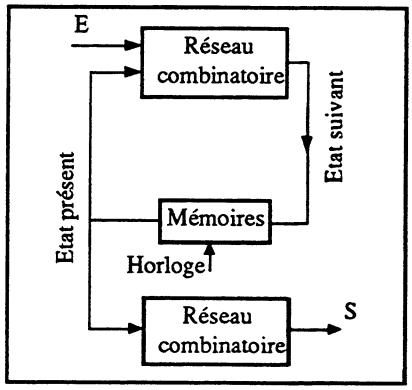


Figure 1.1b. Modèle de Moore

1.1.2 Représentation d'un SMSS

On distingue principalement quatre techniques de représentation des SMSS [Boo 67, Kre 78]. Ces techniques sont : les tables de transitions, les matrices de transitions, les listes de transitions et les graphes d'états (ou graphes de transitions). Les deux premières techniques de représentation donnent les propriétés des fonctions de transitions et de sorties sous forme de tables ou de matrices. Une liste de transitions est l'ensemble des 4-uplets décrivant les transitions de la machine où chaque 4-uplet est défini comme suit : (valeurs des entrées, état courant, état suivant, valeurs des sorties).

Un graphe d'états est une représentation graphique permettant de connaître l'évolution du système pour toute séquence d'entrée appliquée à partir d'un état initial. Dans ce graphe, un état est représenté par un cercle contenant le numéro de l'état. Pour tout couple d'état (q_i, q_j) un arc orienté connecte l'état q_i à q_j, si et seulement si, il existe un vecteur d'entrée e_i permettant la transition, autrement dit lorsque $\delta(e_i, q_i) = q_j$. Dans ces conditions, l'arc sera étiqueté par la valeur du vecteur e_i et les sorties s_i du système correspondantes (c'est-à-dire : e_i / $[s_i = \lambda(e_i, q_i)]$). Toute variable d'entrée a pour domaine $\{0, 1, \phi\}$ où ϕ désigne la valeur

indéterminée (en englais, don't care), elle sera désignée par "–" sur les graphes d'états. Dans le cas d'un modèle de Moore, la valeur de la fonction de sortie sera associée au cercle de l'état du fait que les sorties dépendent uniquement de l'état présent. Le graphe d'états est la technique de représentation adoptée dans tout ce qui suit.

La figure 1.2b donne le graphe d'états du circuit détecteur de front (figure 1.2a). Ce circuit retourne sur sa sortie la valeur booléenne 1 si, et seulement si, la valeur de l'entrée courante est la valeur booléenne 1 et la valeur de l'entrée précédente est la valeur booléenne 0. Ce circuit possède un état initial q_0 à partir duquel on peut atteindre deux états distincts q_1 et q_2 selon la valeur de l'entrée dans l'ensemble E = {0,1}. Nous supposons que la valeur initiale de la variable d'état associée à l'état initial q_0 (valeur initiale de la mémoire du circuit) vaut la valeur booléenne 0.

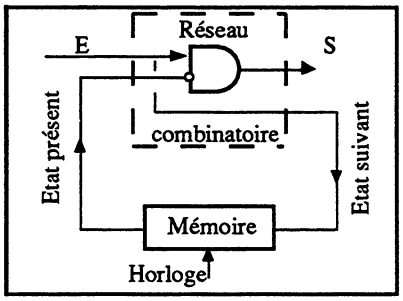


Figure 1.2a. Circuit détecteur de front

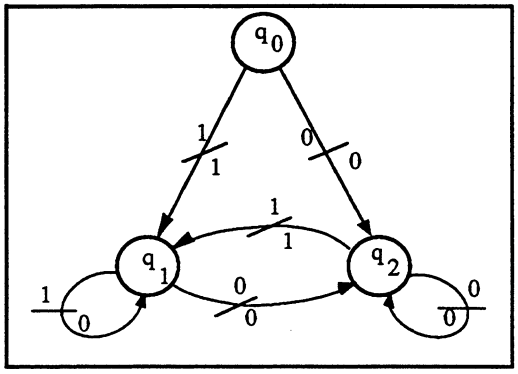


Figure 1.2b. Graphe d'états du circuit détecteur

Notons enfin qu'un système séquentiel peut être spécifié par un diagramme illustrant le comportement des signaux d'entrées et de sorties. Ce diagramme de temps, appelé *chronogramme*, est utilisé pour montrer le séquencement d'un système en termes de séquences d'événements ainsi que des dépendances entre ces événements. Un chronogramme schématise le niveau d'un signal comme une fonction du temps. Un événement peut en déclencher un autre, on parle alors d'une dépendance entre ces deux événements, et on utilise des flèches pour la désigner. On trouvera plus de détails concernant les chronogrammes dans [BT 92, Lau 90], ces rapports développent des méthodes de passage d'un chronogramme vers une propriété temporelle.

1.2 Interconnexion des SMSS

Toute structure complexe de systèmes séquentiels peut être réalisée par l'interconnexion des structures de base : modèles de Moore ou de Mealy (§ 1.1.1, figures 1.1a et 1.1b). Nous donnons, dans cette section, les propriétés d'un système matériel séquentiel issu de l'interconnexion de systèmes évoluant sur les mêmes horloges. Ensuite, nous étudions le cas d'interconnexion de systèmes évoluant sur des horloges synchrones déphasées.

1.2.1 Interconnexion de systèmes évoluant sur la même horloge

Lorsque les éléments de mémorisation des systèmes interconnectés évoluent sur la même horloge, le système composé admet comme modèle une machine d'états finis désignée par machine produit. Soit *n* le nombre des systèmes interconnectés et $M_i = (Q_i, E_i, S_i, \delta_i, \lambda_i, q_{0i})$ $[i = 1, 2, ..., n]$ la machine d'états finis associée au système S_i. La machine produit M_c, associée au système composé, est notée : $M_c = M_1 \times M_2 \times ... \times M_n$. Les caractéristiques de la machine produit M_c dépendent d'une part des propriétés des machines M_i et d'autre part du type d'interconnexion. Trois types d'interconnexion sont possibles : parallèle, série et boucle de retour.

La machine produit issue d'une interconnexion parallèle de deux machines : $M_1 = (Q_1, E_1, S_1, \delta_1, \lambda_1, q_0)$ et $M_2 = (Q_2, E_2, S_2, \delta_2, \lambda_2, q_0)$ (figure1.4a), est définie par le 6-uplet $(Q_p, E_p, S_p, \delta_p, \lambda_p, q_{0p})$ où :

- $Q_p = Q_1 \times Q_2$;
- $E_p = E_1 \times E_2$;
- $S_p = \mathcal{C}\ (S_1 \times S_2)$;
- $\delta_p\ (E_p \times Q_p) = (\delta_1\ (E_1 \times Q_1), \delta_2\ (E_2 \times Q_2))$;
- $\lambda_p\ (E_p \times Q_p) = \mathcal{C}\ (\lambda_1\ (E_1 \times Q_1), \lambda_2\ (E_2 \times Q_2))$;
- $q_{0p} = (q_0, q_0')$.

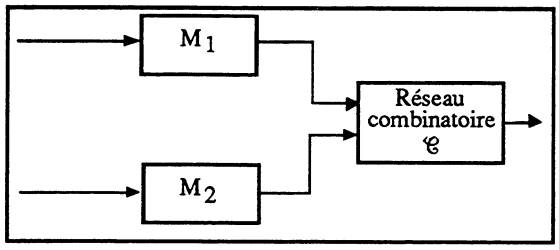


Figure 1.4a. Interconnexion parallèle

La figure 1.4b schématise une interconnexion série des deux machines M_1 et M_2. La machine produit M_S est caractérisée par le 6-uplet $(Q_S, E_S, S_S, \delta_S, \lambda_S, q_{0S})$ où :

- $Q_S = Q_1 \times Q_2$;
- $E_S = E_1$;
- $S_S = S_1$;
- $\delta_S (E_S \times Q_S) =$
  $(\delta_1 (E \times Q_1), \delta_2 (\lambda_1 (E_1 \times Q_1) \times Q_2))$;
- $\lambda_p (E_S \times Q_S) = \lambda_2 (\lambda_1 (E \times Q_1 \times Q_2))$;
- $q_{0S} = (q_0, q_0')$.

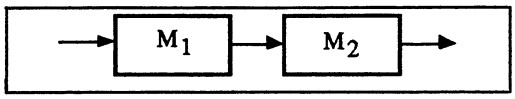


Figure 1.4b. Interconnexion serie

Dans le cas d'une interconnexion en boucle de retour (figure 1.4c), la machine produit est définie par le 6-uplet $(Q_b, E_b, S_1, \delta_b, \lambda_b, q_{0b})$ où :

- $Q_b = Q_1 \times Q_2$;
- $\delta_b (E_b \times Q_b) =$
  $\delta_1 (\mathcal{C} (E \times S_2 \times Q_1), \delta_2 (S \times Q_2))$;
- $\lambda_b (E_S \times Q_S) = \lambda_1 (\mathcal{C} (E_b \times S \times Q_1))$;
- $q_{0S} = (q_0, q_0')$.

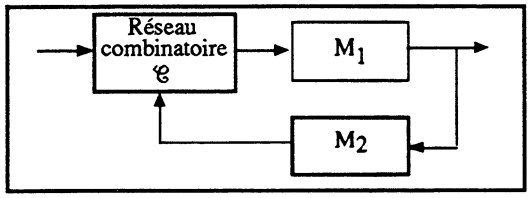


Figure 1.4c. Interconnexion en boucle de retour

Dans le cas général, le produit synchrone d'un ensemble fini de machines $M_i = (Q_i, E_i, S_i, \delta_i, \lambda_i, q_{0i})_{[i = 1, ..., n]}$ est caractérisé par le 6-uplet $(Q_c, E_c, S_c, \delta_c, \lambda_c, q_{0c})$ où :

- Q_c est l'ensemble fini $Q_1 \times Q_2 \times ... \times Q_n$;
- q_0 est l'état initial de la machine produit tel que :

$$q_{0c} = (q_0 (M_1), q_0 (M_2), ..., q_0 (M_n)) ;$$

- δ_c (respectivement λ_c) est la fonction de transition (resp. fonction de sortie), définie de $E_c \times Q_1 \times Q_2 \times ... \times Q_n$ dans Q_c (resp. dans S_c).

- E (resp. S) est l'ensemble fini des symboles d'entrée (resp. de sortie) de la machine produit dépendant, d'une part de l'ensemble des symboles d'entrées E_i (resp. de sorties S_i) des machines M_i, d'autre part du type d'interconnexion : parallèle, série et boucle de retour, des systèmes S_i.

1.2.2 Exemple

Pour illustrer la structure d'une machine produit, considérons le cas d'une interconnexion en cascade (série) de deux systèmes matériels S_1 et S_2. S_1 est un détecteur de fronts montants et S_2 est une cellule d'un compteur synchrone dont le changement d'état s'effectue à chaque front. Les valeurs initiales des éléments de mémorisation des systèmes S_1 et S_2 sont respectivement les valeurs booléennes 0 et 1. Cette interconnexion est schématisée par la figure 1.5.

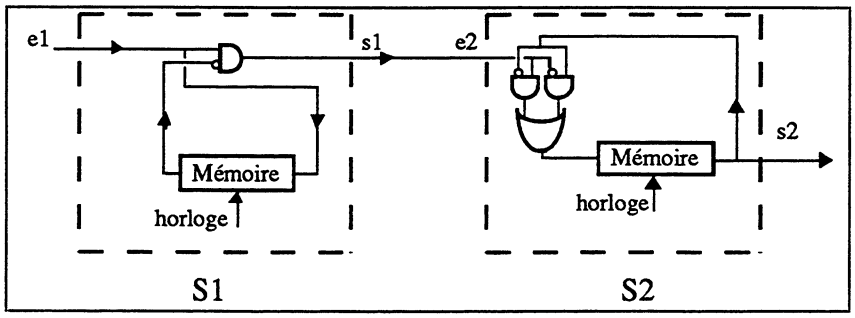


Figure 1.5. Interconnexion en cascade de deux systèmes matériels

Ces deux systèmes évoluent sur la même horloge, leurs FSMs respectives M_1 et M_2 sont définies par les deux 6-uplets suivants :

$M_1 = (Q_1 = \{q_0, q_1, q_2\}, E_1, S_1, \delta_1, \lambda_1, q_0)$ et,
$M_2 = (Q_2 = \{q_0', q_1', q_2'\}, E_2, S_2, \delta_2, \lambda_2, q_0')$.

Les graphes d'états des machines M_1 et M_2 sont donnés dans les figures 1.2b et 1.6.

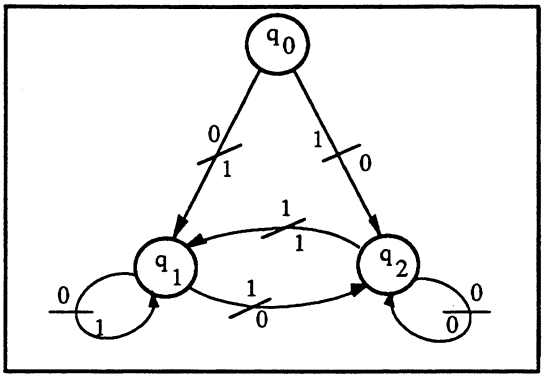


Figure 1.6. Graphe d'états de la cellule compteur synchrone

La machine produit M est définie par le 6-uplet $(Q, E_1, S_2, \delta, \lambda, (q_0, q_0'))$ où :

- Q est le produit $Q_1 \times Q_2$, tel que :

$$\forall\, q \in Q,\ q = (q_i, q_i') \mid q_i \in Q_1 \text{ et } q_i' \in Q_2 ;$$

- δ et λ sont les fonctions de transition et de sortie telles que :

$$\forall\, (e_1, q) \in E_1 \times Q,\ \delta(e_1, q) = (\delta_1(e_1, q_i), \delta_2(q_i', \lambda_1(e_1, q_i))) \text{ et,}$$
$$\lambda(e_1, q) = \lambda_2(\lambda_1(e_1, q_i), q_i').$$

Le graphe d'états de la machine produit est donné en figure 1.7, l'état initial de la machine étant (q_0, q_0').

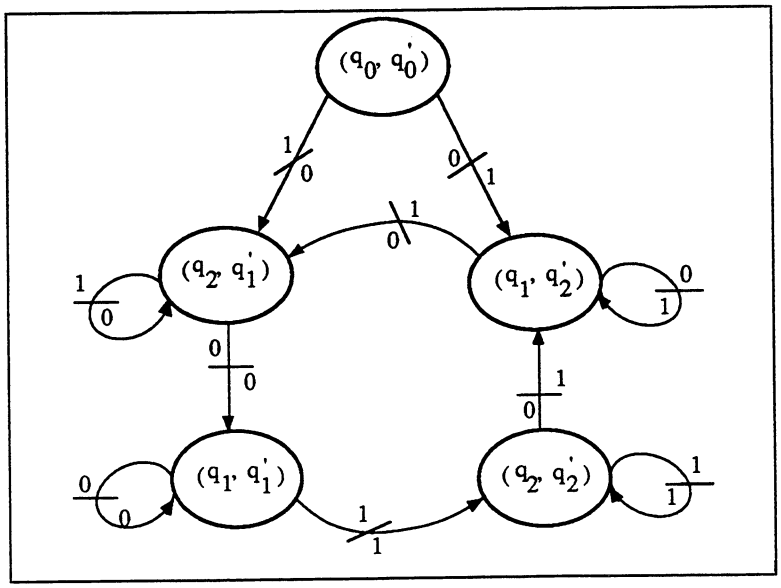


Figure 1.7. Graphe d'états de la machine produit

1.2.3 Cas des systèmes évoluant sur des horloges synchrones déphasées

Lorsqu'on compose des systèmes évoluant sur des horloges différentes, le système produit aura des éléments de mémorisation qui évoluent sur ces horloges différentes. Le modèle de machine associée peut être, a priori, indéterministe. Dans le cas où ces horloges sont synchrones déphasées, en considérant :

- d'une part, que l'horloge h_i de chaque machine $M_i = (Q_i, E_i, S_i, \delta_i, \lambda_i, q_{0i})$ est une entrée et,

- d'autre part, qu'un élément de mémorisation maintient ses données jusqu'au prochain pas d'horloge ;

le modèle associé est alors une machine d'états finis déterministe issue d'un produit synchrone de machines $M_{i\ [i = 1, 2, \ldots, n]}$. Chaque machine M_i est caractérisée par le 6-uplet $(Q_i, E_i \times \{h_i\}, S_i, \delta_{h_i}, \lambda_{h_i}, q_{0i})$ où δ_{h_i} et λ_{h_i} sont les nouvelles fonctions de transition et de sortie telles que :

$$\forall (e_i, 1, q_i) \in \{E_i \times \{h_i\} \times Q_i\},\ \delta_{h_i}(e_i, 1, q_i) = \delta_{h_i}(e_i, q_i) \text{ et, } \lambda_{h_i}(e_i, 1, q_i) = \lambda_{h_i}(e_i, q_i) ;$$

$$\forall (e_i, 0, q_i) \in \{E_i \times \{h_i\} \times Q_i\},\ \delta_{h_i}(e_i, 0, q_i) = q_i \text{ et, } \lambda_{h_i}(e_i, 0, q_i) = \lambda_{h_i}(e_i, q_i).$$

La machine associée au système composé issu de l'interconnexion de *n* systèmes évoluant chacun sur une horloge est alors le produit synchrone des différentes machines associées à chaque système, comme il a été défini au paragraphe 1.2.1. Chaque machine évolue sur la même horloge globale implicite. La machine produit est caractérisée par le 6-uplet $(Q_c, E_c \times \{h_1\} \times \{h_2\} \times \ldots \times \{h_n\}, S_c, \delta_c, \lambda_c, q_{0c})$ où δ_c et λ_c sont les fonctions de transition et de sortie telles que :

$$\forall (e_c, \{1\}^n, q_c) \in \{E_c \times \{1\}^n \times Q_c\},\ \delta_c(e_c, \{1\}^n, q_c) = \delta_c(e_c, q_c) \text{ et, } \lambda_c(e_c, \{1\}^n, q_c) = \lambda_c(e_c, q_c) ;$$

$$\forall (e_c, \{0\}^n, q_c) \in \{E_c \times \{0\}^n \times Q_c\},\ \delta_c(e_c, \{0\}^n, q_c) = q_c \text{ et, } \lambda_c(e_c, \{0\}^n, q_c) = \lambda_c(e_c, q_c).$$

1.3 Conclusion

Nous avons vu dans ce chapitre que les systèmes matériels séquentiels se divisent en deux catégories : les systèmes synchrones et les systèmes asynchrones. Les concepteurs adoptent, en général, l'approche synchrone qui lie le séquencement du système au temps. Celle-ci permet de s'affranchir des problèmes de stabilité et par conséquent, du non déterminisme des systèmes asynchrones. Un système matériel séquentiel synchrone ou SMSS a pour modèle une machine d'états finis déterministe. Un système issu de l'interconnexion d'autres systèmes évoluant sur des horloges synchrones déphasées admet aussi pour modèle une machine d'états finis en considérant : d'une part, que les horloges sont des entrées et d'autre part, que le séquencement est cadencé sur une horloge implicite globale.

Chapitre 2

Validation des systèmes matériels séquentiels

Comme nous l'avons déjà mentionné dans l'introduction générale, deux problèmes de vérification se posent aux différentes étapes de la conception d'un système matériel : la vérification de conformité et la vérification de propriétés. Chacun de ces problèmes a reçu une attention particulière pendant ces dernières années. La vérification de conformité a été étudiée en particulier dans [CCBP 91, CBM 89, GDN 90, Gor 83, TP 90, VAD 91, VC 89] et la vérification de propriétés a été étudiée dans [BC 86, Boc 82, CM 90a, FF 89]. Certains se sont intéressés aux deux problèmes de vérification [CM 90b]. Les auteurs de ces travaux ont concentré leurs études aux techniques de vérification. Par contre, ils ne se sont pas préoccupés des problèmes pratiques qu'un concepteur aura à affronter pour exprimer et résoudre ces deux problèmes de vérification.

Dans ce chapitre nous donnons les différentes façons utilisées pour la description des systèmes matériels. Les différentes approches pour la vérification formelle de ces systèmes sont ensuite discutées. Enfin, l'approche de vérification étudiée dans la suite de cette thèse est présentée.

2.1 Description des systèmes matériels

D'une manière générale, dans le domaine de la description des systèmes matériels, on distingue deux approches [TP 90] :

- l'utilisation de langages spécifiques à la description du matériel et,
- l'utilisation de formalismes mathématiques.

Dans la première approche, d'une part des langages de programmation prenant en compte la notion du temps sont utilisés pour décrire les systèmes matériels. Parmi ces langages, on peut citer le langage ADA [Bar et al 85]. D'autre part, des langages ont été conçus spécialement pour la description du matériel, on peut citer particulièrement les langages : Circal [Mil 83], muFP [She 84], CADOC [Cra 85], IRENE [Mar 86] et le langage VHDL qui est actuellement la norme industrielle [VHDL 87].

Dans la seconde approche, des formalismes mathématiques ont été proposés pour la description formelle des systèmes matériels séquentiels. On trouvera dans [CP 88] un exposé de ces différents formalismes. Nous citons, en particulier, les formalismes suivants : la logique du premier ordre [Dar 79], les logiques temporelles [BC 86, Boc 82, FF 89], la logique d'ordre supérieur ou HOL [Gor 86] et la logique LSM (de l'anglais, Logic of Sequential Machines) [Gor 83].

Discussion. Les langages de descriptions du matériel ou HDL (de l'anglais, Hardware Description Language) ont été conçus, en général, dans des buts particuliers, comme la simulation, la synthèse, ... De ce fait la manipulation de ces langages pour la description et la simulation par exemple, est facile. Cependant, les manipulations formelles des langages de description du matériel sont difficiles. Par exemple, une description d'un système matériel avec une spécification des temps de traversée de ses composants est difficile à lier à un modèle qu'on peut formellement manipuler .

Par contre, les formalismes mathématiques, adaptés pour les manipulations formelles, sont difficiles à utiliser pour décrire du matériel. En effet, les concepteurs de ces formalismes se sont préoccupés du problème de la vérification formelle et ils ont délaissé le problème de la manipulation de ces formalismes par un utilisateur non initié à ce type de formalisme. De plus, il est difficile de lier ces modèles formels à des modèles utilisables en simulation. Notons aussi que des formalismes tels que les logiques temporelles pour la description des propriétés temporelles sont mal adaptés à la description structurelle des systèmes matériels.

2.2 Les différentes approches pour la vérification formelle des systèmes matériels séquentiels

Dans le domaine de la vérification formelle des systèmes matériels séquentiels, nous distingons les deux approches suivantes :

(1) la preuve déductive et,

(2) l'évaluation d'une formule sur un modèle de machine d'états finis associée au système matériel (model checking).

Nous analysons, dans ce qui suit, ces deux approches.

2.2.1 La preuve déductive

Cette approche est fondée sur les formalismes mathématiques décrits précédemment (§ 2.2). La vérification consiste à prouver par raisonnement déductif la correction de la description de la réalisation d'un système matériel par rapport à celle de sa spécification. Cette approche a conduit à des réalisations pratiques de systèmes de vérification. Nous citons les trois systèmes suivants :

- Le système LCF-LSM [Gor 83]. Ce système a été développé par M. Gordon à l'université de Cambridge. Le système permet de décrire la spécification et la réalisation d'un système matériel séquentiel à l'aide du formalisme LSM. La vérification formelle se fait en démontrant l'équivalence entre les deux descriptions par un raisonnement déductif et d'une manière interactive ; l'utilisateur guidera le système dans l'utilisation de règles de déduction.

- Le démonstrateur de théorèmes de Boyer-Moore [BM 79]. Ce démonstrateur travaille sur un sous-ensemble de la logique du premier ordre et permet d'effectuer des preuves par induction (extension de la preuve par récurrence sur les entiers). Des utilisations pratiques de ce démonstrateur ont été faites. En particulier, nous citons le système de W. Hunt [Hun 86] et celui de D. Borrione et al [BPPC 89, Pie 90].

- Le système HOL de Cambridge [Gor 87]. Ce système utilise la logique d'ordre supérieur (une extension de la logique du premier ordre) pour décrire le comportement et la structure des systèmes matériels. La vérification consiste à montrer que la description de la structure du système matériel correspond à son comportement en utilisant des règles d'inférence de la logique. A. Cohn a vérifié avec ce système un microprocesseur 32-bits [Coh 87].

Discussion. De façon générale, les preuves avec ces systèmes sont longues et demandent une certaine "technique logique". Ces systèmes fonctionnent sur un mode de raisonnement déductif et se heurtent ainsi aux problèmes ardus de la démonstration automatique. Par exemple, dans le démonstrateur de théorèmes de Boyer-Moore, l'utilisateur aura à interrompre le démonstrateur à chaque fois que celui-ci choisit une mauvaise direction (choix d'une mauvaise règle de déduction). L'utilisateur corrige la direction de la preuve en donnant la règle

adéquate et il relance le démonstrateur une nouvelle fois. L'obstacle majeur de ces systèmes n'est pas le temps de calcul du système de preuve mais c'est la construction de la preuve qui nécessite un temps exorbitant. De plus, cette construction de preuve est à refaire à chaque modification effectuée sur la description du système à vérifier.

2.2.2 L'évaluation d'une formule sur un modèle de machine d'états finis associé au système matériel : "model checking"

Deux problèmes de vérification se posent aux différentes étapes de la conception d'un système matériel : (1) la vérification de conformité, c'est-à-dire la comparaison entre deux descriptions à des niveaux d'abstraction donnés (de la spécification à la réalisation) et (2) la vérification de propriétés cruciales sur une description d'un système à un certain niveau d'abstraction.

Pour résoudre le premier problème de vérification (1) on compare les deux machines associées aux deux descriptions du système matériel. Cette comparaison peut être définie au sens d'équivalence directe entre les deux machines [CBM 89, GDN 90, TP 90, CGPS 91] ; c'est-à-dire, à partir des états initiaux, elles produisent les mêmes sorties pour les mêmes entrées. Ce cas s'applique lorsque les deux descriptions sont d'une part, au même niveau d'abstraction et d'autre part, lorsqu'elles ont la même architecture. Lorsque les niveaux d'abstraction et / ou les architectures sont différents, la comparaison entre les deux machines est réalisée par rapport à un critère d'observation. V. Aelten et al ont étudié dans [VAD 91] le cas de la comparaison de deux machines à architectures différentes. Ils définissent une "machine de vérification" avec une seule sortie sur laquelle une exploration en profondeur d'abord, avec une technique d'énumération implicite, a été utilisée pour vérifier que la machine retourne toujours la constante booléenne 1. Cette approche a été intégrée dans le système de synthèse logique MIS [BRSW 87].

Pour étudier le comportement des systèmes séquentiels en général, la logique temporelle arborescence ou CTL (de l'anglais, Computational Tree Logic) [BMP 83] a été introduite. Cette logique permet la formulation simple de certaines classes de propriétés. Comme il a été souligné dans [AEF 90] : Queille et Sifakis [QS 81] ainsi que Clarke et al [CES 83] ont montré que pour prouver qu'une formule de logique temporelle arborescente est satisfaite dans un système de transition à états finis, il suffit de montrer que la formule est satisfaite dans le modèle. Cette tâche est algorithmiquement moins complexe que de chercher à prouver la formule par des raisonnements déductifs. De nombreuses recherches ont été menées dans ce sens pour résoudre le deuxième problème de vérification ((2) vérification des propriétés temporelles) sur les systèmes matériels séquentiels. Tout d'abord, il a été proposé dans [CES 83] de construire le graphe d'états de la machine M et de vérifier la propriété pour chaque transition (si M est une

machine de Mealy) ou pour chaque état (si M est une machine de Moore). Même si un parcours du graphe d'états en profondeur d'abord permet d'obtenir un algorithme linéaire, cette méthode est limitée [SF 86] (coûteuse en temps et en mémoire) par la taille du graphe d'états de la machine. Pour s'affranchir de cette limite, des méthodes symboliques ont été proposées [CM 90a, BCM 90]. Ces méthodes sont fondées sur le calcul de la fonction caractéristique d'un ensemble d'états satisfaisant la propriété CTL. Ainsi, au lieu de vérifier que chaque état satisfait la propriété, on vérifie que la fonction caractéristique est valide. Pour le codage des fonctions caractéristiques, la technique des Graphes de Décision Binaire ou BDDs (de l'anglais, Binary Decision Diagrams) [Bry 86, CBM 89] a été utilisée. Cette approche a conduit à des réalisations pratiques de systèmes de vérification automatique. Nous citons, en particulier, le système SIAM du centre de recherche BULL [Cou 91], qui permet aussi de traiter la vérification de conformité dans le cas particulier où les deux descriptions sont de même niveau.

Discussion. Cette approche basée sur le modèle de machine d'états finis a permis la réalisation d'outils de vérification automatiques. Quel que soit le type de vérification réalisé, ces outils réalisent l'exploration d'un graphe d'états. Le parcours du graphe d'états par une méthode énumérative est limité par le nombre d'états à explorer. Des méthodes symboliques avec la technique de représentation des BDDs ont permis de s'affranchir dans de nombreux cas de cette limite. Par exemple, Le système de Clarke et al [BCM 90] a réussi à explorer un graphe d'états à 5×10^{20} états, associé a une unité arithmétique et logique à architecture pipeline, en 22 minutes sur une station SUN 3. Néanmoins, la méthode symbolique est coûteuse en mémoire [Cou 91]. Notons aussi que ces techniques sont fondées sur des algorithmes de parcours d'un graphe en avant. Le problème majeur de cette méthode concerne la taille du graphe parcouru, qui n'est en général pas minimal. Des recherches s'orientent actuellement vers la minimisation du graphe d'états à parcourir soit en décomposant la vérification en plusieurs vérifications sur des sous-machines [VAD 91, Kur 90], soit en minimisant les différentes sous-machines vis-à-vis de la propriété à vérifier telles que leur produit préserve uniquement le comportement nécessaire pour la vérification de la propriété [CSSB 92].

Pour spécifier les systèmes matériels, les divers systèmes de vérification fondés sur cette approche utilisent des langages spécifiques pour la description des systèmes matériels et une logique temporelle pour spécifier les propriétés temporelles relatives aux comportements des systèmes matériels. Par exemple, BULL utilise le langage LDS ou un sous-ensemble du langage VHDL pour décrire les systèmes matériels aux niveaux comportemental et structurel [DM 92] et la logique CTL pour spécifier les propriétés temporelles relatives aux comportements des systèmes matériels. La logique CTL n'est pas facile à utiliser par un utilisateur non initié à ce type de formalisme. Pour un concepteur, il serait intéressant d'utiliser le même langage pour la description des systèmes matériels aux différents niveaux de

conception et pour la spécification des propriétés temporelles. Dans la section suivante, nous présentons une approche de vérification définissant un cadre unifié pour résoudre les problèmes de vérification des SMSS.

2.3 Un cadre unifié pour la description et la vérification des SMSS

L'approche de vérification développée dans ce document est fondée sur le modèle de machine d'états finis. Elle consiste à ramener tout problème de vérification à une vérification d'un invariant sur une machine d'états finis (model checking). Dans cette approche, présentée dans la deuxième partie de ce document, nous considérons que les descriptions d'un SMSS aux différents niveaux d'abstraction dénotent des machines d'états finis (chapitre 1).

Seule la classe de propriétés temporelles définissant un langage qui peut être décrit par une machine d'états finis est considérée. En effet, l'expérience nous a montré qu'un très grand nombre de propriétés temporelles, relatives aux comportements des systèmes matériels, font partie de cette classe de propriétés. La vérification de ces propriétés consiste alors à définir la machine produit issue de l'interconnexion de la machine associée à la propriété temporelle et de la machine associée à la description du SMSS. Sur cette machine produit la vérification peut être réalisée par simple parcours du graphe d'états.

La vérification de conformité est une comparaison entre deux descriptions d'un SMSS de même niveau d'abstraction ou de niveaux différents (de la spécification à la réalisation). Ces descriptions peuvent avoir des architectures différentes. De ce fait, la vérification ne peut être réalisée par une preuve d'équivalence directe entre les deux machines d'états finis associées aux deux descriptions. Pour cela, nous introduisons les notions de machines comparables et de critères d'observation. Par la suite, nous montrons que la vérification de conformité, comme dans le cas de la vérification de propriétés, consiste aussi à définir une machine de vérification sur laquelle la vérification peut être réalisée par simple parcours du graphe d'états.

Finalement, nous abordons dans cette approche, le problème général de la validation d'un SMSS sous un environnement de fonctionnement. Cet environnement est pris en considération durant le processus de vérification en lui associant aussi un modèle de machine d'états finis. Le problème de vérification se ramène ainsi à une définition d'une machine de vérification sur laquelle la vérification peut être aussi réalisée par simple parcours du graphe d'états.

Pour valider cette approche, nous étudions, dans la troisième partie de ce document, l'application du langage Lustre [HCRP 91] et de l'outil de vérification Lesar [Glo 89, Rat 92].

Le langage Lustre et l'outil de vérification associé permettent de définir un cadre unifié pour la description et la vérification des SMSS. En effet, les deux problèmes de vérification de conformité et de propriétés, sont abordées de la même manière [TB 92]. Dans les deux cas, nous définissons à partir des différents programmes Lustre décrivant la description du SMSS, sa spécification et son environnement de fonctionnement, un programme de vérification avec une seule sortie notée Flag. La vérification se ramène alors à vérifier que l'unique sortie "Flag" de ce programme de vérification est toujours vraie. Un tel programme opérant sur des flots de booléens dénote une machine d'états finis sur laquelle la vérification est réalisée automatiquement par l'outil de preuve Lesar associé au langage Lustre. Lesar vérifie que la sortie Flag de "la machine de vérification" vaut toujours la valeur booléenne 1. Deux stratégies d'exploration du graphe d'états ont été implémentées :

(1) la méthode énumérative [Glo 89] (état par état et transition par transition) et,

(2) une technique symbolique manipulant des ensembles d'états et de transitions [Rat 92].

Partie II

Une approche de vérification unifiée

Chapitre 3

Spécification des propriétés temporelles

Dans ce chapitre, nous définissons tout d'abord les termes que nous utiliserons par la suite. Puis, les principales classes de propriétés relatives au comportement des systèmes matériels sont présentées. Enfin, nous abordons le problème de la spécification des propriétés temporelles.

3.1 Terminologie et notations

Cette section donne la terminologie utilisée par la suite. Nous avons vu précédemment qu'une séquence est une suite de n-uplets de booléens ou symboles et qu'un ensemble de séquences d'un vocabulaire V est appelé langage noté L(V).

- Nous désignerons par langage universel, noté V^*, le langage contenant l'ensemble de toutes les séquences possibles d'un vocabulaire V.

- La concaténation de deux séquences α et β de symboles a_i et b_j est définie comme suit :

$$\alpha\beta = a_0\, a_1 \cdots a_n\, b_0\, b_2 \cdots b_m$$

- Etant donnée une séquence α, la séquence α^* est obtenue par concaténation de la séquence α a elle même un nombre quelconque de fois :

$$\alpha = \alpha\ \alpha \cdots$$

- La longueur d'une séquence α, notée $|\alpha|$, désigne le nombre d'éléments de la séquence. Une séquence d'une longueur finie est notée : $|\alpha| < \infty$

- On appelle langage régulier un langage défini par les clauses suivantes :

  1. les séquences ayant une longueur 1 et 0 définissent un langage régulier ;

  2. si α et β sont des séquences d'un langage régulier, alors les séquences $\alpha\beta$ et α^* sont aussi des séquences d'un langage régulier;

  3. toute séquence obtenue en appliquant les clauses 1 et 2 un nombre fini de fois est une séquence d'un langage régulier.

- On appelle préfixe d'une séquence α toute sous-séquence de α. Un préfixe β de α, noté $\beta \leq \alpha$ ou $\beta < \alpha$, est défini comme suit :

$$\beta \leq \alpha = (a_0\, a_1 \cdots a_n) \Leftrightarrow \beta = \alpha \text{ ou } \beta \leq (a_0\, a_1 \cdots a_{n-1}) \text{ et,}$$

$$\beta < \alpha \Leftrightarrow \beta \leq (a_0\, a_1 \cdots a_{n-1})$$

- Un langage L est dit clos par préfixe (en anglais, prefix-closed), si et seulement si, tout préfixe fini d'une séquence du langage est aussi une séquence du langage :

$$(\sigma \in L) \quad \text{ssi} \quad (\forall\, \beta \mid \beta \leq \sigma \text{ et } |\beta| < \infty, \beta \in L\})$$

3.2 Les différentes classes de propriétés d'un système matériel

Les propriétés temporelles relatives aux comportements des systèmes matériels séquentiels peuvent être réparties, principalement, en deux classes [AEF 90, Boc 82, CP 88, LCGP 89] :

(1) les propriétés de sûreté (en anglais, safety properties) et,

(2) les propriétés de vivacité (en anglais, liveness properties).

Formellement, une propriété temporelle est un sous-langage L' du langage universel des entrées E et des sorties S du système matériel :

$$L' \subseteq (E \times S)^*$$

Une propriété de sûreté (1) définit un langage clos par préfixe [BFH 90]. Tout préfixe fini d'une séquence satisfaisant la propriété de sûreté P satisfait aussi la propriété :

$$\sigma \models P \Leftrightarrow (\forall \beta \mid \beta \leq \sigma \text{ et } |\beta| < \infty, \beta \models P)$$

Les propriétés de vivacité (2) sont des propriétés faisant référence à un intervalle temporel non borné. De telles propriétés sont caractérisées par l'affirmation qu'un tel événement aura lieu. En terme de langage, une propriété de vivacité définit un langage L' pour lequel quelle que soit la séquence β appartenant au langage universel $(E \times S)^*$, il existe une séquence σ telle que la séquence $(\beta\sigma)$ est un élément du langage L' :

$$\forall \beta \in (E \times S)^*, \exists \sigma \mid (\beta\sigma) \in L'$$

Un concepteur vérifie des propriétés de vivacité, lorsqu'il s'intéresse uniquement à la réaction du système matériel en présence d'un événement particulier, sans aucune référence au temps de réponse du système. Par exemple, en présence d'une demande d'interruption d'une priorité donnée, la réaction d'un microprocesseur 68000 [Nac 86] consiste à placer le code de priorité d'interruption sur les trois lignes d'adresses A_3, A_2 et A_1.

Dans la plupart des cas, les propriétés de sûreté suffisent à spécifier le comportement des systèmes matériels. En effet, le temps nécessaire à la réaction d'un système, en présence d'un événement, est généralement fini et connu. Par exemple, la réaction du microprocesseur 68000 suite à une demande d'interruption se fait en un temps fini. De plus, ce temps de réponse fait partie de la spécification du microprocesseur 68000.

Généralement, les propriétés de sûreté qu'un concepteur aura à vérifier sur un SMSS sont du type : une fois un événement pendant un intervalle temporel, toujours un événement pendant un intervalle, ... En d'autres termes, il s'agit d'une classe de propriétés de sûreté dont le langage associé est régulier (§ 3.1). Nous nous limiterons par la suite à ce type de propriétés de sûreté dont le langage associé est régulier.

3.3 Spécification des propriétés de sûreté

Souvent, les propriétés temporelles de sûreté relatives aux comportements des systèmes matériels expriment des relations entre événements pendant un intervalle d'observation. D'une manière générale, une propriété de sûreté P_S s'exprime de la façon suivante : pendant un *intervalle d'observation*, chaque fois qu'un événement spécifique que nous désignons par le *contexte* se produit, une relation relative à un ensemble d'événements, que nous désignons par

la *réaction*, est vérifiée. Par exemple, pour la propriété : "Après le signal reset, à chaque fois qu'un événement P a lieu, un événement Q a eu lieu au moins une fois depuis la dernière occurrence de l'événement P", l'intervalle d'observation est l'intervalle temporel [reset $+\infty$]. Le contexte de cette propriété est l'occurrence de l'événement P et sa réaction est l'occurrence au moins une fois de l'événement Q depuis la dernière occurrence de l'événement P.

Ces événements, cet intervalle d'observation et cette réaction sont des *propriétés de séquences finies*. Toute fonction booléenne exprime une propriété de séquences finies telle qu'une séquence finie σ satisfait la propriété P_{rop}, si et seulement si, la fonction booléenne correspondante vaut la valeur booléenne 1 sur la séquence σ [HLR 92].

Les différents paramètres temporels d'une propriété de sûreté sont étudiés dans cette section. Ensuite, nous abordons le problème de la spécification des propriétés de sûreté à l'aide de ces paramètres temporels. Enfin, nous donnons un ensemble d'opérateurs de base aidant à exprimer des propriétés de sûreté complexes.

3.3.1 Evénements définis

La notion d'événement pour un système matériel est liée le plus souvent à des transitions sur ses lignes d'entrées ou de sorties, autrement dit à des fronts montants ou descendants de celles-ci. Ces événements relatifs au comportement des systèmes matériels définissent un langage régulier fini caractérisé par un ensemble fini de séquences finies B d'un vocabulaire $E \times S$:

$$L = \{ (E \times S)^* B \}$$

Ce langage est appelé le langage des événements définis [Boo 67]. Par exemple, l'ensemble des séquences se terminant par (01) est le langage des événements fronts montants. Un événement défini peut être caractérisé par une expression booléenne *F* sur la séquence définissant l'événement. Un événement est dit présent sur les entrées et sorties d'un système matériel, si et seulement si, l'expression F vaut la valeur booléenne 1.

Un événement peut être vu comme une propriété de séquences finies définie de l'ensemble des séquences du langage universel des entrées et des sorties du système matériel $(E \times S)^*$ dans $\{0, 1\}$. Une séquence $\sigma = (a_0\, a_1 \cdots a_n)$ satisfait un événement P, noté $\sigma \models P$, si et seulement si, la fonction booléenne associée vaut la valeur 1. Par exemple, l'événement front montant (r_edge) sur la séquence $\sigma = (a_0\, a_1 \cdots a_n)$ où les a_i sont des booléens, est exprimé à l'aide de la

formule booléenne B_{edge} telle que :

$$B_{edge} = \overline{a}_{n-1} \cdot a_n$$

Notons enfin, que le contexte d'une propriété de sûreté est en général l'occurrence d'un événement défini.

3.3.2 Intervalle d'observation

En général, une propriété temporelle spécifie un comportement donné d'un système matériel dans un intervalle d'observation borné par des événements définis. En dehors de cet intervalle, la propriété est considérée comme vérifiée. On distingue quatre types d'intervalles d'observation :

- un intervalle non borné $[t_0 \ +\infty]$, t_0 est l'instant initial correspondant à l'état initial du système ;
- un intervalle borné *inférieurement* par un événement P : $[P \ +\infty]$;
- un intervalle borné *supérieurement* par un événement P : $[t_0 \ P[$;
- un intervalle borné par deux événements P et Q : $[P \ Q[$.

Un intervalle d'observation est une propriété de séquences finies notée I_{obs}. Cette propriété est définie de l'ensemble des séquences du langage *L* de la propriété dans {0,1}. Les propriétés relatives aux intervalles temporels $[P \ +\infty]$ et $[P \ Q[$ notées respectivement **After (P)** et **From _to_ (P,Q)** sont des propriétés primitives définies comme suit :

- $[P \ +\infty]$: $\forall \sigma \in L, \sigma \models$ After (P) ssi $\exists \beta \leq \sigma, \beta \models P$
- $[P \ Q[$: $\forall \sigma \in L, \sigma \models$ From_to_ (P, Q) ssi
  $(\exists \beta \leq \sigma, \beta \models P)$ et $(\forall \alpha \mid \beta \leq \alpha \leq \sigma, \alpha \not\models Q)$

La propriété **Before (P)** relative à l'intervalle temporel $[t_0 \ P[$ est formellement définie à partir de la propriété After (P) comme suit :

- $[t_0 \ P[$: $\forall \sigma \in L, \sigma \models$ Before (P) ssi $\sigma \not\models$ After (P)

Notons que la propriété $I_{[t0 \ +\infty]}$ correspondant à l'intervalle $[t_0 \ +\infty]$ est constamment vraie. Soit formellement :

$$\forall \sigma \in L, \sigma \models I_{[t0\ +\infty]}$$

3.3.3 Réaction d'une propriété

La réaction d'une propriété est un comportement particulier relatif à un événement, ou à un ensemble d'événements. Une réaction relative aux événements E_1, E_2, ..., E_n, notée R_{eac} (E_1, E_2, ..., E_n), est une propriété de séquences finies. L'expression des propriétés de sûreté est plus aisée lorsque des réactions de base sont définies. Nous distinguons les deux réactions de base suivantes :

- **Once_since_ (P, Q)** est satisfaite par une séquence σ, si et seulement si, l'événement défini P s'est produit au moins une fois depuis l'occurrence de l'événement Q :

  $\sigma \models$ Once_since_ (P, Q) ssi

  soit $\forall \beta \mid \beta \leq \sigma, \beta \not\models Q$

  ou $(\exists \beta \mid \beta \leq \sigma, \beta \models Q)$ et $(\forall \alpha \mid \beta < \alpha \leq \sigma, \alpha \not\models Q)$ et $(\exists \delta \mid \beta \leq \delta \leq \sigma, \delta \models P)$

- **Onlyonce_since_ (P, Q)** est satisfaite par une séquence σ, si et seulement si, l'événement défini P s'est produit une et une seule fois depuis l'occurrence de l'événement Q :

  $\sigma \models$ Only once_since_ (P, Q) ssi

  soit $\forall \beta \mid \beta \leq \sigma, \beta \not\models Q$

  ou $(\exists \beta \mid \beta \leq \sigma, \beta \models Q)$ et $(\forall \alpha \mid \beta < \alpha \leq \sigma, \alpha \not\models Q)$ et $(\exists! \delta \mid \beta \leq \delta \leq \sigma, \delta \models P)$

Le symbole $\exists!$ désigne le quantificateur : *il existe une et une seule fois.*

Nous déduisons de la réaction Once_since, d'autres réactions. En particulier, nous définissons les deux réactions suivantes :

- **Always_since_ (P, Q)** est satisfaite par une séquence σ, si et seulement si, l'événement défini P est constamment présent depuis l'occurrence de l'événement Q. Cette réaction est définie à l'aide de la réaction primitive Once_since (P, Q) comme suit :

$$\sigma \models \text{Always\_since\_ (P, Q)} \quad \text{ssi} \quad \sigma \not\models \text{Once\_since\_ } (\overline{P}, Q)$$

• **Never_since_ (P, Q)** est satisfaite par une séquence σ, si et seulement si, l'événement défini P ne s'est jamais produit depuis l'occurrence de l'événement Q. Cette réaction est définie à l'aide de la réaction Always_since (P, Q) comme suit :

$$\sigma \models \text{Never\_since\_}(P, Q) \quad \text{ssi} \quad \sigma \models \text{Always\_since\_}(\overline{P}, Q)$$

3.3.4 Propriétés de sûreté

Comme nous l'avons déjà mentionné en section 3.2, une propriété de sûreté est un langage clos par préfixe. En d'autres termes, une propriété de sûreté est une application $\mathbb{P}$ définie de l'ensemble des séquences du langage $(E \times S)^*$ dans $\{0, 1\}$ telle que :

$$\forall \sigma, \sigma \in (E \times S)^*, \quad (\mathbb{P}(\sigma) = 1) \quad \Leftrightarrow \quad (\forall \beta \mid \beta \leq \sigma \text{ et } |\beta| < \infty, \mathbb{P}(\beta) = 1)$$

Formellement, une séquence σ satisfait la propriété de sûreté P_s, si et seulement si, pour tout préfixe fini β de σ satisfaisant la propriété d'intervalle I_{obs}, s'il existe un préfixe α de β satisfaisant le contexte C_{ont} alors le préfixe α satisfait aussi la réaction R_{eac}. Ceci s'exprime formellement de la façon suivante :

$$\sigma \models P_{rop} \text{ ssi } \forall \beta \mid \beta \leq \sigma \text{ et } |\beta| < \infty, (\beta \models I_{obs}) \Rightarrow [(\forall \alpha \mid \alpha \leq \beta, \alpha \models C_{ont}) \Rightarrow (\alpha \models R_{eac})]$$

On distingue deux cas particuliers :

• L'intervalle d'observation de la propriété est $[t_0 \quad +\infty]$, alors la propriété d'intervalle $I_{[t_0 \ +\infty]}$ est constamment vraie. Dans ce cas particulier, une séquence σ satisfait la propriété de sûreté P_{rop}, si et seulement si, pour tout préfixe fini β de σ satisfaisant le contexte C_{ont}, β satisfait aussi la réaction R_{eac} :

$$\sigma \models P_{rop} \quad \text{ssi} \quad \forall \beta \mid \beta \leq \sigma \text{ et } |\beta| < \infty, (\beta \models C_{ont}) \Rightarrow (\beta \models R_{eac})$$

• La propriété de sûreté n'a pas de contexte. Dans ce cas particulier, une séquence σ satisfait la propriété de sûreté P_s, si et seulement si, pour tout préfixe β de σ satisfaisant la propriété d'intervalle I_{obs}, β satisfait aussi la réaction R_{eac} :

$$\sigma \models P_{rop} \quad \text{ssi} \quad \forall \beta \mid \beta \leq \sigma \text{ et } |\beta| < \infty, (\beta \models I_{obs}) \Rightarrow (\beta \models R_{eac})$$

Par exemple, les paramètres temporels de la propriété : "jamais l'événement P avant l'occurrence de l'événement Q" sont : l'intervalle d'observation $[t_0 \quad Q[$ et la réaction "jamais P" pendant cet intervalle. Cette propriété n'a donc pas de contexte.

3.3.5 Opérateurs de base

Dans ce paragraphe, nous définissons un ensemble d'opérateurs de base. Ces opérateurs sont les éléments de base d'un *langage de spécification de propriétés de sûreté* dont le sens intuitif est très simple et ne diffèrent pas de leurs homonymes du langage usuel. L'ensemble de ces opérateurs est défini par rapport à *l'intervalle d'observation* $[t_0 \quad +\infty]$, à l'exception de la propriété Never_before_ (P, Q).

- **Always (P)** : une séquence σ satisfait cette propriété, si et seulement si, l'événement P est présent pendant tout intervalle temporel :

$$\sigma \models \text{Always (P)} \text{ ssi } (\forall \beta \mid \beta \leq \sigma \text{ et } |\beta| < \infty, \beta \models P)$$

- **Never (P)** : une séquence σ satisfait cette propriété, si et seulement si, l'événement P n'est jamais présent pendant tout intervalle temporel :

$$\sigma \models \text{Never (P)} \text{ ssi } (\forall \beta \mid \beta \leq \sigma \text{ et } |\beta| < \infty, \beta \not\models P)$$

- **Once_from_to_ (P, Q, R)** : une séquence σ satisfait cette propriété, si et seulement si, P se produit au moins une fois dans tout intervalle minimal séparant une occurrence de Q et une occurrence de R. Formellement une séquence σ satisfait la propriété de sûreté Once_from_to (P, Q, R), si et seulement si, pour tout préfixe β de σ satisfaisant l'événement R, β satisfait aussi la réaction Once_since (P, Q) :

$$\sigma \models \text{Once\_from\_to\_ (P, Q, R) ssi}$$

$$\forall \beta \mid \beta \leq \sigma \text{ et } |\beta| < \infty \text{ et } |\beta| < \infty, (\beta \models R) \Rightarrow (\beta \models \text{Once\_since\_ (P, Q)})$$

- **Always_from_to_ (P, Q, R)** : une séquence σ satisfait cette propriété, si et seulement si, P est constamment présent entre Q et R. Formellement, une séquence σ satisfait la propriété de sûreté Always_from_to_ (P, Q, R), si et seulement si, pour tout préfixe β de σ satisfaisant le contexte R, β satisfait aussi la réaction Always_since_ (P, Q) :

$$\sigma \models \text{Always\_from\_to\_ (P, Q, R) ssi}$$

$$\forall \beta \mid \beta \leq \sigma \text{ et } |\beta| < \infty, (\beta \models R) \Rightarrow (\beta \models \text{Always\_since\_ (P, Q)})$$

- **Never_from_to_ (P, Q, R)** : une séquence σ satisfait cette propriété, si et seulement si, P ne s'est jamais produit dans tout intervalle minimal séparant une occurrence de Q et une occurrence de R. Soit formellement :

$$\sigma \models \text{Never\_from\_to\_ (P, Q, R)} \text{ ssi}$$

$$\forall \beta \mid \beta \leq \sigma \text{ et } |\beta| < \infty, (\beta \models R) \Rightarrow (\beta \models \text{Never\_since\_ (P, Q)})$$

- **Once_between_andlast_ (P, Q, Q)** : une séquence σ satisfait cette propriété, si et seulement si, pour toute intervalle séparant deux occurrences successives de l'événement Q, l'événement P s'est produit au moins une fois. Ceci s'exprime formellement comme suit :

$$\sigma \models \text{Once\_ between\_andlast\_ (P, Q, Q))} \text{ ssi}$$

$$\forall \beta \mid \beta \leq \sigma \text{ et } |\beta| < \infty, (\beta \models Q) \Rightarrow (\forall \alpha \mid \alpha < \beta, \alpha \models \text{Once\_since (P, Q))}$$

- **Onlyonce_from_to_ (P, Q, R)** : une séquence σ satisfait cette propriété, si et seulement si, pour tout intervalle borné par les événements P et Q, l'événement P s'est produit une et une seule fois :

$$\sigma \models \text{Onlyonce\_from\_to\_ (P, Q, R)} \text{ ssi}$$

$$\forall \beta \mid \beta \leq \sigma \text{ et } |\beta| < \infty, (\beta \models R) \Rightarrow (\beta \models \text{Onlyonce\_since\_ (P, Q))}$$

- **Never_before_ (P,Q)** : une séquence σ satisfait cette propriété, si et seulement si, pour tout préfixe β de σ satisfaisant la propriété d'intervalle Before (Q), β satisfait aussi l'opérateur Never (P) :

$\sigma \models$ Never_before_ (P, Q) ssi $\forall \beta \mid \beta \leq \sigma$ et $|\beta| < \infty$, ($\beta \models$ Before (Q)) $\Rightarrow$ ($\beta \models$ Never (P))

3.4 Conclusion

Nous avons vu, dans ce chapitre, que les propriétés temporelles relatives aux comportements des systèmes matériels peuvent être réparties, principalement, en deux classes : les propriétés de sûreté et les propriétés de vivacité. Dans la plus part des cas, les propriétés de sûreté définissant un langage régulier suffisent à spécifier le comportement des systèmes matériels. Nous avons ensuite montré que toute propriété de sûreté s'exprime à l'aide de propriétés de séquences finies : d'intervalle I_{obs}, de contexte C_{ont} et de réaction R_{eac}. Enfin, nous avons défini un ensemble d'opérateurs de base définissant les éléments de base d'un langage de spécification de propriétés de sûreté.

Chapitre 4

Vérification des propriétés de sûreté

Durant toutes les phases de la conception, la description d'un système matériel, à un niveau d'abstraction donné (de la spécification à la réalisation), peut être validé en vérifiant que cette description satisfait des propriétés cruciales de fonctionnement.

Dans ce chapitre, nous nous intéresserons au problème de la vérification des propriétés de sûreté dont le langage associé est régulier (chapitre 3) sur un système matériel. Ces propriétés de sûreté s'expriment à l'aide de propriétés de séquences finies. Nous montrons que le langage de ces propriétés de séquences finies peut être décrit par une machine d'états finis. A partir de ces machines, la machine d'états finis associée à la propriété de sûreté est construite. Ensuite, nous étudions le problème de la vérification de ces propriétés de sûreté.

4.1 Les machines associées aux propriétés de séquences finies

Une propriété de sûreté s'exprime à l'aide des propriétés de séquences finies : événements définis, propriétés d'intervalles et de réaction. Dans cette section, nous montrons que le langage des propriétés de séquences finies définies dans le chapitre 3 peut être décrit par une machine d'états finis. Nous donnons la définition suivante que nous utiliserons par la suite :

• **Définition.** Soient M une machine d'états finis définie par le 6-uplet (Q, E, S, δ, λ, q_0) et $\sigma = (e_0\, e_1 \cdots e_n)$ une séquence finie de *n* symboles d'entrées de E. On définit M (σ) comme le couple (s_n, q_n) où s_n et q_n sont respectivement le symbole de sortie et l'état de la machine suite à l'application de la séquence σ en entrée :

$$M\,(e_0\, e_1 \cdots e_n) = (s_n, q_n) \mid \lambda(e_i, q_{i-1}) = s_i \text{ et } \delta(e_i, q_{i-1}) = q_i\,,\ 0 < i \leq n$$

4.1.1 Les machines associées aux événements définis

Rappelons que les événements définis relatifs aux comportement des systèmes matériels définissent un langage régulier L (V) caractérisé par un ensemble de séquences finies B :

$$L = \{V^* B\}$$

Nous donnons, dans ce qui suit, deux propriétés permettant de définir la machine décrivant le langage d'un événement défini.

Propriété 1. Le langage L (V) d'un événement défini P, caractérisé par une séquence finie β de longueur *n*, peut être décrit par une machine d'états finis ayant *n* états.

Preuve. Soit β une séquence finie de longueur *n* : $\beta = (b_1\, b_2 \cdots b_n)$ et soit M_p une machine d'états finis à *n* états définie comme suit :

$$M_p\,(V^*\beta) = (1, q_0) \text{ et } \forall\, \xi = (b_1\, b_2 \cdots b_i) < \beta \mid 1 \leq i < n,\ M_p\,(V^*\xi) = (0, q_{i+1}) \text{ et },$$

$$M_p\,(V^*\, b_1\, b_1^*) = (0, q_1)$$

En d'autres termes, pour toute séquence σ du langage satisfaisant un événement défini caractérisé par la séquence finie β, la machine M_p retourne la valeur booléenne 1, sinon la machine retourne la valeur 0 :

$$\forall\, \sigma, \sigma \in V^* \qquad (M_p\,(\sigma) = (1, q_0)) \quad \text{ssi} \quad (\sigma = V^*\beta)$$

D'où, la machine M_p décrit le langage de l'événement P.

Propriété 2. Le langage d'un événement défini caractérisé par un ensemble de *m* séquences finies : $B = \{\beta_1 + \beta_2 + \cdots + \beta_m\}$, peut être décrit par la machine produit $\times^i M_{i,p}$, $1 \leq i \leq m$ (figure 4.1); où chaque machine $M_{i,p}$ décrit le langage d'un événement défini caractérisé par une séquence finie β_i, $1 \leq i \leq m$.

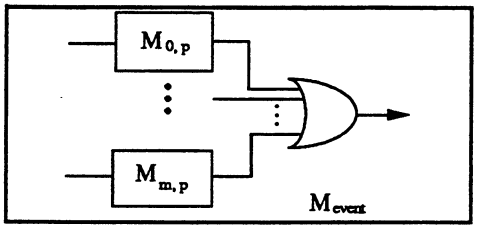


Figure 4.1. La machine décrivant un événement défini à *m* séquences finies

Preuve. La machine produit $\times^i M_{i,p}$ illustrée par la figure 4.1 retourne 1, si et seulement si, au moins une machine $M_{i,p}$, $1 \leq i \leq m$, retourne 1. En d'autre termes, la machine produit retourne 1, ssi, $\sigma = V^*(\beta_1 + \beta_2 + \cdots + \beta_m)$:

$$\forall\, \sigma, \sigma \in V^* \quad \times^i M_{i,p}(\sigma) = (1, q) \mid q \in (\times^i Q_i)) \quad \text{ssi} \quad \sigma = V^*(\beta_1 + \beta_2 + \cdots + \beta_m)$$

D'où, la machine produit $\times^i M_{i,p}$, $1 \leq i \leq m$, décrit le langage d'un événement défini caractérisé par un ensemble de *m* séquences finies.

4.1.2 Les machines associées aux propriétés d'intervalles et de réactions

Nous avons défini au chapitre 3 des propriétés de séquences finies de base : d'intervalles *After (P)* , *From_to_(P, Q)* et de réactions *Once_since_ (P, Q)*, *Onlyonce_since_ (P, Q)*. Nous allons montrer, dans ce qui suit, que les langages de ces propriétés primitives peuvent être décrits par des machines d'états finis notées respectivement : M_{after}, M_{from}, M_{once} et M_{only}. Pour une définition plus aisée de ces machines, nous considérons ces propriétés sur des événements dont le langage est constitué de suites de booléens. Un événement P est caractérisé par un booléen. La valeur booléenne 1 dénote sa présence.

- La propriété **After (P)** est satisfaite par une séquence σ, si et seulement si, il existe un préfixe de σ où l'événement P est présent :

$$\sigma \models \text{After (P)} \quad \text{ssi} \quad \exists\, \beta \leq \sigma, \beta \models P$$

La machine M_{after} décrivant le langage de cette propriété possède deux états : q_0 et q_1. Cette machine dont le graphe d'états est donné en figure 4.2 est définie par le 6-uplet $(Q, \{0, 1\}, \{0, 1\}, \delta_{after}, \lambda_{after}, q_0)$ comme suit :

$$\forall\ \sigma, \qquad \sigma \not\models \text{After (P)} \Leftrightarrow \forall\ \beta \leq \sigma,\ M_{after}(\beta) = (0, q_0)$$

$$\forall\ \sigma, \qquad \sigma \models \text{After (P)} \Leftrightarrow \forall\ \beta \geq \sigma,\ M_{after}(\beta) = (1, q_1)$$

• La propriété **From_to_ (P, Q)** est satisfaite par une séquence σ, si et seulement si :

$$(\exists\ \beta \leq \sigma, \beta \models P) \text{ et } (\forall\ \alpha \mid \beta \leq \alpha \leq \sigma, \alpha \not\models Q)$$

La machine $M_{from} = (Q, \{0, 1\}, \{0, 1\}, \delta_{from}, \lambda_{from}, q_0)$ décrivant le langage de cette propriété possède trois états : q_0, q_1 et q_2. Cette machine dont le graphe d'états est donné en figure 4.3 est définie comme suit :

$$\forall\ \sigma, \sigma \models \text{From\_to\_ (P, Q)} \Leftrightarrow M_{from}(\sigma) = (1, q_2)$$

$$\forall\ \sigma\ \forall\ \beta \mid \beta < \sigma,\ (\sigma \not\models \text{From\_to\_ (P, Q) et } \beta \not\models \text{From\_to\_ (P, Q)}) \Leftrightarrow M_{from}(\sigma) = (0, q_0)$$

$$\forall\ \sigma\ \exists\ \beta \mid \beta < \sigma,\ (\sigma \not\models \text{From\_to\_ (P, Q) et } \beta \models \text{From\_to\_ (P, Q)}) \Leftrightarrow M_{from}(\sigma) = (0, q_1)$$

• La propriété **Once_since_ (P, Q)** est satisfaite par une séquence, si et seulement si :

$$(\forall\ \beta \mid \beta \leq \sigma, \beta \not\models Q) \text{ ou}$$

$$((\exists\ \beta \mid \beta \leq \sigma, \beta \models Q) \text{ et } (\forall\ \alpha \mid \beta < \alpha \leq \sigma, \alpha \not\models Q)) \text{ et } (\exists\ \delta \mid \beta \leq \delta \leq \sigma, \delta \models P))$$

La machine $M_{once} = (Q, \{0, 1\}, \{0, 1\}, \delta_{once}, \lambda_{once}, q_0)$, décrivant le langage de cette propriété possède trois états : q_0, q_1 et q_2. Cette machine dont le graphe d'états est donné en figure 4.4 est définie comme suit :

$$\forall\ \sigma, (\sigma \models \text{Once\_since\_ (P, Q) et } \forall\ \beta \mid \beta < \sigma, \beta \not\models Q) \Leftrightarrow M_{once}(\sigma) = (1, q_0)$$

$$\forall\ \sigma, (\sigma \models \text{Once\_since\_ (P, Q) et } \exists\ \beta \mid \beta < \sigma, \beta \models Q) \Leftrightarrow M_{once}(\sigma) = (1, q_1)$$

$$\forall\ \sigma, \qquad (\sigma \not\models \text{Once\_since\_ (P, Q)} \Leftrightarrow M_{once}(\sigma) = (0, q_2)$$

• La propriété **Onlyonce_since_ (P, Q)** est satisfaite par une séquence, si et seulement si :

$(\forall \beta \mid \beta \leq \sigma, \beta \not\models Q)$ ou

$((\exists \beta \mid \beta \leq \sigma, \beta \models Q)$ et $(\forall \alpha \mid \beta < \alpha \leq \sigma, \alpha \not\models Q))$ et $(\exists! \delta \mid \beta \leq \delta \leq \sigma, \delta \models P))$

La machine $M_{only} = (Q, \{0, 1\}, \{0, 1\}, \delta_{only}, \lambda_{only}, q_0)$, décrivant le langage de cette propriété possède trois états : q_0, q_1 et q_2. Cette machine dont le graphe d'états est donné en figure 4.5 est définie comme suit :

$\forall \sigma, (\sigma \models$ Onlyonce_since_ (P, Q) et $\forall \beta \mid \beta < \sigma, \beta \not\models Q) \quad \Leftrightarrow \quad M_{only}(\sigma) = (1, q_0)$

$\forall \sigma, (\sigma \models$ Onlyonce_since_ (P, Q) et $\exists \beta \mid \beta < \sigma, \beta \models Q) \quad \Leftrightarrow \quad M_{only}(\sigma) = (1, q_1)$

$\forall \sigma, \quad (\sigma \not\models$ Onlyonce_since_ (P, Q) $\quad \Leftrightarrow \quad M_{only}(\sigma) = (0, q_2)$

Enfin, notons que les machines d'états finis décrivant le langage des propriétés de séquences finies, définies à partir des propriétés primitives, sont déduites des machines décrivant le langage de ces propriétés primitives. Par exemple, la propriété d'intervalle Before (P) est définie à partir de la propriété primitive d'intervalle After (P), une séquence σ satisfait la propriété Before (P) si et seulement si la séquence σ ne satisfait pas la propriété After (P) :

$\sigma \models$ Before (P) ssi $\sigma \not\models$ After (P)

La machine M_{before} décrivant le langage de la propriété Before (P) est alors définie, à partir de la machine M_{after}, comme suit :

$M_{after}(\sigma) = (0, q_0) \quad \Leftrightarrow \quad M_{before}(\sigma) = (1, q_0)$

$M_{after}(\sigma) = (1, q_1) \quad \Leftrightarrow \quad M_{before}(\sigma) = (0, q_1)$

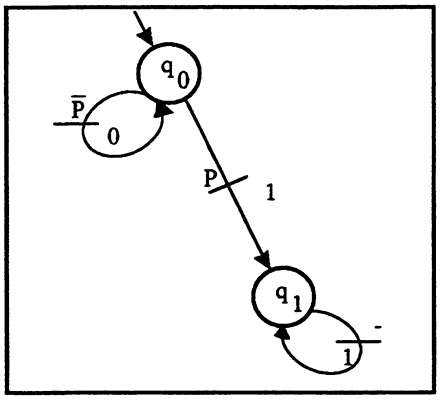


Figure 4.2. Graphe d'états de After (P)

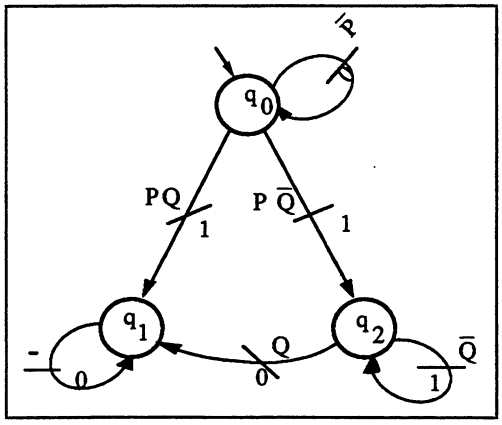


Figure 4.3. Graphe d'états de From_to_(P, Q)

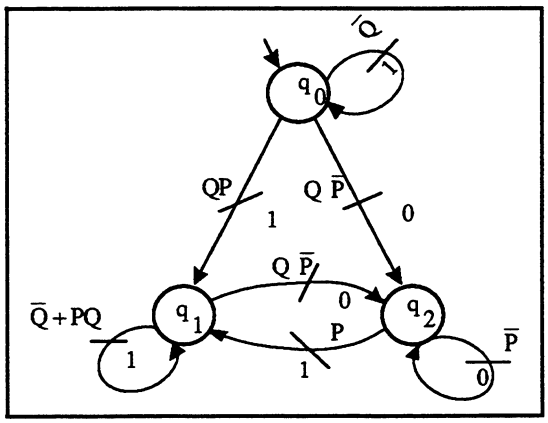


Figure 4.4. Graphe d'états de Once_since_ (P, Q)

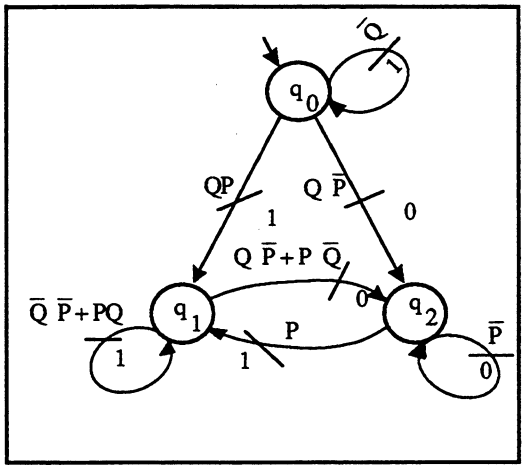


Figure 4.4. Graphe d'états de Onlyonce_since_ (P, Q)

4.2 Les machines associées aux propriétés de sûreté

Rappelons que les propriétés de séquences finies d'intervalle d'observation, de contexte et de réaction sont les paramètres temporels d'une propriété de sûreté P_s telle qu'une séquence σ satisfait la propriété P_s, si et seulement si, pour tout préfixe β de σ satisfaisant la propriété d'intervalle I_{obs}, s'il existe un préfixe α de β satisfaisant la propriété de contexte C_{ont} alors le préfixe α satisfait aussi la propriété de réaction R_{eac} :

$$\sigma \models P_s \quad \text{ssi} \quad (\forall\, \beta \mid \beta \leq \sigma, \beta \models I_{obs}) \Rightarrow ((\forall\, \alpha \mid \alpha \leq \beta, \alpha \models C_{ont}) \Rightarrow (\alpha \models R_{eac}))$$

Nous venons de voir (§ 4.1) que les propriétés de séquences finies données dans le chapitre 3 définissent un langage qui peut être décrit par une machine d'états finis M_p. La machine M_p est

définie par le 6-uplet $(Q, E, \{0, 1\}, \delta_p, \lambda_p, q_0)$ telle qu'une séquence finie σ satisfait la propriété P_{rop}, si et seulement si, la machine M_p retourne sur son unique sortie la valeur booléenne 1 :

$$\sigma \models P_{rop} \quad \Leftrightarrow \quad M_P(\sigma) = (1, q) \mid q \in Q$$

Ceci nous amène à définir la machine d'états finis M_{Ps} décrivant le langage de la propriété de sûreté. La machine M_{Ps} est caractérisée par le 6-uplet $(Q_{Ps}, E \times S, \{0, 1\}, \delta_{Ps}, \lambda_{Ps}, q_0)$. Une séquence σ de $(E \times S)^*$ satisfait la propriété de sûreté, si et seulement si, la machine "réaction" M_{reac} retourne 1, lorsque la machine "contexte" M_{cont} et la machine "intervalle d'observation" M_{obs} retournent 1. Soit formellement :

$$M_{Ps}(\sigma) = (1, q) \Leftrightarrow [(M_{obs}(\sigma) = (1, q_o)) \Rightarrow ((M_{cont}(\sigma) = (1, q_c)) \Rightarrow (M_{reac}(\sigma) = (1, q_r))] \quad (1)$$

où les états q, q_o, q_c, et q_r sont respectivement des états dans l'espace d'états des machines M_{Ps}, M_{obs}, M_{cont}, et M_{reac}.

Comme toute propriété de sûreté est un langage clos par préfixe, la machine d'états finis M_{Ps} décrivant le langage de la propriété retourne la valeur booléenne 1 pour tout préfixe fini d'une séquence satisfaisant la propriété :

$$(M_{Ps}(\sigma) = (1, q) \mid q \in Q_{Ps}) \quad \Rightarrow \quad (\forall \beta \mid \beta \leq \sigma \text{ et } |\beta| < \infty, M_{Ps}(\beta) = (1, q') \mid q' \in Q_{Ps})$$

Par conséquent, la machine M_{Ps} possède un état puits dans son espace d'états. Cet état est accessible par toute séquence σ ne satisfaisant pas la propriété :

$$M_{Ps}(\sigma) = (0, q_f) \quad \Rightarrow \quad (\forall \delta, M_{Ps}(\sigma\delta) = (0, q_f)), \text{ où } q_f \text{ est l'état puit} \quad (2)$$

Nous déduisons de (1) et (2) la structure de la machine M_{Ps} donnée en figure 4.6. La machine Al (de l'anglais, Always) met la machine M_{Ps} dans un état puits lorsqu'une séquence σ ne satisfait pas la propriété.

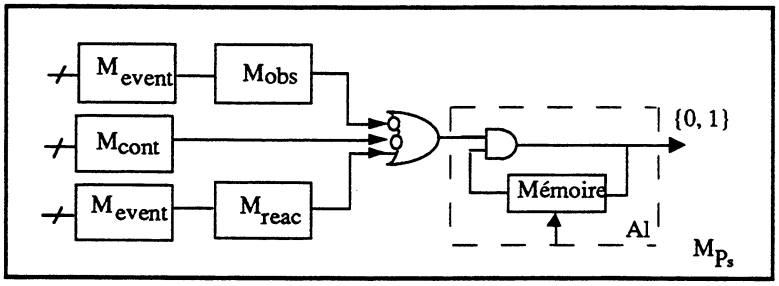


Figure 4.6. La machine décrivant le langage d'une propriété de sûreté

4.3 La méthode de vérification

Nous avons vu au chapitre 1 que toute description d'un système matériel synchrone dénote une machine d'états finis $M_d = (Q_d, E, S, \delta_d, \lambda_d, q_0)$. Nous venons de montrer que le langage d'une propriété de sûreté d'un vocabulaire $(E \times S)$ dont le langage associé est régulier peut être décrit par une machine d'états finis $M_{Ps} = (Q_{Ps}, E \times S, \{0, 1\}, \delta_{Ps}, \lambda_{Ps}, q_0)$. Ceci nous amène à définir une machine de vérification ou d'observation, notée M_v, issue de l'interconnexion de la machine "système matériel" M_d et de la machine "propriété de sûreté" M_{Ps}, comme le montre la figure 4.7. La machine M_v est définie par le 6-uplet $(Q_v, E_v, Flag, \delta_v, \lambda_v, q_0)$ où :

- Q_v est l'ensemble fini $Q_d \times Q_{Ps}$;
- q_0 est l'état initial de la machine produit tel que $q_0 = (q_0 (M_d), q_0 (M_{Ps}))$;
- δ_v (respectivement λ_v) est la fonction de transition (resp. fonction de sortie), définie de $E_v \times Q_{Ps} \times Q_d$ dans Q_v (resp. dans $\{0,1\}$) ;
- E_v est l'ensemble fini de symboles d'entrées du système matériel;
- Flag est l'unique sortie à valeurs dans $\{0,1\}$.

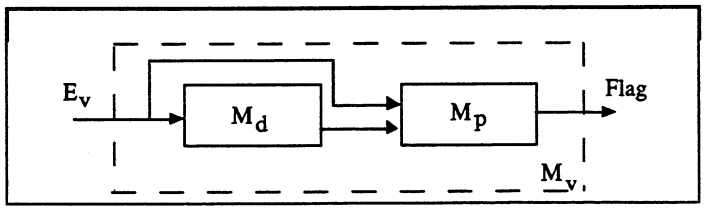


Figure 4.7. La machine d'observation

La vérification peut consister alors à parcourir l'espace d'états de la machine d'observation et à vérifier que la sortie Flag vaut la valeur booléenne 1, dans tous les états accessibles à partir de l'état initial. Un système matériel S dont le langage d'entrée est E^*, satisfera une propriété de sûreté P_s, si et seulement si, la machine d'observation M_v génère à partir de l'état initial le langage 1^* :

$$S \models P_s \Leftrightarrow (\forall \sigma \mid \sigma \in E^*, \quad M_v (\sigma) = (1, q) \mid q \in Q_v)$$

4.4 Conclusion

Nous avons décrit dans ce chapitre une approche de vérification des propriétés de sûreté sur une description d'un système matériel. Dans cette approche de vérification, nous définissons une machine d'observation issue de l'interconnexion de la machine associée à la description du système matériel et de la machine décrivant le langage de la propriété de sûreté. La vérification de propriétés sera réalisée sur cette machine d'observation.

Chapitre 5

Vérification de conformité

La vérification de conformité consiste à vérifier la validité d'une réalisation vis-à-vis de sa spécification donnée sous forme d'une description complète du fonctionnement attendu. Les deux descriptions : réalisation et spécification, dénotent deux machines d'états finis de niveaux d'abstraction différents. En effet, une spécification correspond à un niveau d'abstraction sans horloges explicites alors qu'une réalisation peut avoir plusieurs horloges. D'autre part, la réalisation et la spécification peuvent être de structures (architectures) différentes. Ces structures peuvent être : parallèles, séries ou pipelines. La conformité de la réalisation par rapport à sa spécification ne peut pas alors être vérifiée par une preuve d'équivalence directe ou bisimulation [CGPS 91, Boo 67, Koh 78, Mou 92, Par 81], entre les deux machines d'états finis dénotées par les deux descriptions (spécification et réalisation).

De façon générale, la réalisation d'un système matériel séquentiel synchrone est cadencée par un vecteur d'horloge noté H, tel que : $H = (h_1, h_2, ..., h_n)$, où les h_i sont appelées les phases de l'horloge. Une réalisation peut être vue comme l'interconnexion de plusieurs machines $M_{i\ [i = 1, 2, ..., n]}$; chacune de ces machines évoluant sur une phase d'horloge h_i. Rappelons qu'une telle machine est caractérisée par le 6-uplet $(Q, E \times \{h_1\} \times ... \times \{h_n\}, S, \delta, \lambda, q_0)$ (chapitre 2), où les fonctions de transition et de sortie sont définies comme suit :

$$\forall\ (e, \{1\}^n, q) \in \{E \times \{1\}^n \times Q\},\quad \delta\ (e, \{1\}^n, q) = \delta\ (e, q) \text{ et,}$$
$$\lambda\ (e, \{1\}^n, q) = \lambda\ (e, q)\ ;$$

$$\forall\ (e, \{0\}^n, q) \in \{E \times \{0\}^n \times Q\},\quad \delta\ (e, \{0\}^n, q) = q \text{ et,}$$
$$\lambda\ (e, \{0\}^n, q) = \lambda\ (e, q).$$

Le plus souvent, les réalisations sont mono-horloges où à horloges biphasées (bien adaptée à la technologie MOS) ; c'est-à-dire H = h_1 ou H = (h_1, h_2). Dans le cas d'une réalisation à horloges biphasées, les phases h_1 et h_2 sont exclusives et alternées. Dans ce chapitre, nous considérons uniquement le cas des réalisations à une seule phase d'horloge. Dans le chapitre suivant, nous étudierons de façon générale la vérification sous un environnement.

Dans le but d'obtenir deux machines comparables, ayant les mêmes vocabulaires d'entrée et de sortie et cadencées sur la même horloge, nous introduisons dans ce chapitre trois opérateurs de base. Ces opérateurs sont : l'opérateur de synchronisation Σ, l'opérateur de retard Δ et l'opérateur parallèle Π. L'opérateur de synchronisation Σ est appliqué lorsque les horloges de la réalisation sont explicites. L'opérateur de retard Δ est appliqué pour une réalisation à structure pipeline. Enfin, l'opérateur parallèle Π est utilisé lorsqu'une des deux machines (réalisation ou spécification) a une architecture série alors que l'autre est parallèle. L'application de ces opérateurs sur les flots d'entrée et / ou de sortie des deux machines initiales permet d'obtenir deux machines directement comparables, moyennant des critères de comparaison.

Dans ce chapitre, nous définissons d'abord les opérateurs Σ, Δ et Π. Puis, les différents cas de machines comparables sont étudiés. Ensuite, nous explicitons les critères de comparaison entre les flots de sortie des machines comparables. Enfin, nous abordons le problème de la vérification.

5.1 Les opérateurs de comparaison

Nous définissons, dans cette section, les opérateurs permettant d'obtenir à partir des machines associées aux descriptions de la spécification M_S et de la réalisation M_R, deux machines comparables. Nous donnons tout d'abord la définition formelle de chaque opérateur, puis les propriétés qui découlent de son application.

5.1.1 L'opérateur de synchronisation Σ

Dans le but de comparer les flots de sortie de deux descriptions d'un SMSS dont l'une évolue sur une horloge explicite, nous introduisons l'opérateur de synchronisation Σ donné dans la définition suivante. Cet opérateur permet de maintenir la machine d'états finis, associée à la description sans horloges explicites, dans le même état tant que le signal d'horloge est absent.

Définition. Soient Ω l'espace des machines d'états finis déterministes et ϕ un booléen. Nous appelons opérateur de synchronisation sur Ω par rapport à ϕ, l'application $\Sigma : \Omega \times \phi \rightarrow \Omega$ faisant correspondre à toute machine d'états finis $M = (Q, E, S, \delta, \lambda, q_0)$, la machine $M^S = (Q^S, E^S, S^S, \delta^S, \lambda^S, q_0^S)$ où :

- $E^S = E \times \phi$;
- $Q^S = Q \times S$;
- $\delta^S (e, \{1\}, (q, s)) = \delta (e, q)$ et $\delta^S (e, \{0\}, (q, s)) = q$;
- $\lambda^S (e, \{1\}, (q, s)) = \lambda (e, q)$ et $\lambda^S (e, \{0\}, (q, s)) = s$.

Cette définition nous donne les propriétés suivantes :

Propriété 1. Si M est une machine de Moore, le nombre d'états accessibles de la machine $\Sigma (M, \phi)$ est égal au nombre d'états de la machine M.

Preuve. Soit $M = (Q, E, S, \delta, \lambda, q_0)$ une machine de Moore. Toute sortie de la machine M dépend uniquement de l'ensemble d'états Q : $s = \lambda (q)$, la fonction de sortie λ^S de la machine $\Sigma (M, \phi)$ est alors définie comme suit :

$$\lambda^S (e, \{1\}, (q, s)) = \lambda (q) \text{ et,}$$

$$\lambda^S (e, \{0\}, (q, s)) = s$$

Comme chaque sortie est associée à un état, on a : $s = \lambda (q)$.

D'où les nombres d'états des machines M et $\Sigma (M, \phi)$ sont égaux, du fait que toute sortie de la machine $\Sigma (M, \phi)$ est calculée à partir d'un état de M.

Propriété 2. Si M est une machine de Mealy, le nombre d'états accessibles de la machine $\Sigma (M, \phi)$ peut être supérieur au nombre d'états accessibles de la machine M.

Preuve. Dans le cas d'une machine de Mealy M, deux transitions menant vers le même état peuvent générer deux sorties différentes. Un passage du modèle de Mealy vers le Modèle de Moore est alors nécessaire pour que toute sortie de la machine $\Sigma (M, \phi)$ puisse être calculée. Ce passage peut engendrer une augmentation du nombre d'états (chapitre 1, § 1.1).

5.1.2 L'opérateur de retard Δ

Dans le but de comparer les flots de sortie de deux descriptions d'un SMSS dont l'une a une structure pipeline, nous introduisons l'opérateur de retard Δ donné dans la définition suivante. Cet opérateur permet de décaler le flot de sortie de la machine d'états finis associée à la description à comparer avec celle ayant une structure pipeline.

Définition. Soient L un langage de suites de booléens et $\upsilon = (i_0, i_1, ..., i_n)$ une séquence finie de *n* booléens. Nous appelons opérateur de retard $\Delta_n(\upsilon)$ sur L, l'application faisant correspondre à toute séquence σ de L une séquence $\sigma_{\Delta n} = \upsilon\sigma$. En d'autres termes, l'opérateur Δ_n décale toute séquence σ vers la droite en concaténant la séquence initiale υ à sa gauche.

Propriété. L'opérateur Δ_n dénote une machine d'état finis $M_{\Delta n}$ comprenant 2^n états accessibles.

Preuve. Un registre à décalage à *n* éléments de mémorisation dont les valeurs initiales sont $(i_0, i_1, ..., i_n)$ dénote l'opérateur $\Delta_n(i_0, ..., i_{n-1})$. Tout état du registre est défini par une configuration donnée des *n* éléments de mémorisation. Quelle que soit la configuration initiale, toutes les 2^n configurations possibles d'un registre à décalage sont accessibles.

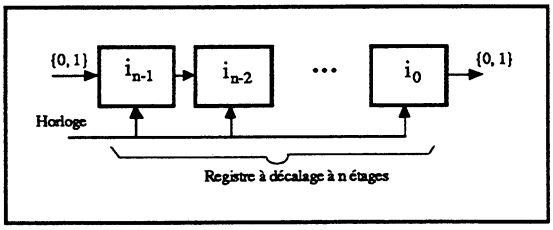


Figure 5.1. La machine $M_{\Delta n}$ associée à l'opérateur $\Delta_n((i_0, ..., i_{n-1}))$

5.1.3 L'opérateur parallèle Π

Nous introduisons l'opérateur parallèle dans le but de comparer une description ayant une structure parallèle avec une description ayant une structure série. Cet opérateur est donné par la définition suivante :

Définition. Soit L un langage de suites de booléens et $\upsilon = (i_0, i_1, ..., i_n)$ une séquence finie de *n* booléens. Nous appelons opérateur parallèle $\Pi_n(\upsilon)$ sur L, l'application faisant correspondre à toute séquence $\sigma = (s_0, ..., s_{n-1}, ..., s_{2n-1}, ...)$ de L une séquence

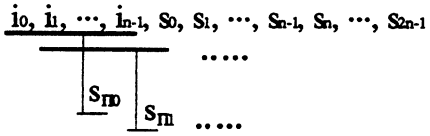
$\sigma_{\Pi n} = (s_{\Pi 0}, s_{\Pi 1}, ..., s_{\Pi n})$ où chaque symbole $s_{\Pi i}$ est un uplet de n booléens tel que :

$s_{\Pi 0} = (i_0, i_1, ..., i_{n-1})$, $s_{\Pi 1} = (i_1, i_2, ..., s_0)$, ...

$s_{\Pi n-1} = (s_0, s_1, ..., s_{n-1})$, $s_{\Pi n} = (s_1, ..., s_n)$, ...

$s_{\Pi 2n-1} = (s_n, ..., s_{2n-1})$, ...

En d'autre termes, l'opérateur $\Pi_n(\upsilon)$ comprime la séquence $(\upsilon\ \sigma)$ d'un facteur n en décalant à droite :



Propriété. L'opérateur Π_n dénote une machine d'état finis $M_{\pi n}$ comprenant 2^n états accessibles.

Preuve. Un registre à décalage à n éléments de mémorisation et à sorties parallèles (figure 5.2) dont les valeurs initiales sont $(i_0, i_1, ..., i_{n-1})$ dénote l'opérateur Π_n. Comme nous l'avons mentionné précédemment (§ 5.1.2), les 2^n états possibles d'un registre à décalage sont accessibles.

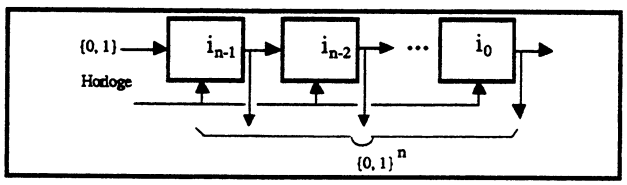


Figure 5.2. La machine $M_{\pi n}$ associée à l'opérateur $\Pi_n((i_0, i_1, ..., i_{n-1}))$

5.2 Les machines comparables

L'application des opérateurs Σ, Δ et Π sur les flots d'entrée et / ou de sortie des machines associées aux descriptions de la réalisation et de la spécification M_S et M_r permet d'obtenir deux machines comparables notées respectivement M_S^c et M_r^c. Ces machines ont d'une part, le même vocabulaire d'entrée et de sortie et d'autre part, elles sont cadencées par la même horloge. Ces machines comparables dépendent du niveau d'abstraction des descriptions de la réalisation et de la spécification et de leurs structures (parallèle, série ou pipeline). Nous allons, dans ce qui suit, étudier les différents cas de machines comparables.

5.2.1 Cas d'une réalisation et d'une spécification de même structure

Soient $M_S = (Q_S, E, S, \delta_S, \lambda_S, q_0)$, la machine d'états finis associée à une spécification d'un système matériel et $M_r = (Q_r, E \times h, S, \delta_r, \lambda_r, q_0)$ la machine d'états finis associée à une réalisation de même structure que la spécification.

La comparaison des flots de sorties des deux machines M_S et M_r par rapport au langage d'entrée $(E \times h)^*$ nécessite une synchronisation de la machine M_S par rapport à l'horloge h. La synchronisation est réalisée en définissant la machine $M_S^c = \Sigma (M_S, h)$. Les machines M_S^c et M_r sont alors directement comparables.

5.2.2 Cas d'une réalisation pipeline

Soit $M_r = (Q_r, \{0,1\}^p \times h, \{0,1\}^n, \delta_r, \lambda_r, q_0)$ la machine associée à une réalisation pipeline à k étages, à p lignes d'entrées et n lignes de sorties. La machine comparable M_S^c est obtenue en synchronisant la machine initiale sur l'horloge de la réalisation h et en retardant par la suite le flot de la machine synchronisée de k cycles de l'horloge h (figure 5.3) :

$$M_S^c = \Sigma (M_S, h) \times \Sigma (M_{\Delta k}, h)$$

Les machines M_S^c et M_r sont alors directement comparables par rapport au langage d'entrée $(\{0,1\}^{p+1})^*$.

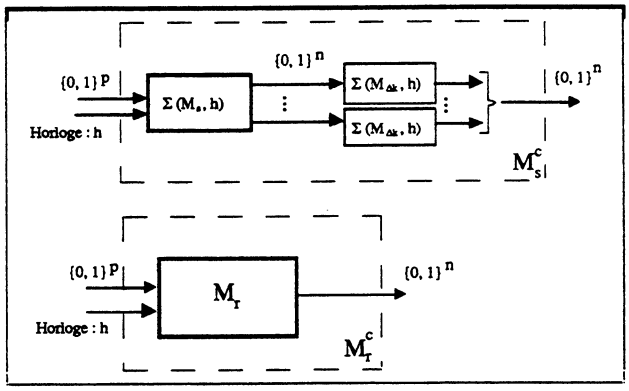


Figure 5.3. Les machines comparables : réalisation pipeline

5.2.3 Cas d'une réalisation parallèle et d'une spécification série

Soient $M_S = (Q_S, \{0,1\}, \{0,1\}, \delta_S, \lambda_S, q_0)$ la machine associée à une spécification série et $M_T = (Q_T, \{0,1\}^n \times h, \{0,1\}^n, \delta_T, \lambda_T, q_0)$ la machine associée à une réalisation parallèle, où n représente le facteur du parallélisme. Les machines comparables M_S^c et M_T^c (figure 6.4) sont obtenues en tranformant les machines initiales vers des machines ayant le même format : série en entrée et parallèle en sortie; c'est-à-dire des machines ayant $\{0,1\}$ comme vocabulaire d'entrée et $\{0,1\}^n$ comme vocabulaire de sortie :

$$M_S^c = M_S \times M_{\Pi n}$$

$$M_T^c = M_T \times M_{\Pi n} \times M_{count}$$

M_{count} est la machine associée au compteur modulo n, retournant la valeur booléenne 1 tous les n cycles.

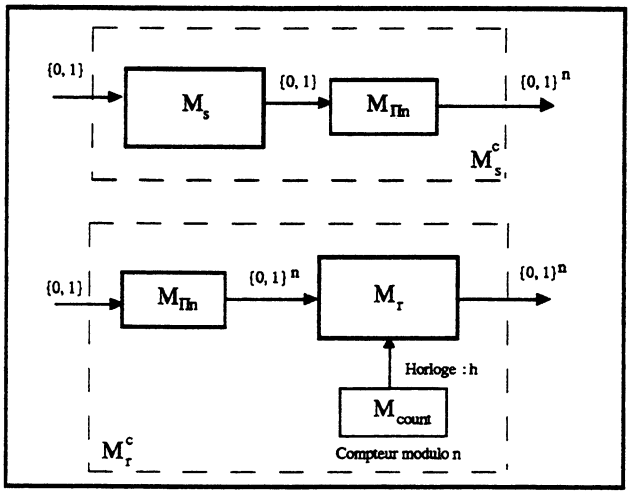


Figure 5.4. Les machines comparables : réalisation parallèle, spécification série

5.2.4 Cas d'une réalisation série et d'une spécification parallèle

Soit $M_S = (Q_S, \{0,1\}^n, \{0,1\}^n, \delta_S, \lambda_S, q_0)$ la machine associée à une spécification parallèle, où n est le facteur du parallélisme et $M_T = (Q_T, \{0,1\} \times \{0,1\}, \{0,1\}, \delta_T, \lambda_T, q_0)$ la machine associée à une réalisation série. Les machines comparables M_S^c et M_T^c (figure 5.5) sont obtenues en transformant les machines initiales en des machines ayant le même format :

série en entrée et parallèle en sortie; c'est-à-dire des machines ayant {0,1} comme vocabulaire d'entrée et $\{0,1\}^n$ comme vocabulaire de sortie :

$$M_S^c = \Sigma\,(M_S,\ h) \times \Sigma\,(M_{\Pi n},\ \Sigma\,(M_{count},\ h)) \times \Sigma\,(M_{count},\ h)$$

$$M_r^c = M_r \times \Sigma\,(M_{\Pi n},\ h)$$

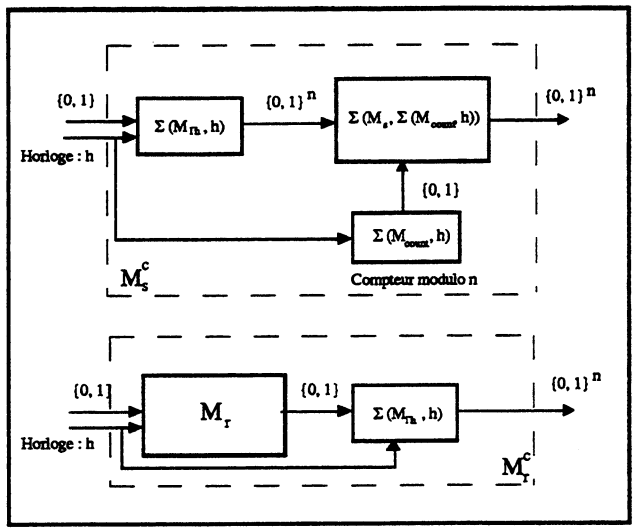


Figure 5.5. Les machines comparables : réalisation série, spécification parallèle

5.3 Critères d'observation

Nous venons de voir que lorsqu'une des deux machines associée à la spécification M_S ou à la réalisation M_r a une architecture série alors que l'autre est parallèle, nous les transformons en machines comparables M_S^c et M_r^c ayant le même format : série en entrée et parallèle en sortie. Cependant, la comparaison entre les flots de sortie de ces machines comparables n'est pas directe. En effet, la comparaison entre les flots de sortie des deux machines comparables est significative tous les *n* cycles d'horloge ; où *n* est le facteur du parallélisme. La comparaison entre les flots de sortie des machines M_S^c et M_r^c est donc réalisée à des instants bien définis.

Nous définissons aussi ces instants de comparaison lorsque nous nous intéressons à la conformité d'une réalisation par rapport à sa spécification, lorsque des conditions particulières de fonctionnement sont satisfaites. Par exemple, nous nous intéressons à la conformité d'une réalisation d'un microprocesseur par rapport à sa spécification pendant les cycles d'écriture. Le critère d'observation, dans ce cas, est la plage temporelle séparant l'instant de détection du code d'écriture et un événement caractérisant la fin du cycle d'écriture. Nous désignerons ces instants ou plages temporelles de comparaison par critères d'observation.

Ces critères d'observation (instants ou plages temporelles), notés C_{obs}, sont des événements définis ou des intervalles temporels bornés par des événements définis. D'où, formellement, il s'agit de propriétés de séquences finies d'événements définis ou d'intervalles (chapitre 3) telles que :

- une séquence σ satisfait un critère relatif à un événement défini P, si et seulement si, l'événement défini a lieu :

$$\sigma \models C_{obs}(P) \qquad \text{ssi} \qquad \sigma \models P$$

- une séquence σ satisfait un critère relatif à un intervalle temporel borné par les événements définis P et Q, si et seulement si, σ satisfait la propriété d'intervalle From_to (P, Q) (chapitre 3) :

$$\sigma \models C_{obs}([P \; Q[) \quad \text{ssi} \quad \sigma \models \text{From\_to}(P, Q)$$

Rappelons que le langage de toute propriété de séquences finies P_{rop} peut être décrit par une machine d'états finis M_p (chapitre 4) telle qu'une séquence σ satisfait la propriété P_{rop}, si et seulement si, la machine M_p retourne toujours la valeur booléenne 1 :

$$\forall \sigma, \quad (\sigma \models P) \quad \Leftrightarrow \quad M_p(\sigma) = (1, q)\text{, où q est un état de la machine } M_p$$

Par conséquent, nous associons à tout critère d'observation C_{obs} une machine d'états finis $M_c = (Q_c, E_c, \{0, 1\}, \delta_c, \lambda_c, q_0)$, telle qu'une séquence σ satisfait le critère d'observation, si et seulement si , M_c retourne la valeur booléenne 1 :

$$\forall \sigma \mid \sigma \in E_c^*, \quad (\sigma \models C_{obs}) \quad \Leftrightarrow \quad M_c(\sigma) = (1, q), \mid q \in Q_c$$

5.4 La méthode de vérification

La méthode de vérification présentée ici est similaire à celle décrite dans le chapitre 4. En effet, nous définissons une machine de vérification ou d'observation M_v (figure 5.6) ayant une unique sortie notée Flag. La machine M_v est issue de l'interconnexion des machines comparables : M_S^c, M_r^c et de la machine associée aux critères d'observation M_c. La machine M_v est caractérisée par le 6-uplet $(Q_v, E_v, \{0, 1\}, \delta_v, \lambda_v, q_0)$ où :

- Q_v est l'ensemble fini $Q_S^c \times Q_r^c \times Q_c$;

- q_0 est l'état initial de la machine produit tel que $q_0 = (q_0(M_S^c), q_0(M_r^c), q_0(M_c))$;

- δ_V (respectivement λ_V) est la fonction de transition (resp. fonction de sortie), définie de $E_V \times Q_S^c \times Q_r^c \times Q_c$ dans Q_V (resp. dans $\{0,1\}$) ;
- E_V est le vocabulaire d'entrée des machines comparables.

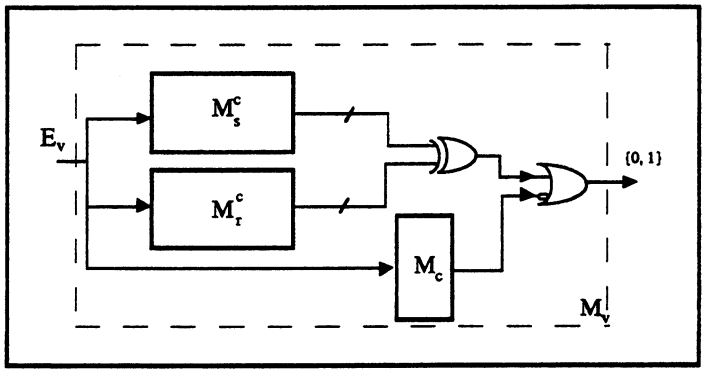


Figure 5.6. La machine d'observation M_V

Une réalisation R sera conforme à sa spécification S, ce qui est noté S |= R, si et seulement si pour toute séquence σ appartenant au langage d'entrée $(E_V)^*$, la machine d'observation M_V génère à partir de l'état initial le langage 1^* :

$$S \models R \quad \Leftrightarrow \quad (\forall \sigma | \sigma \in (E_V)^*,\ M_V(\sigma) = (1, q) \mid q \in Q_V)$$

La vérification de conformité peut consister alors à vérifier que la fonction de sortie Flag est toujours vraie dans tous les états accessibles à partir de l'état initial.

5.5 Conclusion

Pour comparer une réalisation et une spécification de niveaux d'abstraction différents et / ou de structures différentes, nous avons introduit dans ce chapitre trois opérateurs de base. Ces opérateurs sont : l'opérateur de synchronisation Σ, l'opérateur de retard Δ et l'opérateur parallèle Π. L'application de ces opérateurs sur les flots d'entrée et / ou de sortie des machines associées à la réalisation et à la spécification permet d'obtenir deux machines comparables, moyennant des critères de comparaison. Nous avons montré ensuite que la vérification de conformité, comme dans le cas de la vérification des propriétés de sûreté, consiste à définir une machine d'observation ayant une unique sortie, notée Flag, sur laquelle la vérification sera réalisée.

Chapitre 6

Vérification sous un environnement

Un système matériel est généralement intégré dans un environnement, celui-ci fournit au système les séquences d'entrées que ce système doit traiter. Durant le processus de vérification, il est nécessaire de tenir compte de l'environnement du système. En effet, d'une part, la conception d'un système matériel est généralement faite par rapport à un environnement dans lequel il va fonctionner. Ainsi, le bon fonctionnement du système dépend d'un séquencement adéquat des signaux issus de cet environnement. Par exemple, une réalisation à horloges biphasées fonctionne correctement si les deux phases d'horloges sont toujours exclusives et alternées. D'autre part, la spécification d'une propriété temporelle peut être faite par rapport à un environnement de fonctionnement spécifique à cette propriété. Par exemple, pour un arbitre de bus la propriété : "l'émission d'un acquittement doit se faire en direction de l'unité la plus prioritaire ayant émis une requête d'allocation du bus", est spécifiée par rapport à l'environnement suivant : toute requête de bus doit être maintenue tant que l'arbitre ne retourne pas un acquittement.

Dans ce chapitre, nous allons définir d'abord les notions de langage d'entrée accepté par un système matériel et de propriétés d'environnement. Ensuite, nous donnons deux approches de modélisation de l'environnement d'un système matériel sous forme de machines d'états finis : un *reconnaisseur* décrivant le langage de la propriété d'environnement et un *générateur* du langage accepté par un système matériel. Enfin, nous abordons le problème de la vérification sous un environnement avec ces deux approches de modélisation.

6.1 Le langage accepté et les propriétés d'environnement

La prise en compte de l'environnement d'un système matériel, durant le processus de vérification, consiste à réaliser la vérification par rapport à un langage d'entrée spécifique (un sous-langage de E^*). Nous désignons par L_a le langage d'entrée accepté par un système matériel. Dans l'exemple de la bascule RS, les deux entrées *R* et *S* ne doivent jamais être à la valeur booléenne 1 en même temps. Le langage L_a de la bascule RS est l'ensemble des séquences d'entrées pour lesquelles les entrées *R* et *S* sont exclusives :

$$L_a = ((0, 0) + (1, 0) + (0, 1))^*$$

Ce langage L_a peut dépendre du langage de sortie du système matériel. Par exemple, le langage L_a d'un arbitre de bus est tel que toute demande de bus est maintenue tant que l'arbitre ne retourne pas un acquittement. Ceci nous amène à introduire la notion de propriété d'environnement notée P_e, définissant un sous-langage de $(E \times S)^*$. Par exemple, la propriété : "l'entrée *a* d'un circuit doit être maintenue au niveau logique 1 tant que la sortie *b* est au niveau logique 0" est une propriété d'environnement. Généralement, ces propriétés d'environnement sont des propriétés de sûreté dont le langage associé est régulier. Ces propriétés sont du type des propriétés que nous avons décrites au chapitre 3.

6.2 Modélisation de l'environnement d'un système matériel

Le langage d'entrée L_a accepté par un système matériel et les propriétés d'environnement que nous venons de définir nous amène à proposer deux approches de modélisation de l'environnement d'un système matériel, sous forme de machines d'états finis. La première approche consiste à définir un *reconnaisseur* de langage défini par la propriété d'environnement. Dans la seconde approche, nous définissons un *générateur* du langage L_a.

6.2.1 Le modèle reconnaisseur

Dans cette approche de modélisation, nous définissons la machine d'états finis décrivant le langage de la propriété d'environnement P_e (chapitre 4). Nous désignons cette machine, notée M_r, par reconnaisseur du langage de la propriété d'environnement. La machine M_r est caractérisée par le 6-uplet $(Q_r, E \times S, \{0, 1\}, \delta_r, \lambda_r, q_0)$ tel qu'une séquence σ satisfait la propriété d'environnement P_e, si et seulement si, la machine M_r retourne toujours la valeur

booléenne 1. Sinon la machine retourne 0 et transite vers un état puits q_p :

$$\forall \sigma, \sigma \in (E \times S)^* \qquad (\sigma \models P_e) \quad \Leftrightarrow \quad M_r(\sigma) = (1, q) \mid q \in Q_r$$

$$(\sigma \not\models P_e) \quad \Leftrightarrow \quad M_r(\sigma) = (0, q_p)$$

Nous distinguons le cas particulier donné dans la définition suivante :

Définition. Nous appelons un reconnaisseur non causal, le reconnaisseur ayant un état de transition q_t dans son espace d'états qui mène uniquement vers l'état puits :

$$\exists\ q_t \in Q' \mid \forall\ e \in E,\ \delta_r(q_t, e) = q_p$$

Un reconnaisseur causal sera donc un reconnaisseur ne possédant pas d'états de transition menant uniquement vers l'états puits.

Exemple. Soit l'environnement suivant : l'entrée *a* d'un circuit est maintenue à 1 tant que le circuit génère le langage 0^* sur sa sortie *b*. La propriété d'environnement P_e spécifiant cet environnement est la propriété de sûreté : "Always (a, a, b)" (chapitre 3, § 3.3.5), décrivant le sous-langage L de $(\{0,1\} \times \{0,1\})^*$ défini comme suit :

$$L = (((0, 0) + (0, 1) + (1, 1))^* (1, 0) (1, 0)^* ((1, 1) + (0, 1)))^*$$

La machine d'états finis M_r décrivant ce langage est caractérisée par le 6-uplet $(\{q_0, q_1, q_p\}, \{0,1\} \times \{0,1\}, \{1, 0\}, \delta_r, \lambda_r, q_0)$ tel que : pour toute séquence σ de $(\{0,1\} \times \{0,1\})^*$ la machine M_r retourne 1, si et seulement si, elle transite vers les états $\{q_0, q_1\}$ (la séquence σ appartient à L). Sinon, la machine M_r retourne 0 et transite vers l'état puits q_p :

$$\forall \sigma, \sigma \in L \qquad M_r(\sigma) = (1, q) \mid q \in \{q_0, q_1\}.$$

$$\forall \sigma, \sigma \notin L \qquad M_r(\sigma) = (0, q_p)$$

Nous en déduisons la représentation du reconnaisseur M_r sous forme de graphe d'états (figure 6.1).

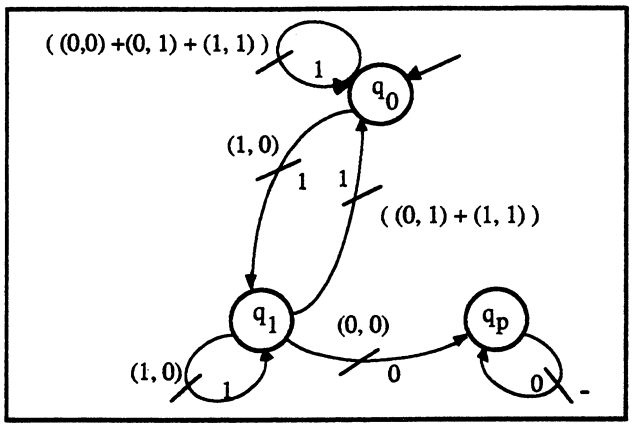


Figure 6.2. Graphe d'état du reconnaisseur de langage :
$(((0, 0) + (0, 1) + (1, 1))^* (1, 0) (1, 0)^* ((1, 1) + (0, 1)))^*$

6.2.2 Le modèle générateur

Dans cette approche de modélisation, nous définissons un générateur du langage L_a. Formellement, nous appelons générateur du langage L_a, l'application Γ définie de $(E \times S)^*$ dans L_a telle que :

$$\forall \sigma = ((e_0, s_0) \cdots (e_n, s_n)) \mid \sigma \in (E \times S)^* \quad \Gamma(\sigma) = (e_0 \cdots e_n) \quad \text{si} \quad (e_0 \cdots e_n) \in L_a$$

$$\Gamma(\sigma) = \beta \mid \beta \in L_a \quad \text{si} \quad (e_0 \cdots e_n) \notin L_a$$

Propriété. Le langage L_a peut être généré par une machine d'états finis.

Preuve. Soient P_e une propriété d'environnement définissant un sous-langage L de $(E \times S)^*$ et $M_r = (Q_r, E \times S, \{0, 1\}, \delta_r, \lambda_r, q_0)$ le reconnaisseur causal de ce langage. Considérons à présent la machine $M_g = (Q_g, E \times S, E_g, \delta_g, \lambda_g, q_0)$ ayant le même espace d'états que la machine M_r excepté l'états puits q_p : $Q = Q_r - \{q_p\}$. Nous définissons les fonctions de sortie et de transition δ_g et λ_g comme suit :

$$\forall ((e, s), q) \in (E \times S) \times Q, \qquad \lambda_g((e, s), q) = \begin{cases} e & si \ \lambda_r((e, s), q) = 1 \\ e' \mid \delta_r((e', s), q) \neq q_p & si \ \lambda_r((e, s), q) = 0 \end{cases}$$

$$\forall ((e, s), q) \in (E \times S) \times Q, \qquad \delta_g((e, s), q) = \begin{cases} \delta_r((e, s), q) & si \ \delta_r((e, s), q) \neq q_p \\ \delta_r(\lambda_g((e, s), q), q) & si \ \delta_r((e, s), q) = q_p \end{cases}$$

Par définition, la machine M_g génère, pour toute séquence $\sigma = ((e_0, s_0) \cdots (e_n, s_n))$ de $(E \times S)^*$, la séquence $(e_0, \cdots e_n)$, si et seulement si, la séquence σ satisfait la propriété d'environnement.

Sinon, la machine M_g génère une séquence correspondant à une séquence σ de $(E \times S)^*$ satisfaisant la propriété d'environnement. D'où, la machine M_g est génératrice du langage L_a.

Exemple. Pour illustrer cette deuxième approche, reprenons l'exemple précédent (§ 6.2.1) : l'entrée *a* d'un circuit est maintenue à 1 tant que la sortie *b* est à 0. Nous avons vu que le reconnaisseur du langage de la propriété de cet environnement est défini par le 6-uplet $(\{q_0, q_1, q_p\}, \{0,1\} \times \{0,1\}, \{1, 0\}, \delta_r, \lambda_r, q_0)$ (figure 6.2).

Ce reconnaisseur de langage permet de définir la machine d'états finis M_g générant le langage L_a. La machine générateur M_g est caractérisée par le 6_uplet $(\{q_0, q_1\}, \{0,1\} \times \{0,1\}, \{0, 1\}, \delta_g, \lambda_g, q_0)$ où les fonctions de sortie et de transition sont définies comme suit :

$$\forall ((e, s), q) \in (E \times S) \times Q, \qquad \lambda_g((e, s), q) = \begin{cases} e & si \ \lambda_r((e, s), q) = 1 \\ 1 & si \ (e, s), q) = ((0, 0), q_1) \end{cases}$$

$$\forall ((e, s), q) \in (E \times S) \times Q, \qquad \delta_g((e, s), q) = \begin{cases} \delta_r((e, s), q) & si \ \delta_r((e, s), q) \neq q_p \\ q_1 & si \ (e, s), q) = ((0, 0), q_1) \end{cases}$$

Nous en déduisons la représentation sous forme de graphe d'états du générateur M_g (figure 6.2).

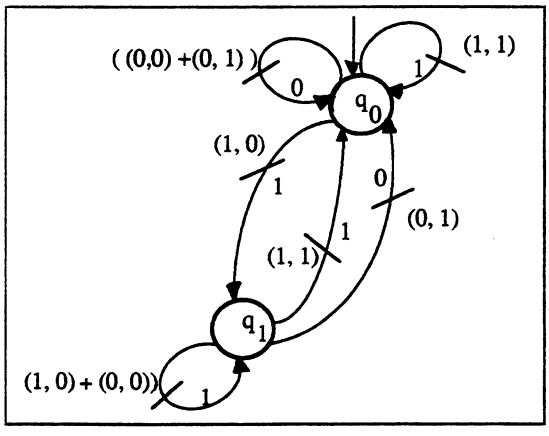


Figure 6.3. Graphe d'états du générateur du langage :
$(((0, 0) + (0, 1) + (1, 1))^* (1, 0) (1, 0)^* ((1, 1) + (0, 1)))^*$

6.3 La vérification sous un environnement

Rappelons d'abord que dans la méthode de vérification présentée pour résoudre les deux problèmes de vérification : la vérification de propriétés (chapitre 4) et la vérification de conformité (chapitre 5), nous définissons une machine d'observation M_v ayant une unique sortie notée *Flag*. La machine M_v a le vocabulaire d'entrée E du système matériel comme vocabulaire d'entrée. Le problème de la vérification se ramène à vérifier que pour toute séquence d'entrée σ de E^*, la machine M_v génère sur son unique sortie *Flag* le langage 1^*.

Dans cette section, nous allons étudier le cas général de la vérification sous un environnement, à l'aide des deux approches de modélisation de l'environnement sous forme de machines d'états finis : reconnaisseur du langage de la propriété d'environnement et générateur du langage L_a, présentées en section 6.2.

6.3.1 Vérification avec un modèle reconnaisseur de l'environnement

Nous venons de voir que les propriétés d'environnement sont des propriétés de sûreté définissant un sous-langage de $(E \times S)^*$ qui peut être décrit par une machine d'états finis (un reconnaisseur) notée M_r telle que : pour toute séquence σ de $(E \times S)^*$, la machine M_r retourne le langage 1^*, si et seulement si, σ satisfait la propriété d'environnement P_e.

Avec cette approche de modélisation, la vérification d'une propriété de sûreté P_s ou de la conformité d'une réalisation R par rapport à sa spécification S_{pec} est réalisée sur toute séquence σ de $(E \times S)^*$ satisfaisant la propriété d'environnement P_e :

$$\forall\, \sigma, \sigma \in (E \times S)^* \qquad (\sigma \models P_e) \quad \Rightarrow \quad \sigma \models P_s$$

$$\forall\, \sigma, \sigma \in (E \times S)^* \qquad (\sigma \models P_e) \quad \Rightarrow \quad S_{pec} \models R$$

Ceci nous amène à considérer la machine produit $M_{v/r} = (Q_{v/r}, E, \{0, 1\}, \delta_{v/r}, \lambda_{v/r}, q_0)$ issue de l'interconnexion des machines M_v (définie au chapitre 4 et 5) et M_r (figure 6.3). La fonction de sortie $\lambda_{v/e}$ est définie comme suit :

$$\forall\, ((e\ q) \in E \times Q_{v/r}, \quad \lambda_{v/r}((e, q) = \begin{cases} 1 & si \quad \lambda_r((e, s), q) = 0 \\ \lambda_v((e, q) & si \quad \lambda_r((e, s), q) = 1 \end{cases}$$

Lorsque le reconnaisseur est causal la vérification sous un environnement reconnaisseur consistera alors à parcourir l'espace d'états de la machine d'observation $M_{v/r}$ et à vérifier que l'unique sortie vaut la valeur booléenne 1, dans tous les états accessibles à partir de l'état initial. Dans le cas où le reconnaisseur est non causal, les états de transitions menant uniquement vers l'états puits sont considérés comme non valides.

D'une manière générale, un système matériel S satisfera une propriété de sûreté P_s ou une réalisation R sera conforme à sa spécification S_{pec}, si et seulement si, la machine d'observation $M_{v/r}$ génère à partir de l'état initial le langage 1^* pour toute séquence d'entrée ne menant pas vers un état de transition q_t (menant uniquement vers l'états puits) sur la machine reconnaisseur :

$$(S \models P_s \text{ ou } S_{pec} \models R) \Leftrightarrow (\forall \sigma \mid \sigma \in E^* \text{ et } M_r(\sigma) \neq (1, q_t),\ M_{v/r}(\sigma) = (1, q) \mid q \in Q_v)$$

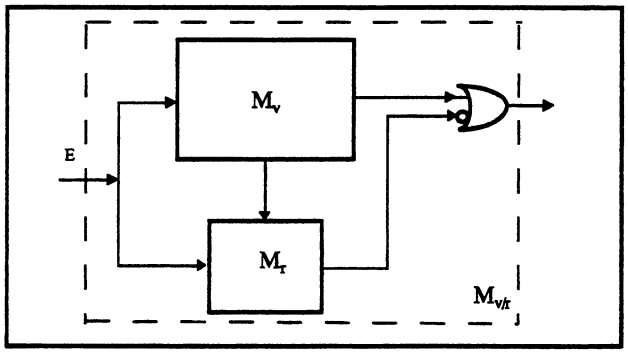


Figure 6.3. La machine d'observation sous un environnement reconnaisseur

6.3.2 Vérification avec un modèle générateur de l'environnement

Dans l'approche de modélisation de l'environnement sous forme de générateur, nous définissons une machine $M_g = (Q_g, E \times S, E_g, \delta_g, \lambda_g, q_0)$ générant le langage L_a telle que : pour toute séquence $\sigma = ((e_0, s_0) \cdots (e_n, s_n))$ de $(E \times S)^*$ satisfaisant la propriété de l'environnement, la machine M_g génère la séquence $(e_0 \cdots e_n)$. Sinon, la machine M_g génère une autre séquence appartenant au langage L_a.

Avec cette approche de modélisation de l'environnement, la vérification d'une propriété de sûreté P_s ou de la conformité d'une réalisation R par rapport à sa spécification S_{pec} est réalisée sur toute séquence σ de L_a :

$$(\forall \sigma, \sigma \in L_a) \Rightarrow \sigma \models P_e$$

$$(\forall \sigma, \sigma \in L_a) \Rightarrow S_{pec} \models R$$

Ceci nous amène à définir la machine produit $M_{v/g}$ (figure 6.4) issue de l'interconnexion de la machine générateur M_g et et de la machine d'observation M_v (définie au chapitre 4 et 5). La vérification se ramène ainsi à parcourir l'espace d'états de la machine $M_{v/g}$ et à vérifier que son unique sortie vaut toujours la valeur booléenne 1. Un système matériel S satisfera une propriété de sûreté P_s, ou une réalisation R sera conforme à sa spécification S_{pec}, si et seulement si, la machine d'observation $M_{v/g}$ génère à partir de son état initial le langage 1^* :

$$(S \models P_s \text{ ou } S_{pec} \models R) \quad \Leftrightarrow \quad (\forall \sigma \mid \sigma \in E^*, \quad M_{v/g}(\sigma) = (1, q) \mid q \in Q_v)$$

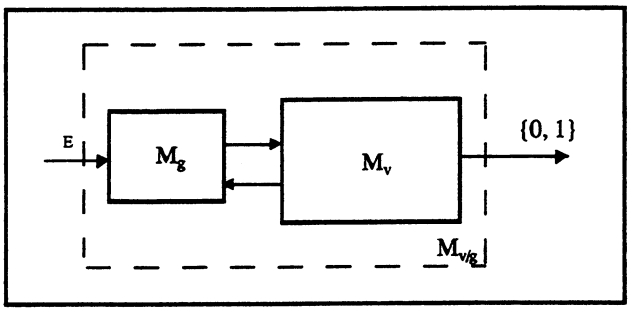


Figure 6.4. La machine d'observation sous un environnement générateur

6.4 Conclusion

Dans ce chapitre, nous avons étudié le problème de la vérification d'un système matériel dans son environnement de fonctionnement. Nous avons proposé deux approches de modélisation sous forme de machines d'états finis :

- un reconnaisseur décrivant le langage de la propriété de l'environnement
- un générateur du langage accepté par un système matériel

Le problème de la vérification se ramène alors à définir une machine d'observation sur laquelle la vérification sera réalisée.

Partie III

Application du Langage Lustre de l'outil de vérification Lesar

Chapitre 7

Le langage Lustre

Le langage Lustre [CPHP 87, Pla 88, HCRP 91, Ray 91] a été conçu originellement pour la programmation des *systèmes réactifs*. Le rôle de ces systèmes est de réagir continûment à leur environnement [Ber 89]. La classe des systèmes réactifs englobe les systèmes suivants : les automates, les systèmes de traitement du signal, les protocoles de communication, etc .... Ces systèmes présentent deux caractéristiques principales : (1) ils sont intrinsèquement parallèles et (2) ils sont soumis à des contraintes temporelles sur le rythme d'acquisition des données et le temps de réponse entrée / sortie.

Lustre adopte l'approche *synchrone* pour décrire les systèmes réactifs à partir d'un modèle *flot de données* [Kah 74]. Dans le modèle flot de données, un système est constitué d'un réseau d'opérateurs agissant en parallèle au rythme de leurs entrées. L'hypothèse de synchronisme consiste à supposer que chaque opérateur de ce réseau répond instantanément à ses entrées.

Un programme Lustre est constitué d'un système d'équations opérant sur des flots. Un flot est représenté par une suite infinie de valeurs et une horloge associée représentant la suite d'instants où ces valeurs apparaissent. Tout programme Lustre a un comportement cyclique définissant son horloge de base. Toutes les autres horloges sont définies à partir de cette horloge de base. Dans le langage Lustre, les types prédéfinis sont les réels, les entiers et les booléens.

Tout programme Lustre est une hiérarchie de noeuds. Un noeud est un programme qui reçoit des flots d'entrées et calcule à l'aide d'un système d'équations les flots de sortie. Ainsi, un

programme complexe défini par un ensemble de noeuds peut être considéré comme un opérateur complexe obtenu par interconnexion d'autres opérateurs plus simples.

Plus récemment, le langage Lustre a été utilisé pour la description du matériel en vue de la vérification [Rat 92 , TP 90, TB 92], du test [Mah 90] et de la synthèse [RH 91a, RH 91b]. En effet, un système matériel peut être vu comme un système réactif. A cet effet, plusieurs extensions ont été ajoutées pour simplifier la description du matériel [Roc 92], tout en conservant la simplicité du langage et sa sémantique précise.

Dans ce chapitre, le noyau du langage Lustre est présenté en sections 1, 2 et 3. Nous nous restreignons dans la présentation du langage Lustre et par la suite au cas où tout les flots sont des booléens. Les extensions ajoutées au langage sont décrites dans les sections 4 et 5. Une description plus détaillée du langage se trouve dans [HCRP 91, Ray 91, Roc 92].

7.1 Horloges et flots

Dans le langage Lustre, toute variable ou expression dénote un flot; c'est-à-dire un couple formé d'une suite infinie de valeurs et d'une suite de valeurs booléennes représentant une suite d'instants. Par définition, un flot possède la n-ième valeur de sa suite de valeurs au n-ième instant de son horloge. Tout programme Lustre a un comportement cyclique, qui définit son *horloge de base*. Toutes les autres horloges sont définies à partir de celle-ci. Tout flot booléen peut être utilisé pour définir une horloge, qui est la suite des instants où sa valeur est vraie. L'exemple de la table 7.1 illustre les échelles de temps définies par un flot C sur l'horloge de base, et par un flot C' sur l'horloge définie par C.

L'horloge de base est considérée comme l'échelle de temps minimale discernable par le programme. Le temps physique sera généralement perçu comme une entrée du programme : par exemple, un flot booléen dont la valeur est vraie signale l'occurrence de l'événement "seconde". Ainsi, tout flot booléen peut être considéré comme définissant une échelle de temps.

Horloge de base	:	(0,	1,	2,	3,	4,	5,	...)
C	:	(true,	false,	true,	true,	false,	false,	...)
temps sur C	:	(0,		1,	2,			...)
C'	:	(true,		false,	true,			...)
temps sur C'	:	(0,			1,			...)

Table 7.1. Flots et horloges

7.2 Variables, équations et expressions

Les variables qui ne sont pas des entrées du programme sont définies au moyen d'équations. Une équation "X = E" où E est une expression, définit une synonymie complète entre la variable X et l'expression E : si E = (e_0, e_1, ···) alors la variable X est définie comme la suite ($x_0 = e_0$, $x_1 = e_1$, ···). Ceci nous fournit l'un des grands principes du langage, *le principe de substitution* : une telle équation permet de substituer X à E, et inversement, partout dans le programme. La notion d'affectation n'existe donc pas dans le langage Lustre.

Un autre grand principe de Lustre est *le principe de définition* : une variable est complètement définie par sa déclaration et l'équation où elle apparaît en membre de gauche. Les équations s'entendent donc *au sens mathématique du terme* : leur ordre est indifférent et l'introduction de variables intermédiaires pour nommer des expressions est sans aucune conséquence.

Les expressions sont construites avec des variables, des constantes et des opérateurs. Les constantes sont définies comme des suites infinies de même valeur sur l'horloge de base ; par exemple, la constante booléenne "true" dénote la suite : (true, true, ···). Dans ce qui suit, nous présenterons les opérateurs sur les données et les opérateurs opérant spécifiquement sur les suites du langage lustre.

7.2.1 Opérateurs sur les données

Les opérateurs usuels sont disponibles (booléens et conditionnels). Tous ces opérateurs, qualifiés d'opérateurs sur les données ne peuvent être appliqués qu'à des opérandes de même horloge, sur les valeurs desquels ils opèrent point par point. Par exemple, si X et Y sont deux flots sur l'horloge de base, de suites de valeurs respectives (x_0, x_1, ···) et (y_0, y_1, ···), alors l'expression "X and Y" représente la suite dont le n-ième terme est la multiplication logique des n-ièmes valeurs de X et de Y :

$$\text{"X and Y"} = (x_0 \text{ and } y_0, \cdots, x_n \text{ and } y_n, \cdots)$$

7.2.2 Opérateurs sur les suites

En plus des opérateurs sur les données, Lustre possède quatre opérateurs *temporels* opérant explicitement sur des flots. Ces opérateurs sont : l'opérateur de retard "pre", l'opérateur d'initialisation "→", l'opérateur d'échantillonnage "when" et l'opérateur de projection "current".

- L'opérateur "pre" (précédent) sert à mémoriser la valeur d'une expression à l'instant précédent de son horloge :

  si $(x_0, x_1, x_2, \cdots)$ est la suite de valeurs de l'expression X, alors "pre X" est une expression de même horloge que X et dont la suite de valeurs est : (nil, x_0, x_1, ···) où nil dénote une valeur indéfinie correspondant à l'absence d'initialisation.

- L'opérateur d'initialisation "→" permet d'initialiser un flot en définissant sa valeur initiale :

  si $X = (x_0, x_1, x_2, \cdots)$ est la suite de valeurs de l'expression X, et $Y = (y_0, y_1, y_2, \cdots)$ est la suite de valeurs de l'expression Y ayant la même horloge que X, alors "X → Y" est une expression de même horloge que X et Y dont la suite de valeurs est : $(x_0, y_1, y_2, \cdots)$.

- L'opérateur d'échantillonnage "when" sert à filtrer une expression sur une horloge plus lente. Soit E une expression et C une expression booléenne sur la même horloge que E. "E when C" est une expression sur l'horloge définie par C dont les valeurs, définies uniquement quand C est vraie, sont les valeurs de E à ces instants (table 7.2). L'expression "E when C" n'a pas la même référence temporelle que E et C.

- L'opérateur "current" sert à projeter une expression sur une horloge plus rapide. Si une expression E a pour horloge C, alors "current E" a pour horloge la même horloge que C et sa valeur à chaque cycle de cette horloge est la valeur prise par E au dernier cycle où C était vraie (table 7.2).

temps de base	:	(0,	1,	2,	3,	4,	5,	6,	···)
E	:	(e_0,	e_1,	e_2,	e_3,	e_4,	e_5,	e_6,	···)
C	:	(false,	true,	true,	false,	true,	false,	false,	···)
temps sur C	:	(	0,	1,		2,			···)
Y = E when C	:	(	e_1,	e_2,		e_4,			···)
current Y	:	(nil,	e_1,	e_2,	e_2,	e_4,	e_4,	e_4,	···)

Table 7.2. Opérateur d'échantillonnage et de projection

7.3 Les assertions

Le langage Lustre offre la possibilité d'utiliser des assertions. Elles spécifient des propriétés connues de l'environnement et jouent un rôle important dans la vérification des programmes Lustre. Une assertion est de la forme "assert E" et spécifie que l'expression booléenne E est toujours vraie. Les assertions peuvent être statiques ou dynamiques. Une assertion statique est une relation satisfaite à chaque instant, entre les valeurs présentes d'un ensemble de variables. Par exemple, l'assertion statique qui spécifie que deux événements d'entrée, représentés par les flots booléens X et Y, ne surviennent pas en même temps, s'écrira "assert not (X and Y)". Une assertion dynamique est une relation satisfaite à chaque instant, entre les valeurs présentes et passées d'un ensemble de variables. Une telle assertion peut être vue comme un reconnaisseur de langage. Par exemple, l'assertion qui spécifie que les deux variables X et Y ne changent pas de valeurs en même temps, s'écrira "assert (true → (X = pre X) or (Y = pre Y)". On notera l'utilisation de l'opérateur d'initialisation "→" , pour éviter l'apparition de la valeur nil au premier instant.

7.4 Structuration des programmes

La représentation en termes de réseaux des systèmes d'équations Lustre suggère naturellement une notion de sous-programme. Dans le langage Lustre, on appelle un tel sous-programme un noeud. Une déclaration de noeud consiste en une spécification d'interface (donnant les paramètres d'entrée et de sortie) et un système d'équations et d'assertions définissant les sorties et éventuellement les variables locales en fonctions des entrées. Par exemple, la déclaration suivante définit une cellule d'un demi-additionneur de deux bits a et b :

```
node H_ADDER (a, b : bool) returns (S, C_o: bool);
let
  S   =  a xor b;
  C_o =  a and b;
tel.
```

Le corps du noeud H_ADDER est réduit à deux équations définissant les deux paramètres de sortie S et C_o en fonction des paramètres d'entrée a et b. Le corps d'un noeud peut contenir des déclarations de variables locales.

Un noeud déclaré peut être instancié de manière fonctionnelle dans une expression Lustre. Par exemple, le noeud H_ADDER est utilisé pour définir un additionneur complet de la façon suivante :

```
node F_ADDER (a, b, Ci: bool) returns (S, Co: bool);
var S0, R0, R1 : bool;
let
   (S0, R0) = H_ADDER (a, b);
   (S, R1)  = H_ADDER (Ci, S0);
   Co       = R0 or R1;
tel.
```

En ce qui concerne les horloges, l'exécution d'un noeud est déterminée par l'horloge de ses paramètres d'entrée. Par exemple, l'instantiation F_ADDER ((a, b) when C) ne prend les valeurs du couple (S, Co) que lorsque la valeur du flot d'entrée C est vraie.

Un noeud peut admettre des paramètres d'entrée sur des horloges différentes. Si l'horloge d'un paramètre n'est pas l'horloge de base du noeud, cette horloge doit être passée en paramètre. De plus, les horloges qui ne sont pas l'horloge de base sont indiquées dans la déclaration d'interface. Dans l'exemple suivant :

```
node N (seconde: bool; (X: bool) when seconde) returns ... ;
```

les paramètres du noeud N sont : d'une part le flot booléen "seconde", sur l'horloge de base du noeud, d'autre part le paramètre X, sur l'horloge définie par le flot booléen "seconde".

Un noeud peut aussi retourner des paramètres d'horloges différents de l'horloge de base, à la condition que les variables définissant ces horloges soient visibles de l'extérieur du noeud. Par exemple, le noeud N' suivant retourne le paramètre X sur l'horloge B :

```
node N' (a, b, c: bool) returns (X, B: bool);
var Z: bool;
let
   Z = a and b;
   B = not c;
   X = Z when B;
tel.
```

7.5 Tableaux et structures

Dans le but de simplifier la description des systèmes matériels ayant une structure régulière, un certain nombre d'opérateurs ont été introduits pour permettre de définir et de manipuler des tableaux et des structures [Roc 92]. Avec l'introduction des tableaux et structures, la notion d'assertion a été généralisée. Les opérateurs prédéfinis et les noeuds ont été étendus pour pouvoir être appliqués à tous les éléments d'un tableau . Nous nous restreindrons ici, comme dans le cas de la présentation du langage noyau, au cas de structures et tableaux dont les éléments sont des booléens ou tableaux de booléens.

7.5.1 Opérateurs de construction

Un tableau peut être construit de plusieurs façons :

- Etant donné *n* expressions booléennes : E_0, E_1, ..., E_{n-1}, l'expression [E_0, ..., E_{n-1}] définit un tableau de type *bool^n* dont le ième élément est E_{i-1}.

- Etant donné *n* expressions entières : e_0, e_1, ..., e_{n-1}, et E une expression booléenne, l'expression E^(e_0, e_1, ..., e_{n-1}) définit un tableau de *n* dimensions où la ième dimension contient e_i copies de E. Par exemple, E^(2, 3) définit un tableau de 2 dimensions : la première dimension est de taille 2 et la deuxième est de taille 3 (figure 7.1).

Les structures considérées sont des tableaux dont les éléments sont des tableaux de dimensions différentes ou des expressions booléennes. De telles structures sont construites de la façon suivante : soient e_0, e_1, ..., e_{n-1}, *n* expressions, où chaque expression est une expression booléenne ou un tableau construit par une des façons décrites ci-dessus, l'expression [l_0: e_0, l_1: e_1, ..., l_{n-1}: e_{n-1},] définit une structure dont le label l_i est le tableau ou l'expression booléenne e_i. Par exemple [a: [e_0, e_1], b: e_3], où e_0, e_1 et e_3 sont des expressions booléennes, définit une structure à deux éléments : le premier élément est un tableau de deux expressions booléennes et le deuxième élément est une expression booléenne.

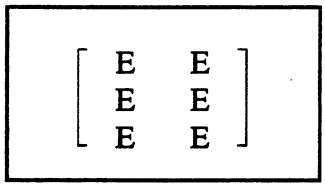


Figure 7.1. Un tableau à 2 dimensions

7.5.2 Opérateur de sélection

L'opérateur de sélection permet d'extraire, d'une structure ou d'un tableau, un élément ou une partie de ses éléments.

La sélection d'un élément d'une structure se fait en la suffixant par le label du champ sélectionné. Si S est une structure, l'expression "$S.l_i$" permet de sélectionner le champ de S ayant pour label l_i. Par exemple, si la structure S est égale à [a: E_0, b: [E_1, E_2]], l'expression "S.b" définit la structure [E_1, E_2].

La sélection d'un élément d'un tableau se fait en le repérant par sa position (les éléments d'un tableau sont indexés de 0 à n-1). Par exemple, si T est le tableau [E_0, E_1, E_2] où E_0, E_1 et E_2 sont des expressions booléennes, l'expression T[2] définit l'élément E_2 et l'expression T[1 .. 2] définit le tableau [E_1, E_2]. Notons que dans le cas de tableaux de tableaux, il est possible d'appliquer successivement plusieurs sélections pour extraire les éléments désirés.

7.5.3 Opérateur de concaténation

L'opérateur de concaténation permet de définir une structure ou un tableau par concaténation de deux autres tableaux ou structures.

Etant données une structure S_1 de *n* éléments et une structure S_2, de *m* éléments, l'expression "S_1 | S_2" définit une structure de $n + m$ éléments. Par exemple, si la structure S_1 est égale à [a: E_0, b: [E_1, E_2]] et la structure S_2 est égale à [c: [E_3, E_4], d: E_5], l'expression "S_1 | S_2" définit la structure [a: E_0, b: [E_1, E_2], c: [E_3, E_4], d: E_5].

Etant donnés un tableau T_1 de *n* éléments et un tableau T_2 de *m* éléments, l'expression "T_1 | T_2" définit un tableau de $n + m$ éléments. Par exemple, si le tableau T_1 est égal à [E_0, E_1] et le tableau T_2 est égal à [E_3, E_4], l'expression "T_1 | T_2" définit le tableau [E_0, E_1, E_3, E_4].

7.5.4 Assertions sur les structures et tableaux

La notion d'assertion a été généralisée aux tableaux et structures. Une assertion peut porter sur un tableau ou une structure. Si E est une structure ou un tableau de *n* éléments de type booléen alors assert E spécifie que chaque élément de E est toujours vrai.

7.5.5 Extension homomorphe des opérateurs

Les opérateurs Lustre peuvent admettre ainsi que les noeuds des structures ou des tableaux sur leurs opérandes. Par exemple, l'expression :

$$\text{if } [C_0, ..., C_{n-1}] \text{ then } [A_0, ..., A_{n-1}] \text{ else } [B_0, ..., B_{n-1}]$$

est équivalente à l'expression :

$$[\text{if } C_0 \text{ then } A_0 \text{ else } B_0, ..., \text{if } C_{n-1} \text{ then } A_{n-1} \text{ else } B_{n-1}].$$

Cette extension homomorphe des opérateurs permet de simplifier la description des parties régulières d'un programme. Par exemple, le noeud de la figure 7.2 utilise la description d'un additionneur complet de 2 bits, décrit en section 3, pour décrire un additionneur 8 bits.

```
node ADDER_8 (A, B: bool^8, Ci: bool)
     returns (S: bool^8, Co: bool);
var C: bool^8;
let
  (S, C) = F_ADDER (A, B, [Ci] | C[0 .. 6]);
  Co     = C[7];
tel.
```

Figure 7.2. Le programme Lustre décrivant un additionneur 8 bits

7.5.6 Paramètres statiques et récursivité

Dans le but d'écrire des bibliothèques générales avec des tailles de tableaux quelconques, les paramètres statiques ont été introduits. La déclaration du paramètre entier définissant la taille du tableau est précédé par le mot clé "static". Par exemple, la bibliothèque pourra contenir la description Lustre d'un additionneur n bits (ADDER_N, figure 7.3a). Le noeud paramétré ADDER_N est utilisé pour définir un additionneur 8 bits , donné en figure 7.3b.

Pour qu'un programme Lustre puisse contenir des appels récursifs l'opérateur "with" a été introduit. Par exemple, le programme de la figure 7.4 teste l'égalité entre deux tableaux d'éléments booléens. L'égalité sur les premières et secondes moitiés des deux tableaux est testée récursivement. La condition d'arrêt est déterminée par l'opérateur with, lorsque les deux tableaux à comparer n'ont qu'un élément.

```
node ADDER_N (static n:int; A, B: bool^n, Ci: bool)
      returns (S: bool^n, Co: bool);
var C: bool^n;
let
  (S, C) = F_ADDER (A, B, [Ci] | C[0..n-2]);
  Co     = C[n-1];
tel.
```

Figure 7.3a. Le programme Lustre décrivant un additionneur n bits

```
node ADDER_8 (A, B: bool^8; Ci: bool)
     returns (S: bool^8, Co: bool);
let
  (S, Co) = ADDER_N (8, A, B, Ci);
tel.
```

Figure 7.3b. Le programme Lustre décrivant un additionneur 8 bits.

```
node COMP (static n:int; A, B: bool^n) returns (Flag: bool);
let
  Flag = with (n = 1) then (A[0] = B[0])
            else COMP (n/2, A[0 ..(n/2)-1], B[0 ..(n/2)-1]) and
                 COMP (n+1/2, A[n/2 ..n-1], B[n/2 ..n-1]);
tel.
```

Figure 7.4. Le noeud Lustre récursif décrivant la comparaison entre 2 tableaux à n éléments

7.6 Conclusion

La conception du langage Lustre est fondée sur une approche synchrone à partir d'un modèle flot de données. Ainsi, tout programme Lustre est un réseau d'opérateurs agissant en parallèle au rythme de leurs entrées. Chaque opérateur de ce réseau répond instantanément à ses entrées. Les opérateurs du langage Lustre opèrent sur des flots, qui sont des suites infinies de valeurs et des horloges associées, qui représentent les suites d'instants où ces valeurs apparaissent. En plus des opérateurs classiques (arithmétiques, booléens, conditionnel), le langage Lustre possède quatre opérateurs temporels opérant spécifiquement sur des flots. Des extensions (structures et tableaux) ont été ajoutées pour simplifier la description des systèmes matériels.

Nous montrerons dans le chapitre 9 que les caractéristiques du langage Lustre lui permettent d'être utilisé en tant qu'outil de description des SMSS aux différents niveaux d'abstraction (de la spécification à la réalisation). Bien plus, nous montrerons dans le chapitre 8 que tout programme Lustre opérant sur des flots de booléens dénote une machine d'états finis. Ceci va nous permettre de définir, dans le chapitre 9, un cadre unifié pour la description et la vérification des SMSS.

Chapitre 8

Sémantique de Lustre en termes de machines d'états finis

Un langage pour la description du matériel en vue de la vérification de celui-ci doit avoir une sémantique formelle précise. Cette sémantique ne doit pas seulement répondre à la question "que fait le programme?", mais aussi à la question "quand le fait-il?". Nous allons montrer que ces deux questions reçoivent des réponses claires dans le langage Lustre.

Dans ce chapitre nous montrons que les opérateurs et les assertions du langage Lustre, opérant sur des flots booléens, dénotent des machines d'états finis. Cette sémantique formelle permet d'utiliser le langage Lustre comme une application de l'approche de vérification, présentée dans la deuxième partie de cette thèse.

8.1 Les opérateurs sur les données

Les opérateurs sur les données (and, or, non et conditionnel) opèrent point par point sur des suites infinies de booléens. Ces opérateurs dénotent une machine à un seul état caractérisée par le 5-uplet $(q_0, E, S, \delta_I, \lambda)$ où :

- q_0 est l'unique état de la machine ;
- E est l'ensemble des entrées : $\{0, 1\}$ pour la machine "non", $\{0, 1\}^2$ pour les machines "and" et "or", et $\{0, 1\}^3$ pour la machine "conditionnelle" ;

- S est l'unique sortie à valeur dans {0, 1} ;
- δ_I est la fonction identité de transition ;
- λ est la fonction logique : "and", "or", "not" ou conditionnelle.

L'exemple du graphe d'états de la machine "or" est donné en figure 8.1 .

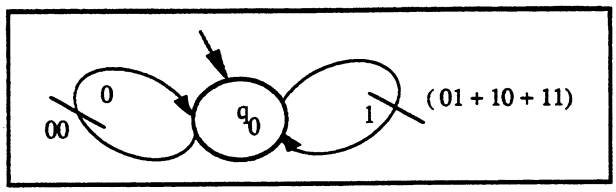


Figure 8.1. Graphe d'états de la machine "or"

8.2 Les opérateurs temporels

Les opérateurs temporels opèrent spécifiquement sur des flots. Rappelons que ces opérateurs sont : l'opérateur de retard "pre", l'opérateur d'initialisation "→", l'opérateur d'échantillonnage "when" et l'opérateur de projection "current".

8.2.1 L'opérateur "pre"

L'opérateur de mémorisation "pre" décale une suite dans le temps. Cet opérateur peut être vu comme une machine retournant l'élément précédent de la suite d'entrée, sauf à l'instant initial où cette machine retourne la valeur indéfinie "nil". La structure d'une telle machine est donnée en figure 8.2. La machine "pre" possède deux mémoires mem_1 et mem_2 : la mémoire mem_1 mémorise la suite d'entrée au rythme de son horloge et la mémoire mem_2 , initialement à la valeur booléenne 1, reçoit en entrée la constante booléenne 0. La mémoire mem_2 commande un multiplexeur (Mux) tel que la valeur initiale retournée par la machine est "nil".

L'opérateur "pre" dénote donc une machine d'états finis dont le graphe d'états est donné en figure 8.3. Cette machine est caractérisée par le 6-uplet $(Q, E, S, \delta_{pre}, \lambda_{pre}, q_0)$ où :

- Q est l'ensemble de trois états $\{q_0, q_1, q_2\}$ correspondant aux trois configurations possibles des deux mémoires de la machine : ($q_0 \equiv (mem_1 = \phi,\ mem_2 = 1)$, $q_1 \equiv (mem_1 = 0,\ mem_2 = 0)$, $q_2 \equiv (mem_1 = 1,\ mem_2 = 0)$) où ϕ correspond à la valeur indéterminée ou "don't care" ;

- E est l'entrée à valeur dans {0, 1} ;
- S est la sortie à valeur dans {0, 1} ;
- δ_{pre} est la fonction de transition définie de $\{0, 1\} \times \{q_0, q_1, q_2\}$ dans $\{q_1, q_2\}$;
- λ_{pre} est la fonction de sortie définie de $\{q_0, q_1, q_2\}$ dans {0, 1} ;
- q_0 est l'état initial.

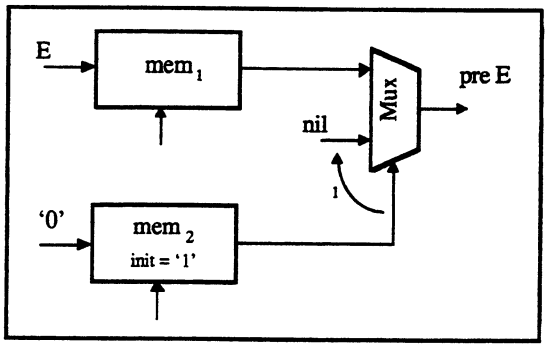


Figure 8.2. Structure de la machine "pre"

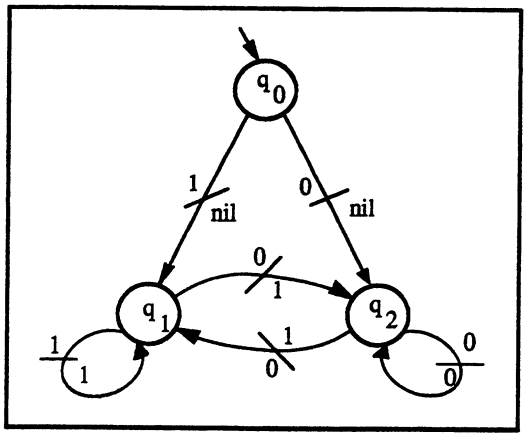


Figure 8.3. Graphe d'états de la machine "pre"

8.2.2 L'opérateur "→"

L'opérateur d'initialisation "→" reçoit en entrée deux suites et retourne la seconde suite, sauf au premier instant où il retourne le premier élément de la première suite. L'opérateur d'initialisation "→" peut être vu comme une machine dont la structure est donnée par la figure 8.4. La machine "→" possède une mémoire *mem*, initialement à la valeur booléenne 1, recevant sur son entrée la constante booléenne 0. Cette mémoire commande un multiplexeur tel que la valeur initiale retournée par la machine "→" est le premier élément de la première suite.

L'opérateur "→" dénote donc la machine d'états finis dont le graphe d'états est donné en figure 8.5. La machine "→" est définie par le 6-uplet $(Q, E, S, \delta_{\rightarrow}, \lambda_{\rightarrow}, q_0)$ où :

- Q est l'ensemble de deux états $\{q_0, q_1\}$ correspondant aux deux configurations possibles de la mémoire de la machine ($q_0 \equiv$ (mem = 1) et $q_1 \equiv$ (mem = 0)) ;
- E est l'ensemble d'entrée à valeur dans $\{0, 1\}^2$;
- S est la sortie à valeur dans {0, 1} ;

- $\delta_{\rightarrow}$ est la fonction de transition définie de $\{0, 1\}^2 \times \{q_0, q_1\}$ dans $\{q_1\}$;
- $\lambda_{\rightarrow}$ est la fonction de sortie définie de $\{q_0, q_1\}$ dans $\{0, 1\}$;
- q_0 est l'état initial.

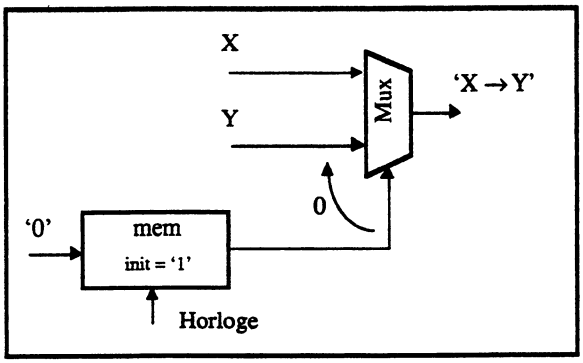


Figure 8.4. Structure de la machine "→"

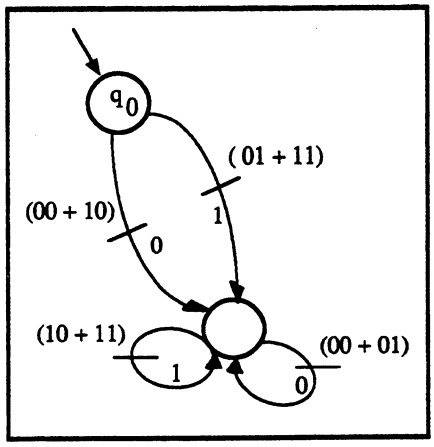


Figure 8.5. Graphe d'états de la machine "→"

8.2.3 L'opérateur "when"

L'opérateur d'échantillonnage "when" reçoit en entrée deux suites X et C et retourne la valeur de la suite X si la valeur de la suite C vaut 1, sinon il ne retourne aucune valeur (cette absence de valeur est noté ε).

L'opérateur "when" peut être vu comme une machine ayant un unique état dont le graphe d'états est donné en figure 8.6. Cette machine est définie par le 5-uplet $(q_0, E, S, \delta_{when}, \lambda_{when})$ où :

- q_0 est l'unique état de la machine;
- E est l'ensemble d'entrée à valeur dans $\{0, 1\}^2$;
- S est la sortie à valeur dans $\{0, 1, \varepsilon\}$;
- δ_{when} est la fonction de transition définie de $\{0, 1\}^2 \times \{q_0\}$ dans $\{q_0\}$;
- λ_{when} est la fonction de sortie définie de $\{0, 1\}^2 \times \{q_0\}$ dans $\{0, 1, \varepsilon\}$.

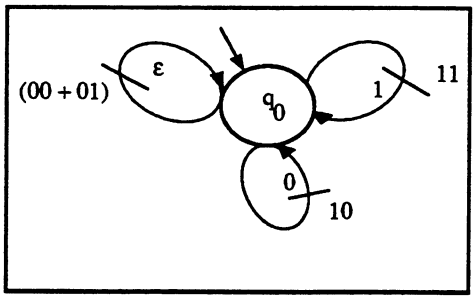


Figure 8.6. Graphe d'états de la machine "when"

8.2.4 L'opérateur "current"

L'opérateur de projection "current" reçoit en entrée deux suites Y et C et retourne la projection de la suite Y sur la suite C. L'opérateur current peut être vu comme une machine retournant à chaque instant la valeur de la suite Y si la valeur de la suite C est 1, sinon la machine retourne la valeur précédente.

La machine "current" possède deux mémoires mem_1 et mem_2 : la mémoire mem_1 mémorise la sortie et la mémoire mem_2 , initialement à la valeur booléenne 1, reçoit en entrée la constante booléenne 0. La mémoire mem_2 commande un multiplexeur tel que la valeur initiale retournée par la machine "current" est nil, si la valeur initiale de la suite C est 0. La structure de la machine "current" est donnée en figure 8.7.

L'opérateur "current" dénote la machine d'états finis dont le graphe d'états est donné en figure 8.8. Cette machine est définie par le 6-uplet $(Q, E, S, \delta_{current}, \lambda_{current}, q_0)$ où :

- Q est l'ensemble de trois états $\{q_0, q_1, q_2\}$ correspondant aux trois configurations suivantes des deux mémoires de la machine : $(q_0 \equiv (mem_1 = \phi,\ mem_1 = 1)$, $q_1 \equiv (mem_1 = 0, mem_2 = 0)$ et $q_2 \equiv (mem_1 = 1, mem_2 = 0))$;
- E est l'ensemble d'entrée à valeur dans $\{0, 1\}^2$;
- S est la sortie à valeur dans {0, 1, nil} ;
- $\delta_{current}$ est la fonction de transition définie de $\{0, 1\}^2 \times \{q_0, q_1, q_2\}$ dans $\{q_1, q_2\}$;
- $\lambda_{current}$ est la fonction de sortie définie de $\{0, 1\}^2 \times \{q_0, q_1, q_2\}$ dans {0, 1, nil} ;
- q_0 est l'état initial.

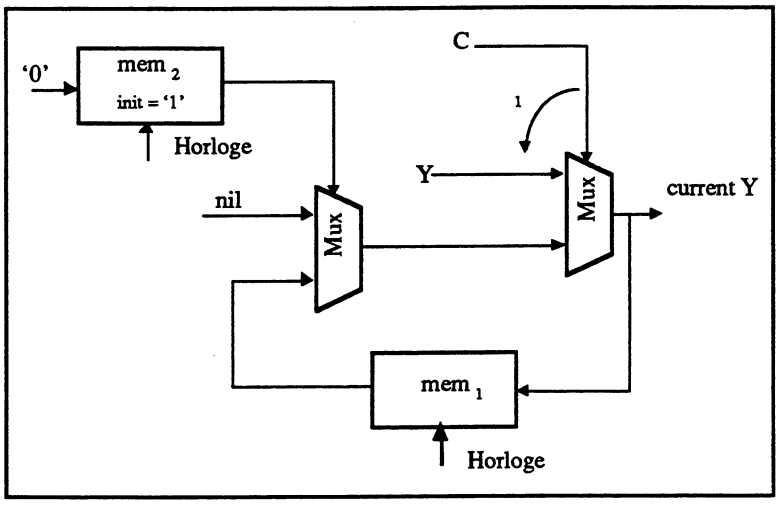


Figure 8.7. Structure de la machine "current"

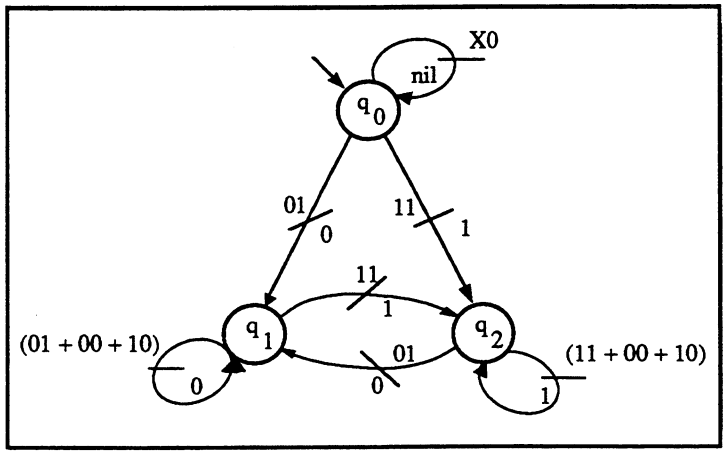


Figure 8.8. Graphe d'états de la machine "current"

8.3 Un programme Lustre

Un programme Lustre peut être vu comme un réseau d'opérateurs connectés par des fils. Lorsque ces opérateurs opèrent sur des flots constitués de suites de valeurs booléennes et ayant la même horloge, un programme Lustre dénote une machine d'états finis M. Cette machine est issue du produit synchrone des machines M_i (chapitre 1, §1.2.1) associées aux différents opérateurs du programme :

$M = M_1 \times M_2 \times ... \times M_n$, où n est le nombre des opérateurs constituant le programme.

Un programme Lustre P, dont les flots sont booléens, peut être vu comme un nouvel opérateur dénotant une machine d'états finis $M_P = (Q_P, E_P, S_P, \delta_P, \lambda_P, q_0)$. Considérons l'exemple du programme suivant :

```
node EXEMPLE (A, B: bool) returns (S: bool);
var X,Y: bool;
let
Y = true -> B;
X = current (A when Y);
S = true -> pre (X);
tel.
```

Ce programme reçoit en entrée les flots booléens A et B, calcule les flots locaux X et Y, et retourne le flot booléen S. Ce programme peut être vu comme un nouvel opérateur issu de l'interconnexion des trois opérateurs: "true → ()", "current () when ()" et "true → pre ()". Ces trois opérateurs opèrent sur des flots booléens définis sur l'horloge de base. Nous avons vu précédemment que chacun de ces trois opérateurs dénote une machine d'états finis. Par conséquent, le noeud "EXEMPLE" dénote la machine issue de l'interconnexion de ces trois machines. La structure de cette machine et son graphe d'états sont représentés par les figures 8.9 et 8.10. Cette machine produit a cinq mémoires, par conséquent son espace d'états contiendra au plus 2^5 états. Le graphe d'états de cette machine montre que seul trois états sont accessibles. Ces trois états sont définis par les configurations suivantes des cinq mémoires de la machine :

$$q_0 \equiv (mem_1, mem_2, mem_3, mem_4, mem_5) = (1, 1, 1, \phi, \phi) ;$$

$$q_1 \equiv (mem_1, mem_2, mem_3, mem_4, mem_5) = (0, 0, 0, 0, 0) ;$$

$$q_2 \equiv (mem_1, mem_2, mem_3, mem_4, mem_5) = (0, 0, 0, 1, 1) ;$$

où ϕ correspond à la valeur "don't care"

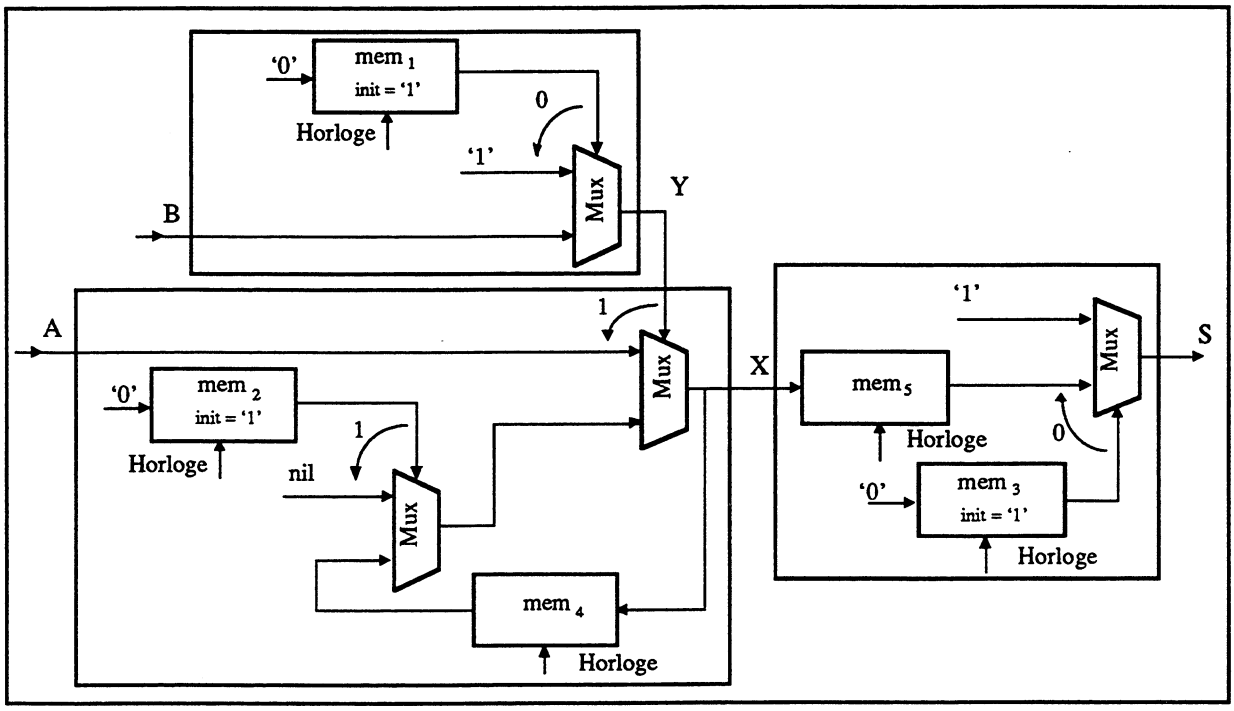


Figure 8.9. Structure de la machine "EXEMPLE"

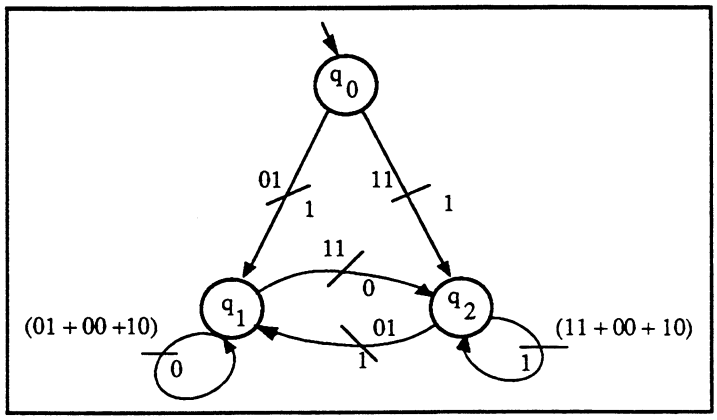


Figure 8.10. Graphe d'états de la machine "EXEMPLE"

8.4 Les assertions

Soit X une expression booléenne construite à l'aide d'opérateurs de données et / ou d'opérateurs temporels. L'expression X dénote une machine d'états finis M_X, définie par le 6-uplet $(Q_X, E, \{0, 1\}, \delta_X, \lambda_X, q_0)$. Une assertion sur l'expression X est de la forme "assert X". Cette assertion spécifie que l'expression booléenne X est toujours vraie. En d'autres termes, "assert E" dénote la machine d'états finis M'_X caractérisée par le 6-uplet

$(Q', E, \{1, 0\}, \delta'_X, \lambda'_X, q_0)$ où :

- $Q' = Q_{ass} \cup q_p$, où $Q_{ass} \subseteq Q_X$ et q_p est un état puits ;

- $$\delta'_X(e, q) = \begin{cases} \delta_X(e, q) & si \quad \lambda_X(e, q) = 1 \\ q_p & si \quad \lambda_X(e, q) = 0 ; \end{cases}$$

- $$\lambda'_X(e, q) = \begin{cases} \lambda_X(e, q) & \forall (e\ q) \in E \times Q_{ass} \\ 0 & si \quad q = q_p \text{ ou } \delta'_X(e, q) = q_p \end{cases}$$

Nous distingons le cas particulier donné par la définition suivante :

Définition. Nous appelons une assertion non causale, l'assertion qui dénote la machine d'états finis $M'_X = (Q', E', \{0, 1\}, \delta'_X, \lambda'_X, q_0)$ ayant un état de transition q_t dans son espace d'états qui mène uniquement vers l'état puits :

$$\exists\ q_t \in Q' \mid \forall\ e \in E, \delta'_X(q_t, e) = q_p$$

Exemple. Considérons l'exemple du programme Lustre donné en section 8.3. Ce programme reçoit les flots booléens A et B et retourne le flot booléen S. L'expression S dénote une machine d'états finis dont le graphe d'états est donné en figure 8.10. Soit l'assertion "assert S" spécifiant que l'expression S est toujours vraie. Une telle assertion dénote une machine d'états finis M' dont le graphe d'états est donné en figure 8.11. La machine M' est déduite de la machine associée à l'expression S, en remplaçant toutes les transitions générant la valeur booléenne 0 par des transitions menant vers l'états puit. Cette assertion est non causale. En effet, l'état q_1 de l'espace d'états de la machine M' est un état de transition menant uniquement vers l'état puits.

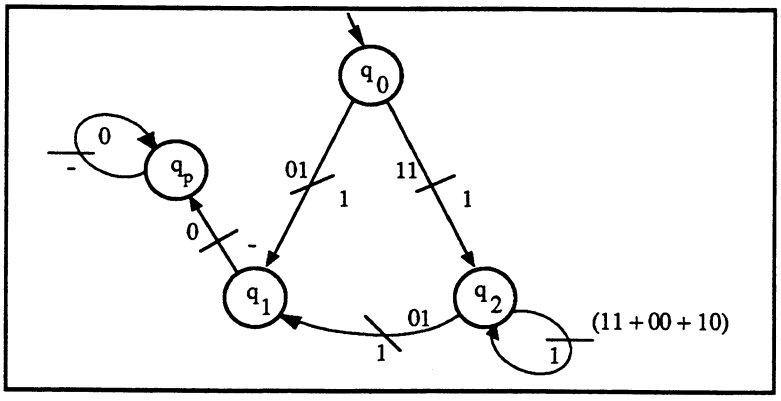


Figure 8.11. Graphe d'états de la machine associée à l'assertion "assert S"

Par la suite, nous considérons que tout programme contenant des assertions dénote une machine d'états finis caractérisée par la fonction d'assertion α. Cette fonction α est la fonction de sortie de la machine d'états finis associée à l'expression "assertée".

8.5 Conclusion

Nous avons montré dans ce chapitre que les opérateurs et les assertions du langage Lustre, opérant sur des flots booléens, ont une sémantique formelle simple : chaque opérateur dénote une machine d'états finis. Un programme Lustre peut être vu comme un réseau d'opérateurs connectés par des fils. Lorsque ces opérateurs opèrent sur des flots booléens, un tel programme dénote une machine d'états finis issue de l'interconnexion des diverses machines associées aux opérateurs du réseau.

Chapitre 9

Un cadre unifié pour la description et la vérification fonctionnelle des SMSS

Il a été montré dans [TP 90] que le langage Lustre peut être utilisé pour la description des systèmes matériels aux différents niveaux de conception (de la spécification à la réalisation). L'utilisation du même langage pour la description des systèmes matériels aux différents niveaux d'abstraction présente un double avantage : (1) l'apprentissage du même langage pour le concepteur et (2) la possibilité de faire des descriptions multi-niveaux d'un système matériel. Notons que cette notion de langage unique a été proposé par ailleurs [Boc 82, Pai 86, Lev 91]. De plus, il a été montré [BFH 90] que le langage Lustre a exactement le pouvoir d'expression pour spécifier toute propriété de sûreté de tout programme Lustre. Par conséquent, les propriétés de sûreté relatives au comportement des systèmes matériels peuvent aussi être décrites dans le langage Lustre.

Nous avons montré dans le chapitre 8 que tout programme Lustre, opérant sur des flots booléens, dénote une machine d'états finis. Cette sémantique formelle permet d'utiliser le langage Lustre comme une application de l'approche unifiée de vérification, présentée dans la deuxième partie de cette thèse, pour résoudre les problèmes de vérification de propriétés et de conformité. Rappelons que dans cette approche nous définissons une machine de vérification M_V ayant une unique sortie notée *Flag*. Le problème de la vérification se ramène à vérifier que la machine M_V génère sur son unique sortie *Flag* le langage 1^*.

Dans ce chapitre, nous présentons d'abord le langage Lustre en tant qu'outil de description des SMSS. Ensuite, nous montrons que ce même langage peut être utilisé pour décrire un programme de vérification qui dénote la machine de vérification sur laquelle la vérification de propriétés ou de conformité sera réalisée.

9.1 Description des SMSS

Le langage Lustre peut être utilisé pour la description des systèmes matériels aux différents niveaux de conception (de la spécification à la réalisation). Dans cette section, nous présentons le langage Lustre en tant qu'outil de description des SMSS aux niveaux d'abstraction comportemental et structurel. Cette présentation est illustrée par des exemples.

9.1.1 Description de la spécification d'un SMSS en Lustre

Au plus haut niveau d'abstraction, un système matériel peut être considéré comme un système réactif retournant à chaque instant des valeurs de sortie dépendant des valeurs présentes et précédentes de ses entrées. En d'autres termes, toute sortie s du système à l'instant présent est considérée comme une fonction $\mathbb{F}$ des entrées à l'instant présent et aux instants précédents :

$$\forall\, t\,,\ s\,(t) = \mathbb{F}\ (e(t), e(t\text{-}1), ..., e(0))$$

Cette description au niveau comportemental est appelée spécification du système matériel. Notons que cette description n'implique pas une architecture spécifique du système.

Le langage Lustre permet de décrire les SMSS au niveau comportemental. Ceci est possible grâce à l'approche synchrone à partir d'un modèle flot de données adopté par le langage Lustre. En effet, une description flot de données permet une prise en compte de la dimension temporelle : les variables manipulées s'interprètent comme des fonctions du temps. L'hypothèse de synchronisme consiste à supposer que les primitives du langage sont "idéales" : toute primitive réagit instantanément aux événements d'entrée. Chaque événement est daté par rapport aux événements externes assurant ainsi un comportement déterministe du point de vue temporel.

Les opérateurs temporels "pre" et "→" du langage Lustre facilitent l'expression des fonctions temporelles décrivant le comportement d'un système matériel : l'opérateur de retard "pre" permet de mémoriser les valeurs précédentes des entrées et l'opérateur d'initialisation "→" permet de donner des valeurs initiales aux flots retardés.

9.1.2 Exemple de spécification d'un SMSS en Lustre

L'exemple traité ici est un petit processeur de signal appelé MinMax [Leu 89]. Ce processeur reçoit en entrée des données "In" ainsi que trois signaux de contrôle : Reset, Enable et Clear. Il fournit en sortie des données "Out" dépendant de l'entrée et des signaux de contrôle comme le montre la table 9.1.

La définition du noeud MINMAX_SPEC spécifiant le processeur MinMax est donnée en figure 9.1. Cette description de la spécification définit les fonctions Max, Min et Last_in et la fonction de sortie Out sur *n* bits, selon la valeur des signaux de contrôle : Clear, Reset, Enable. La notion de la décomposition hiérarchique du langage Lustre permet de faciliter cette description. Ainsi, le noeud MinMax utilise les noeuds ADD_n et COM_n définissant respectivement un additionneur *n* bits donné au chapitre 7 (section 7.3) et un comparateur de deux tableaux *n* bits de type booléen. Notons que pour la vérification ou la simulation du noeud spécifiant le processeur MinMax, le programme principal ne doit pas avoir de paramètre statique, mais une valeur de ce paramètre.

Clear	Reset	Enable	Out
1	X	X	0
0	0	1	(Min + Max) / 2
0	1	1	In
0	X	0	Last_in

Table 9.1. Table illustrant le fonctionnement du processeur MinMax

où :

- Max est le maximum atteint par In depuis que (Clear, Reset, Enable) = (0, 0, 1) ;
- Min est le minimum atteint par In depuis le même instant ;
- Last_in est égal à In tant que (Clear, Reset, Enable) = (0, X, 1).

```
node MINMAX_SPEC (static n:int; In: bool^n; Clear, Reset, Enable: bool)
                 returns (Out: bool^n);
var Last, Sum, Max, Min, div2: bool^n; add: bool^n+1; Co:bool;
let
  Max       = if (Clear^n or Reset^n) then false^n else
                 if COM_n (n, In, pre(Max))^n then In else pre(Max);

  Min       = if (Clear^n or Reset^n) then true^n else
                 if COM_n (n, pre(Min), In)^n then In else pre(Min);

  Last      = if Clear^n then false^n else
                 if (not Enable)^n then (false^n -> pre(Last)) else In;

  (Sum, Co) = ADD_N (n, Max, Min, false);

  add = Sum | Co;

  div2 = add[1 .. n];

  Out       = if Clear^n then false^n else
                 if (not Enable)^n then Last else
                       if Reset^n then In else div2;
let.

node COM_N (static n:int; A, B: bool^n) returns (So: bool);
var S, A_p, B_p: bool^n;
let
  A_p = A[n-1 .. 0];
  B_p = B[n-1 .. 0];
  S   = SUP (A_p, B_p, false | S[0 .. n-2]);
  So  = S[n-1];
tel.

node SUP (a, b, si: bool) returns (so: bool);
let
  so = (not(a) and si) or (a and not(b) and not(si)) or (a and b and si);
tel.
```

Figure 9.1. La spécification fonctionnelle du processeur MinMax en Lustre

9.1.3 Description de la réalisation d'un SMSS en Lustre

La description d'une réalisation d'un SMSS est une description architecturale (ou structurelle) dans laquelle le SMSS est représenté comme un assemblage d'éléments matériels primitifs. Les éléments primitifs d'un SMSS sont composés d'éléments combinatoires et d'éléments de mémorisation. Un élément de mémorisation est un composant retournant sa valeur d'entrée sur un top d'horloge et maintient sa valeur de sortie jusqu'au prochain top d'horloge. La bascule D est un exemple d'élément de mémorisation.

Grâce à son parallélisme implicite et ses caractéristiques temporelles, le langage Lustre permet de décrire les systèmes matériels synchrones au niveau architectural. En effet, un programme Lustre est un réseau d'opérateurs agissant en parallèle et sa notion du temps est proche de celle du matériel : tout flot booléen peut être utilisé pour définir une horloge.

La notion de décomposition hiérarchique du langage Lustre permet la définition d'une bibliothèque de systèmes de base analogue à la notion de bibliothèque de circuits dans les systèmes de CAO. Bien plus, cette bibliothèque de systèmes peut être générale dans le cas où ces systèmes ont une structure régulière. Une telle définition est possible grâce aux opérateurs de définition et de manipulation des tableaux du langage Lustre.

La temporisation et la synchronisation des différents blocs du SMSS, évoluant sur une ou plusieurs horloges biphasées, sont réalisées à l'aide de l'opérateur d'échantillonnage "when" et de l'opérateur de projection "current".

9.1.4 Exemple de description d'une réalisation d'un SMSS en Lustre

L'exemple examiné ici est une réalisation possible d'un arbitre de bus [TB 92]. Cet arbitre est responsable d'allouer un bus à trois unités de traitement U_i (i = 0, 1, 2). Chaque unité envoie une requête req_i (i = 0, 1, 2) d'allocation à l'arbitre et, dès que le bus n'est pas utilisé, celui ci l'affecte à l'une des unités en attente, en retournant la valeur booléenne 1 sur la sortie gr_i (i = 0, 1, 2). Dans le cas où plusieurs unités sont en attente, une priorité d'accès au bus permet de choisir l'unité. Cette priorité est fixe, l'unité 0 a le plus haut niveau de priorité, et l'unité 2 a le plus bas niveau.

La réalisation, que nous donnons de cet arbitre de bus a une architecture pipeline à deux étages commandés par deux horloges synchrones bipahasées: Ck_1, Ck_2. Le schéma de cette

réalisation est donné par la figure 9.2. L'élément de mémorisation de base de cette réalisation est la bascule D.

La description Lustre de la bascule D est donnée par le noeud FLIP_FLOP (figure 9.3). La bascule D échantillonne la valeur d'entrée sur l'horloge Ck : "D when Ck", et la retient jusqu'au prochain top d'horloge. Initialement la bascule D est à la valeur booléenne Init. Le temps de traversée de la bascule D n'est pas instantané, l'opérateur "pre" nous permet de prendre en compte ce temps. Notons que l'initialisation à la valeur booléenne 1 de la variable Ck est nécessaire pour éviter l'apparition de la valeur indéfinie "nil".

Le noeud Lustre ARBITER_IMP décrivant la réalisation est donné en figure 9.4. Ce noeud décrit exactement les différents blocs du circuit et leurs connexions : les bascules D, les portes logiques AND et NOR et le circuit logique FUN. Notons que le noeud ARBITER_IMP utilise le noeud général REGISTER, qui lui même utilise le noeud FLIP_FLOP décrivant la bascule D.

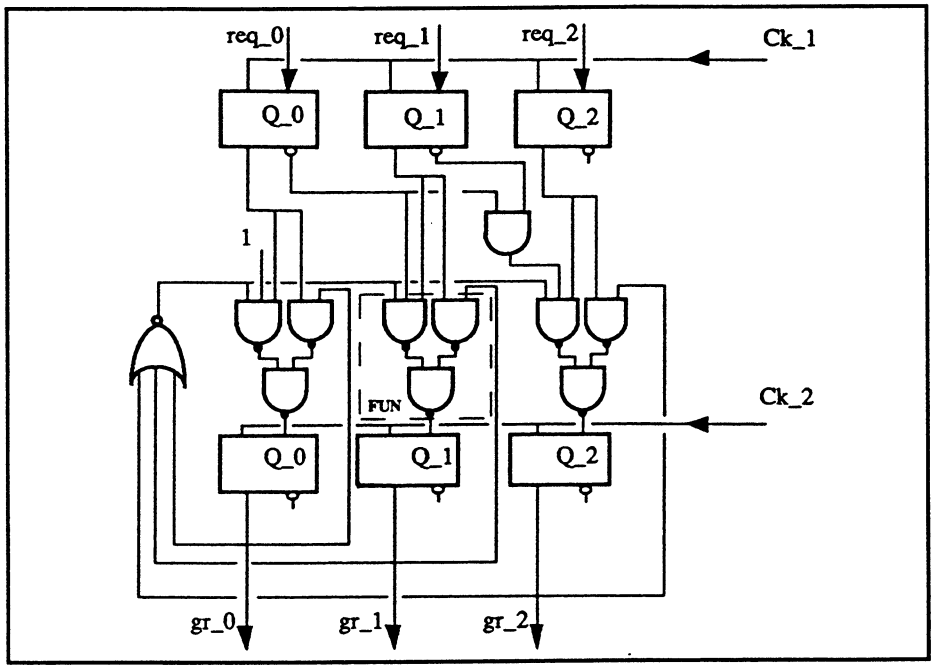


Figure 9.2. Une réalisation pipeline de l'arbitre de bus

```
node FLIP_FLOP (D, Ck, Init: bool) returns (Q, Q_: bool);
let
Q    = Init -> pre (current (D when (true -> Ck)));
Q_   = not (Q);
tel.
```

Figure 9.3. La bascule D en Lustre

```
node ARBITER_IMP (req: bool^3, Ck_1, Ck_2: bool) returns (gr: bool^3);
var A, A_, dr, gr_: bool^n; dr_0, dr_1, dr_2, D, P_1, P_2: bool;
let
-- Mémorisation des demandes de bus
   (A, A_) = REGISTER (3, false, req, Ck_1);

 -- Calcul des priorités
  P_1 = A_ [0];
  P_2 = A_ [1] and A_ [0];

-- Bus libre
  D = not (gr [0] or gr [1] or gr [2]);

-- Allocation du bus
  dr_0 = FUN (D, true, A_[0], A_[0], gr_[0]);
  dr_1 = FUN (D, P_1, A_[1], A_[1], gr_[1]);
  dr_2 = FUN (D, P_[2], A_[2], A_[2], gr_[2]);
  dr   = dr_0 | dr_1 | dr;

-- Envoi des acquittements vers les unités de calcul
  (gr, gr_)  = REGISTER (3, false^3, dr, Ck_2^3);
tel.

node FUN (A, B, C, D, E: bool) returns (s: bool);
var j, k: bool;
let
  j  = not (A and B and C);
  k  = not (D and E);
  s  = not (j and k);
tel.

node REGISTER (const n: int; init, D, Ck: bool^n)
returns (Q, Q_: bool^n);
let
  (Q, Q_)  = FLIP_FLOP (D, init, Ck);
tel.
```

Figure 9.4. La réalisation de l'arbitre de bus en Lustre

9.2. Vérification des propriétés de sûreté

La vérification de propriétés consiste à vérifier une propriété temporelle sur une description d'un SMSS à niveau d'abstraction donné. Nous avons vu en section 9.1 que le langage Lustre permet la description d'un SMSS aux différents niveaux d'abstraction et l'expression de toute propriété de sûreté de tout programme Lustre.

Dans le chapitre 4, nous avons décrit une méthode de vérification de propriétés de sûreté, définissant un langage régulier, sur une description d'un système matériel. Cette méthode de vérification consiste à définir une machine de vérification issue de l'interconnexion de la machine associée à la description du système matériel et de la machine décrivant le langage de la propriété de sûreté.

Dans cette section, nous montrons que le langage Lustre permet de décrire le programme de vérification dénotant la machine de vérification sur laquelle la vérification sera réalisée. Ce programme dénote la machine de vérification décrite en chapitre 4. Nous étudions d'abord la méthode de spécification des propriétés de séquences finies, puis celle des propriétés de sûreté dans le langage Lustre.

9.2.1 Spécification des propriétés de séquences finies en Lustre

Nous avons vu au chapitre 3 que les propriétés de sûreté relatives aux comportements des SMSS dont le langage associé est régulier, s'expriment à l'aide des propriétés de séquences finies d'intervalles d'observation, de contexte et de réaction.

Comme les expressions du langage Lustre sont des suites de valeurs avec une horloge associée définissant les instants où ces valeurs apparaissent, les expressions booléennes construites à l'aide de variables, de constantes et d'opérateurs, peuvent être vues comme des propriétés de séquences ou suites finies. Une expression booléenne E exprime la propriété P telle que la suite de valeurs ($e_0\ e_1 \cdots e_n$) associée à l'expression E satisfait la propriété P, si et seulement si, l'expression booléenne E vaut la valeur booléenne 1 à l'instant t_n de son horloge :

$$\forall\ E = (e_0\ e_1 \cdots e_n),\ E \models P \quad \text{ssi} \quad e_n = 1$$

Par exemple, l'expression Lustre "E = true → (a and not pre (a))" exprime l'événement défini $edge_a$: transition montante de la suite *a*. Une transition montante est présente (propriété

d'événement défini satisfaite), si et seulement si, la suite *a* vaut la valeur 1 à l'instant présent et valait la valeur 0 à l'instant précédent de son horloge :

$$\forall\, a = (a_0\, a_1 \cdots a_n),\ E\,(a) \models edge_a \quad ssi \quad E = a_{n-1} \bullet a_n = 1$$

Les propriétés de séquences finies d'intervalle, de contexte et de réaction seront exprimées de cette façon. Les noeuds Lustre exprimant les propriétés de séquences finies d'intervalles et de réaction définis dans la deuxième partie de cette thèse (chapitre 3) sont donnés en annexe. A titre d'exemple, le noeud Lustre exprimant la propriété Once_since_ est donné en figure 9.5. Ce noeud retourne la valeur booléenne 1, si et seulement si, A et B valent la valeur 1 ou bien B vaut 0 et A valait au moins une fois la valeur 1 depuis la dernière fois que B valait 1.

```
node ONCE_SINCE_ (A, B: bool) returns (Once_since: bool);
let
  Once_since  = if B then A else (A or pre (Once_since));
tel.
```

Figure 9.5. Le noeud Lustre décrivant l'opérateur Once_since

9.2.2 Spécification des propriétés de sûreté en Lustre

Rappelons que, de façon générale, une propriété de sûreté s'exprime de la manière suivante : pendant un intervalle d'observation, chaque fois qu'un événement spécifique se produit (le contexte), une relation relative à un ensemble d'événements (la réaction) est vérifiée. Par conséquent, le programme Lustre exprimant la propriété de sûreté sera construit à l'aide des noeuds décrivant les diverses propriétés de séquences finies. La notion hiérarchique du langage Lustre permet aisément une telle construction.

La structure du programme Lustre SAFETY exprimant une propriété de sûreté est donnée par la figure 9.6. Ce programme retourne la valeur booléenne 1, si et seulement si, à chaque fois que les noeuds exprimant les propriétés d'intervalle et de contexte retournent la valeur 1, le noeud exprimant la propriété de réaction retourne aussi la valeur 1. Notons que l'opérateur "Always" exprime le fait que le langage associé à la propriété de sûreté est clos par préfixe. Ainsi, si à un instant donné le programme retourne la valeur 0, alors quel que soit le comportement futur du programme la valeur générée est 0.

```
node SAFETY (():  bool) returns (Safety: bool);
var P: bool;
let
-- I () , C () et R () sont les noeuds décrivant les propriété d'intervalle
   d'observation, de contexte et de réaction
  P       = not (I ()) or (not (C ()) or R ());
  Safety = ALWAYS (P);
tel.

node ALWAYS (a: bool) returns (Always: bool);
let
  Always = a and (true -> pre (Always));
tel.
```

Figure 9.6. La structure d'un noeud Lustre décrivant une propriété de sûreté

9.2.3 Exemple de spécification d'une propriété de sûreté

Sur l'exemple de l'arbitre de bus donné en section 9.1 (§ 9.1.4), la propriété : "l'émission d'un acquittement gr_i (i = 0, 1, 2) doit se faire en direction de l'unité la plus prioritaire ayant émis une requête d'allocation du bus req_i", est une propriété de sûreté assurant un bon fonctionnement du circuit. Cette propriété désignée par *réactivité* exprime le fait qu'à chaque occurrence d'une transition montante de gr_i notée t_gr_i, une transition montante de req_i (de priorité supérieure) notée t_req_i a eu lieu depuis la dernière occurrence de t_gr_i. Le contexte temporel de cette propriété est l'événement t_gr_i et sa réaction est la présence d'au moins une transition montante du signal req_i depuis la dernière occurrence de t_gr_i. Nous considérons que l'intervalle d'observation est l'intervalle temporel $[t_0\ +\infty]$.

La propriété de réactivité s'exprime à l'aide de l'opérateur *Once_between_andlast_*, défini au chapitre 3 (§ 3.3.5), relatif aux événements définis : t_req_i et t_gr_i. Rappelons que l'opérateur *Once_between_andlast_ (P, Q)* est défini comme suit : une séquence σ satisfait cet opérateur, si et seulement si, pour tout intervalle séparant deux occurrences successifs de l'événement Q, l'événement P s'est produit au moins une fois.

Le noeud Lustre décrivant la propriété de réactivité est donné par la figure 9.7. Ce noeud retourne la valeur booléenne 1, si et seulement si, à chaque transition montante du signal

d'allocation du bus : gr_i (i = 0, 1, 2) , une transition montante du signal de demande de bus : req_i (i = 0, 1, 2) a eu lieu depuis la dernière transition montante du signal gr_i.

```
node REACTIVITY (req, gr: bool^3) returns (Reac: bool);
var t_req, t_gr: bool^3;
let
  t_req = P_EDGE (req);

  t_gr  = P_EDGE (gr);

-- La propriété Once_between_andlast_ relative aux événements t_req et t_gr
  R     = ONCE_BETWEEN_ANDLAST_ (t_req, t_gr);

-- La propriété de réactivité sur les trois lignes de l'arbitre de bus
 Reac  = ALWAYS (R[0] and R[1] and R[2]);
tel.

-- Description de l'opérateur Once_between_andlast_
node ONCE_BETWEEN_ANDLAST_ (A, B: bool) returns (Once_B&L: bool);
let
  Once_B&L = not (B) or (ONCE_SINCE_ (A, B -> pre(B)));
tel.

-- Description de l'événement défini : transition montante sur un signal
   d'entrée
node P_EDGE (In: bool) returns (P_edge: bool);
let
  P_edge = In and not (true -> pre (In));
end.
```

Figure 9.7. La propriété de réactivité en Lustre

9.2.4 Programme de vérification

Nous avons vu en section 9.1 que la description d'un SMSS, quel que soit le niveau d'abstraction, peut être spécifiée par un programme Lustre. Ce programme de description P_d dénote une machine d'états finis M_d définie en chapitre 8 (section 8.3) par le 6-uplet $(Q_d, E_d, S_d, \delta_d, \lambda_d, q_0)$.

Nous avons aussi vu que toute propriété de sûreté d'un programme Lustre décrivant un SMSS peut être exprimée dans le langage Lustre. Le noeud Lustre P_p décrivant la propriété de sûreté a comme entrées un sous-ensemble des entrées et sorties du programme P_d décrivant le système matériel à vérifier. Le programme P_p dénote une machine d'états finis définie par le 6-uplet $(Q_p, E_p, \{0, 1\}, \delta_p, \lambda_p, q_0)$.

L'instantiation des noeuds P_d et P_p dans le programme de vérification P_v, comme le montre le figure 9.8, consiste à connecter leurs machines associées dans le but de définir la machine de vérification M_v présentée dans le chapitre 4. Le programme de vérification P_v dénote donc la machine de vérification M_v définie par le 6-uplet $(Q_v, E_v, \{0, 1\}, \delta_v, \lambda_v, q_0)$, où l'ensemble d'états $Q_v = Q_d \times Q_p$, l'ensemble d'entrées $E_v = E_d$, δ_v est la fonction de transition définie de $Q_v \times E_v$ dans Q_v, λ_v est la fonction de sortie définie de $Q_v \times E_v$ dans $\{0, 1\}$. Le problème de la vérification se ramène à vérifier que le flot de sortie du programme de vérification P_v, dénotant la machine de vérification M_v, est la constante 1. Cette vérification sera réalisée par l'outil de preuve Lesar (chapitre 10).

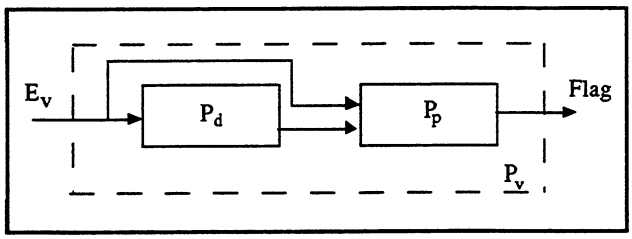


Figure 9.8. Vérification d'une propriété de sûreté

9.2.5 Exemple

La méthode que nous venons de décrire est appliquée à l'arbitre de bus donné en section 9.1. Nous donnons une spécification du circuit sous forme d'un graphe d'états (figure 9.9) sur lequel nous voulons vérifier la propriété de réactivité. Cet arbitre à quatre états A, B, C et D. Dans son état initial A, le bus est libre. Dans les états B, C et D, le bus est utilisé par les unités 0, 1 et 2 respectivement. Le noeud Lustre `ARBITER_SPEC` décrivant cette spécification est donné en figure 9.10.

La réactivité exprime le fait que toute allocation du bus est précédée par une demande de priorité supérieure. Le noeud Lustre `REACTIVITY` donnant la description de cette propriété a été donné en figure 9.7. Le programme de vérification `VERIFY` est défini par l'instantiation des noeuds décrivant la spécification `ARBITER_SPEC` et la propriété `REACTIVITY`, comme le montre la figure 9.10. Ce programme dénote la machine de vérification sur laquelle la vérification de la propriété de réactivité sera réalisée.

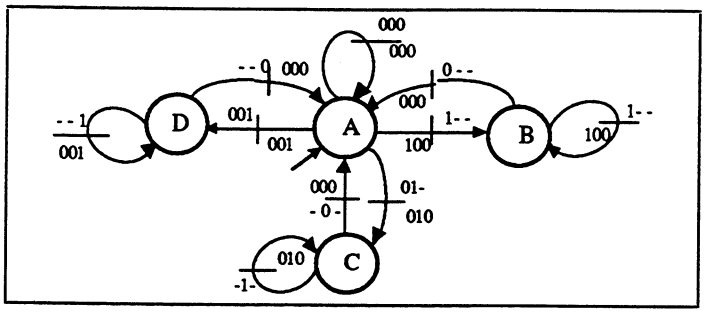


Figure 9.9. Graphe d'états de l'arbitre de bus

```
node ARBITER_SPEC (req: bool^3) returns (gr: bool^3);
var A, B, C, D: bool;
let
-- Etats de l'arbitre
  A = true  -> (pre(A) and (not(req[0]) and not(req[1]) and not(req[2])))
               or (pre(B) and (not req[0]))
               or (pre(C) and (not req[1]))
               or (pre(D) and (not req[2]));

  B = false -> (pre(A) or pre(B)) and req[0];

  C = false -> (pre(A) and ((not req[0]) and req[1]))
               or (pre(C) and req[1]);

  D = false -> (pre(A) and (not(req[0]) and not(req[1]) and req[2]))
               or (pre(D) and req[2]);

  gr = B | C | D;
tel.

node VERIFY (req: bool^3) returns (Flag: bool);
let
-- Spécification de l'arbitre
 gr = ARBITER_SPEC (req);

-- Vérification de la propriété de réactivité sur l'arbitre
Flag  = REACTIVITY (req[0], gr[0]) and  REACTIVITY (req[1], gr[1]) and
        REACTIVITY (req[2], gr[2]);
tel.
```

Figure 9.10. Vérification de la propriété de réactivité sur une spécification de l'arbitre de bus en Lustre.

9.3 Vérification de conformité

La vérification de conformité consiste à vérifier la validité de la description d'une réalisation possible d'un SMSS vis-à-vis de la description de sa spécification [BT 90]. Nous avons vu en section 9.1 que le langage Lustre permet la description d'un SMSS à ces deux niveaux d'abstraction.

Dans le chapitre 5, nous avons présenté une approche de vérification de conformité basée sur le modèle de machines d'états finis. La méthode de vérification consiste à définir une machine de vérification ayant une unique sortie sur laquelle la vérification sera réalisée. Cette machine de vérification est le produit des deux machines comparables associées aux descriptions de la réalisation et de la spécification. Les machines comparables sont obtenues en appliquant les opérateurs de comparaison, définis en chapitre 5 (section 5.1), sur les flots d'entrée et de sortie des machines associées aux descriptions de la réalisation et de la spécification .

Dans cette section, nous montrons que le langage Lustre permet de décrire le programme de vérification relatif au problème de la vérification de conformité. Ce programme dénote la machine de vérification décrite en chapitre 5 (section 5.2). Nous présentons la description Lustre des opérateurs de comparaison, des machines comparables et de la machine de vérification.

9.3.1 Les opérateurs de comparaison

Nous donnons, dans ce paragraphe, la description Lustre des opérateurs de comparaison. Rappelons que ces opérateurs sont l'opérateur de synchronisation Σ, l'opérateur de retard Δ et l'opérateur parallèle Π, qui ont été définis en chapitre 5 (section 5.1).

9.3.1.a. L'opérateur de synchronisation Σ

L'opérateur de synchronisation Σ permet de maintenir une machine dans un état donné tant qu'un événement particulier, caractérisé par une valeur booléenne, est absent. Cet opérateur est appliqué lorsque les horloges de la réalisation sont explicites en maintenant la machine associée à la spécification dans un état donné jusqu'au prochain top du signal qui définit l'horloge de la réalisation.

Grâce aux opérateurs temporels : "when" et "current", le langage Lustre permet la description de cet opérateur. Soient x la variable booléenne caractérisant un signal de sortie et Ck la variable booléenne caractérisant le signal d'horloge de la réalisation. La synchronisation de la variable x par rapport à la variable Ck est réalisée par l'expression "current x when Ck". L'opérateur d'échantillonnage "when" filtre x sur l'horloge Ck et l'opérateur "current" projette l'expression "x when Ck" sur l'horloge de base définie par la suite constante (1, 1, ...). D'une manière générale, la description Lustre de l'opérateur de synchronisation d'un vecteur A de *n* booléens par rapport à une variable booléenne Ck est donnée en figure 9.11. Le noeud SYNC décrivant l'opérateur de synchronisation reçoit en entrée un vecteur de booléens A et un paramètre *n* et retourne le flot synchronisé A_sync.

```
node SYNC (const n: int; A: bool^n, Ck: bool) returns (A_sync: bool^n);
let
  A_sync = current (A when Ck);
tel.
```

Figure 9.11. L'opérateur de synchronisation en Lustre

9.3.1.b. L'opérateur de retard Δ_n

L'opérateur de retard Δ_n permet de décaler un flot de sortie vers la droite. Cet opérateur est appliqué dans le cas où la réalisation du SMSS a une structure pipeline, en retardant le flot de sortie de la spécification pour que les deux flots des deux machines soient en phase.

La description Lustre de cet opérateur est réalisée grâce à l'opérateur de retard "pre". Une description générale de cet opérateur est donnée en figure 9.12. Le noeud DELAY reçoit un flot x et un paramètre entier *n* et retourne un flot x_ret retardé de *n* tops d'horloge.

```
node DELAY (const n: int; x: bool) returns (x_ret: bool);
var B: bool^n;
let
  B     = false -> pre (x) | false^n-1 -> pre B[0 .. n-2];
  x_ret = B[n-1];
tel.
```

Figure 9.12. L'opérateur de retard en Lustre

9.3.1.c. L'opérateur parallèle Π

Cet opérateur comprime une séquence σ d'un facteur *n* en la décalant à droite. L'opérateur parallèle est utilisé lorsque la réalisation ou la spécification a une structure série alors que l'autre est parallèle. Le noeud Lustre PAR décrivant l'opérateur parallèle Π est donné en figure 9.13. Ce noeud reçoit en entrée un flot booléen x et un paramètre *n*, retourne un vecteur de flots de booléens X_p. Chaque valeur de X_p est une compression du vecteur d'entrée d'un facteur *n*. Initialement, toutes les valeurs du vecteur A_p sont à 0.

```
node PAR (const n: int; a: bool) returns (A_p: bool^n);
let
  A_p  = false -> pre (a) | false^n-1 -> pre A_p[0 .. n-2];
tel.
```

Figure 9.13. Le noeud Lustre décrivant l'opérateur parallèle

9.3.2 Obtention des machines comparables en Lustre

Nous avons vu en chapitre 5 que les opérateurs de comparaison nous permettent d'obtenir des machines dites comparables. Ces machines ont d'une part, le même vocabulaire d'entrée et de sortie, et d'autre part elles sont cadencées par la même horloge. Rappelons que ces machines dépendent du niveau d'abstraction des descriptions de la réalisation et de la spécification et de leurs structures : parallèle, série ou pipeline. Nous avons étudié en chapitre 5 les différents cas de machines comparables. Nous étudions ici de façon générale le cas de comparaison entre une description d'une réalisation avec une horloge explicite Ck et une spécification de haut niveau sans horloge. Les deux descriptions dénotent deux machines d'états finis de même structure.

Considérons le programme Lustre P décrivant la spécification d'un SMSS. Le programme P retourne le vecteur A de *m* booléens et reçoit un vecteur B de *n* booléens. Ce programme dénote la machine d'états finis $M = (Q, E, S, \delta, \lambda, q_0)$ où S est l'ensemble de *m* symboles de sortie. La synchronisation du flot de sortie du programme P par rapport au flot d'horloge Ck de la réalisation consiste à définir un nouveau programme P_SYNC par instantiation du programme P et du programme décrivant l'opérateur de synchronisation SYNC, comme le montre la figure 9.14. Le programme P_SYNC dénote la machine d'états finis $M^S = (Q^S, E \times Ck, S, \delta^S, \lambda^S, q_0^S)$ où :

- $Q^S = Q \times S$;
- $\forall (e, q) \in \{E \times Q\}, \delta^S ((e, 1), (q, s)) = \delta (e, q)$ et $\delta^S ((e, 0), (q, s)) = q$;

• $\forall$ (e, q) $\in$ {E × Q}, λ^S ((e , 1), (q, s)) = λ (e, q) et λ^S((e , 0)„ (q, s)) = s.

La machine M^S et la machine associée à la description de la réalisation sont dites comparables par rapport au langage d'entrée E × Ck.

```
node P_SYNC (const m: int; const n: int; A: bool^n; Ck:bool)
            returns (B_sync: bool^m);
var B: bool^m;
let
-- Appel du noeud P décrivant la description du SMSS
  B = P (n, A);
-- Synchronisation du flot de sortie sur Ck
  B_sync = SYNC (m, B, Ck);
tel.
```

Figure 9.14. Synchronisation de la spécification d'un SMSS en Lustre

9.3.3 Programme de vérification

Le programme de vérification P_V (figure 9.15) est obtenu par instantiation des programmes associés aux descriptions des deux machines comparables P_{d1} et P_{d2}, du programme décrivant les critères d'observation P_C et du programme EVAL. Ce dernier retourne la sortie Flag et reçoit les flots de sortie des deux machines comparables et de la machine associée aux critères d'observation. Rappelons que ces critères d'observation sont des événements définis ou des intervalles temporels bornés par des événements définis. Formellement, il s'agit de propriétés de séquences finies d'événements définis ou d'intervalles. La description dans le langage Lustre de telles propriétés a été étudiée dans la section 9.2.

Ce programme de vérification P_V dénote une machine d'états finis M_V définie par le 6-uplet (Q_V, E_V, S_V, δ_V, λ_V, q_0) où l'ensemble d'états $Q_V = Q_{d1} \times Q_{d2} \times Q_C$, δ_V est la fonction de transition définie de $Q_V \times E_V$ dans Q_V, λ_V est la fonction de sortie définie de $Q_V \times E_V$ dans {0, 1}.

Comme dans le cas de la vérification de propriétés de sûreté, le problème de vérification consistera à vérifier que le flot de sortie du programme de vérification P_V dénotant la machine de vérification M_V est la constante 1. Cette vérification sera réalisée par l'outil de preuve Lesar (chapitre 10).

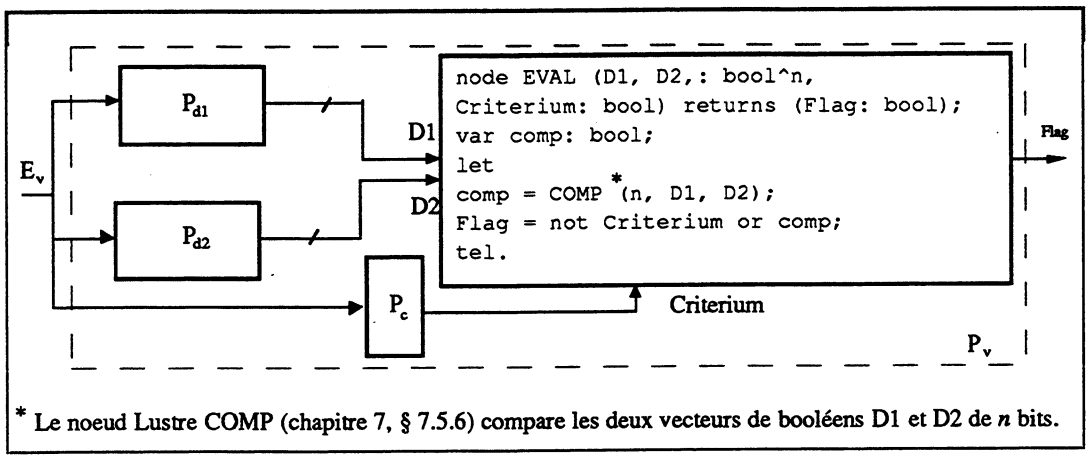


Figure 9.15. Vérification de conformité

La méthode que nous venons de décrire sera appliquée, dans la section 9.4, pour vérifier la conformité de la réalisation de l'arbitre de bus (§ 9.1.4) vis-à-vis de sa spécification (§ 9.2.4) dans son environnement de fonctionnement.

9.4 Vérification sous un environnement

Dans cette section, nous montrons que le langage Lustre permet de décrire les reconnaisseurs de propriétés d'environnement et les générateurs de langage accepté par un système matériel définis dans le chapitre 6. Nous abordons ensuite le problème de la vérification sous un environnement. Puis, nous étudions les problèmes de vérification de propriétés et de conformité sous un environnement sur l'arbitre de bus défini précédemment.

9.4.1 Description du modèle reconnaisseur de l'environnement

Dans cette approche de modélisation nous définissons un reconnaisseur de séquences ayant pour modèle une machine d'états finis. Cette machine décrit le langage de la propriété d'environnement.

Les assertions du langage Lustre permettent la description d'un reconnaisseur de séquences. Soit E l'expression booléenne décrivant la propriété d'environnement P_e, "assert E" spécifie que l'expression E est toujours vraie. En d'autre termes, l'assertion "assert E" spécifie que la propriété d'environnement P_e est toujours satisfaite.

Par exemple, pour exprimer dans le langage Lustre le fait qu'une entrée *a* d'un SMSS ne peut être vraie à deux instants consécutifs, nous définissons l'expression Lustre suivante :

$$E = \text{not (a and (false} \rightarrow \text{pre (a)))}$$

L'expression E vaut la valeur 0 à chaque fois que la variable *a* vaut la valeur booléenne 1 à deux instants consécutifs. L'assertion "assert E" spécifie que l'expression E est toujours vraie ; c'est-à-dire que toutes les séquences où la variable a vaut la valeur booléenne 1 à deux instants consécutifs ne sont pas valides.

D'une façon générale, pour définir un reconnaisseur de séquences dans le langage Lustre, nous définissons d'abord le noeud P_ENV décrivant la propriété d'environnement P_e. Ce noeud a comme paramètres d'entrée un sous-ensemble des entrées et / ou des sorties du SMSS. Le noeud P_ENV retourne la valeur booléenne 1, si et seulement si, les entrées du noeud satisfont la propriété d'environnement P_e. L'instantiation du noeud P_ENV dans le programme REC donné en figure 9.16 définit le reconnaisseur des séquences satisfaisant la propriété d'environnement P_e.

La machine d'états finis M_r associée au programme décrivant le modèle reconnaisseur de l'environnement est obtenue à partir de la machine M_p associée au programme décrivant la propriété d'environnement. La machine M_r est obtenue à partir de la machine M_p en introduisant un état puits q_p tel que toutes les transitions de la machine M_p pour lesquelles la propriété d'environnement n'est pas satisfaite (c'est-à-dire, les transitions pour lesquelles la fonction de sortie vaut 0), mènent vers cet état puits :

$$\forall\ (e,q) \in (E_p \times Q_p)\ ,\ \delta_r\ (e, q) = q_p \qquad \text{si} \qquad \lambda_p\ (e, q) = 0$$

L'exemple du graphe d'états de la machine d'états finis associée au programme décrivant l'environnement reconnaisseur : l'entrée *a* ne peut pas être vraie à deux instants consécutifs, est donné en figure 9.17.

```
node REC ():bool) returns (rec: bool);
var P: bool;
let
-- Appel du noeud P décrivant la propriété d'environnement
  P = P_ENV ();

-- Définition des séquences satisfaisant l'environnement
  rec = assert P;
tel.
```

Figure 9.16. Structure du noeud décrivant un reconnaisseur en Lustre

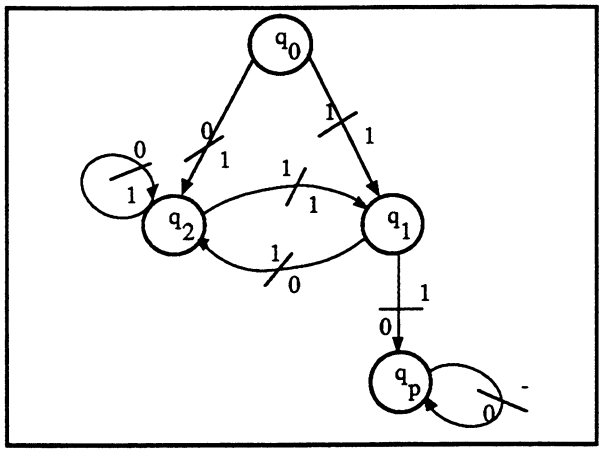


Figure 9.17. Graphe d'états du reconnaisseur de séquences : $(0^* 1 0 0^*)^*$

9.4.2 Description d'un environnement générateur

Dans cette approche de modélisation, nous définissons un générateur du langage accepté par un système matériel. Nous avons montré dans le chapitre 6 qu'il était possible de définir une machine d'états finis générant ce langage régulier. Le langage Lustre permet la description de tels générateurs. Pour expliciter ceci, reprenons l'exemple du paragraphe 9.4.1, dans lequel l'entrée *a* d'un SMSS ne peut être vraie à deux instants consécutifs. Le langage accepté L_a défini par cet environnement est le suivant :

$$L_a = (0^* 1 0 0^*)^*$$

Pour générer le langage L_a à l'aide du langage Lustre, nous définissons le noeud GNE (figure 9.18), recevant la variable d'entrée *a* et retournant la variable de sortie *b*. A chaque

instant, si la valeur 1 a été retournée à l'instant précédent, le noeud GNE retourne la valeur 0. Sinon, le noeud GNE retourne la valeur de *a* en sortie.

Dans certains cas, la description d'un générateur du langage accepté par un SMSS est aisée à écrire en Lustre. Par exemple pour générer deux signaux exclusifs et alternés, il suffit de définir un noeud recevant la variable d'entrée *a* et retournant les deux variables de sortie x et y tel que : x = a and not pre (a) et y = not a and pre (a). En d'autres termes, x est vraie à chaque fois que *a* passe de 0 à 1 et y est vraie à chaque fois que *a* passe de 1 à 0. Ainsi, les deux variables x et y sont d'une part exclusives et d'autre part alternées.

```
node GEN (a:bool) returns (b: bool);
let
  b = if (false -> pre b) and a then false else a;
tel.
```

Figure 9.18. Le générateur du langage $(0^* \; 1 \; 0 \; 0^*)^*$ en Lustre

9.4.3 Vérification avec un modèle reconnaisseur de l'environnement

L'instantiation du programme P_v décrivant la machine de vérification (définie en sections 9.2 et 9.3) et du programme P_r décrivant l'environnement reconnaisseur (paragraphe 9.4.1) dans le programme de vérification avec un environnement reconnaisseur $P_{v/r}$, consiste à connecter leurs machines associées dans le but de définir la machine de vérification $M_{v/r}$ présentée dans le chapitre 6. La machine $M_{v/r}$ est caractérisée par le 6-uplet $(Q_{v/r}, E_{v/r}, \{0, 1\}, \delta_{v/r}, \lambda_{v/r}, q_0)$. Le problème de vérification se ramène à vérifier que le flot de sortie du programme de vérification $P_{v/r}$ dénotant la machine de vérification $M_{v/r}$ est la constante 1.

Cependant, la méthode de définition d'un reconnaisseur de séquences qui consiste d'abord à définir un programme décrivant la propriété d'environnement puis à l'"asserter", peut engendrer la définition d'un *reconnaisseur non causal* (chapitre 6, § 6.). Rappelons qu'un reconnaisseur non causal a dans son espace d'états au moins un état de transition menant uniquement vers l'état puits du reconnaisseur. La définition d'un reconnaisseur non causal peut conduire à des résultats erronés lors de la vérification, du fait de l'incompatibilité entre l'interprétation intuitive des assertions et leurs sémantique exacte. Pour expliciter ceci considérons l'exemple du paragraphe 9.4.1 : l'entrée *a* d'un SMSS ne peut pas être vraie à deux instants consécutifs. L'expression Lustre exprimant la propriété d'exclusivité d'une variable *a* à deux instants consécutifs peut s'écrire, en faisant intervenir les deux dernières valeurs de l'entrée *a*, comme suit :

E = not ((false → pre a) and (false → pre (false → pre a))) ;

L'assertion "assert E" dénote la machine d'états finis M_E donnée en figure 9.19. Notons que la machine M_E a un état de transition menant uniquement vers l'état puits et que le reconnaisseur a reconnu la séquence initiale "11" ne satisfaisant pas la propriété d'environnement.

L'exemple que nous venons d'examiner a montré qu'une incompatibilité entre le sens intuitif des assertions et leur sémantique peut fausser les résultats de la vérification. La solution retenue pour résoudre le problème des assertions non causales, au cours du processus de vérification, consiste à considérer ces états de transitions menant inévitablement vers l'état puits du reconnaisseur comme des états non valides. Cette tâche est réalisée par l'outil de preuve Lesar (chapitre 10).

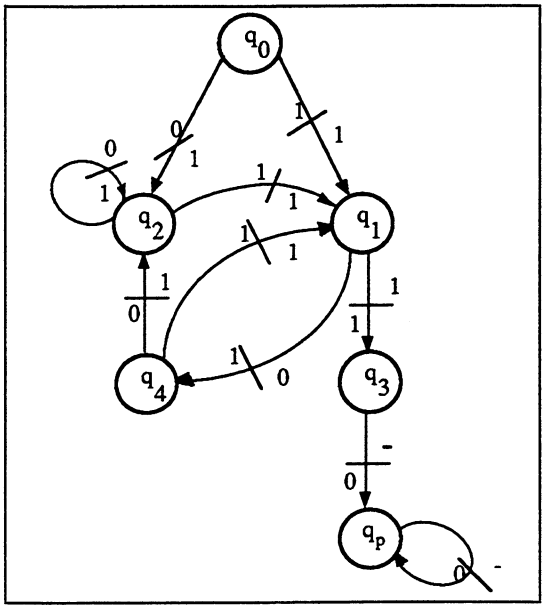


Figure 9.19. Le graphe d'états d'un exemple de reconnaisseur non causal

9.4.4 Vérification avec un modèle générateur de l'environnement

L'instantiation du programme décrivant la machine de vérification P_v (définie en sections 9.2 et 9.3) et du programme décrivant l'environnement générateur P_g (§ 9.4.2) dans le programme de vérification avec un environnement générateur $P_{v/g}$, consiste à connecter leurs machines associées dans le but de définir la machine de vérification $M_{v/g}$ présentée dans le chapitre 6. La machine $M_{v/g}$ est caractérisée par le 6-uplet $(Q_{v/g}, E_{v/g}, \{0, 1\}, \delta_{v/g}, \lambda_{v/g}, q_0)$. Le problème de vérification se ramène aussi à vérifier que le flot de sortie du programme de vérification $P_{v/g}$ dénotant la machine de vérification $M_{v/g}$ est la constante booléenne 1.

9.4.5 Exemples de vérification sous un environnement

Dans ce paragraphe nous étudions deux exemples de vérification (1) de propriétés et (2) de conformité sur l'arbitre de bus.

9.4.5.a Exemple 1

Dans cet exemple, la propriété d'alternance est vérifiée sur la description de la réalisation de l'arbitre de bus (§ 9. 1.4). Cette propriété exprime l'alternance entre les signaux de demande de bus et les signaux d'allocation du bus. En d'autres termes, cette propriété désignée par *alternance* exprime le fait qu'entre deux occurrences successives d'une transition montante du signal de demande de bus, une et une seule transition montante du signal d'allocation de bus a eu lieu.

La propriété d'alternance s'exprime à l'aide de l'opérateur *Once_between_andlast_*, défini au chapitre 3 (§ 3.3.5), relatif aux événements définis : t_req_i, t_gr_i et t_req_i. La description Lustre de cet opérateur est donnée en Annexe. Le noeud ALTERNATE décrivant la propriété d'alternance est donné en figure 9.21. Ce noeud retourne la valeur booléenne 1, si et seulement si, une et une seule transition montante du signal d'allocation du bus : gr_i (i = 0, 1, 2) a lieu, entre tout intervalle minimal défini par l'occurrence de deux transitions montantes du signal de demande de bus : req_i (i = 0, 1, 2).

La propriété d'alternance caractérise un comportement de l'arbitre de bus dans son environnement de fonctionnement. D'une part, le circuit arbitre de bus est intégré dans un environnement comprenant trois unités de traitement, tel que toute demande de bus est maintenue tant que l'arbitre ne lui accorde pas le bus. Autrement dit, l'entrée req_i de l'arbitre est maintenue à 1 tant que sa sortie gr_i est à 0. D'autre part, la réalisation donnée de l'arbitre de bus a une structure pipeline à deux étages commandés par deux signaux d'horloges exclusifs et alternés.

Pour considérer cet environnement, durant la vérification, nous définissons le noeud Lustre ENVIRONMENT (figure 9.21). Ce noeud comprend à la fois (1) un générateur de séquences et (2) un reconnaisseur de séquences :

(1) Le générateur permet de générer les signaux d'horloges. La description Lustre de ce générateur est donnée au paragraphe 9.4.2 .

(2) Le reconnaisseur définit les séquences satisfaisant la propriété d'environnement : "quand une unité envoie une demande de bus, elle doit continuer d'en faire la demande jusqu'à ce

que l'arbitre retourne un acquittement". Cette propriété s'exprime à l'aide de l'opérateur *Always_from_to_*, défini en chapitre 3 (§ 3.3.5), relatif aux événements définis : req_i, t_req_i et t_gr_i. Le noeud Lustre ALWAYS_FROM_TO_ (annexe) décrivant cet opérateur est alors "asserté" et l'assertion définit un reconnaisseur de séquences.

L'instantiation des noeuds Lustre ARBITER_IMP (figure 9.4), ALTERNATE et ENVIRONMENT dans le programme de vérification VERIFY (figure 9.20) consiste à connecter leurs machines associées dans le but de définir la machine de vérification sur laquelle la vérification de la propriété d'alternance sera réalisée. La vérification consiste à vérifier que l'unique sortie "Flag" du programme de vérification VERIFY vaut toujours la valeur booléenne 1.

```
node VERIFY (req, a: bool) returns (Flag: bool);
var Ck_1, Ck_2 : bool;
let
-- Réalisation de l'arbitre
 gr = ARBITER_IMP (req, Ck_1, Ck_2);

-- Environnement
(Ck_1, Ck_2) = Pe (req, gr, a);

-- Propriété d'alternance
Flag  = ALTERNATE (req[0], gr[0]) and  ALTERNATE (req[1], gr[1]) and
        ALTERNATE (req[2], gr[2]);
tel.

node ALTERNATE (req_i, gr_i: bool) returns ( alter: bool);
var t_req_i, t_gr_i: bool;
let
-- Transitions montantes de req_i et gr_i
t_req_i = req_i -> (req_i and not pre(req_i));
t_gr_i  = gr_i  -> (gr_i and not pre(gr_i));

-- Propriété d'alternance
alter = ONCE_BETWEEN_AND LAST_ (t_gr_i, t_req_i, t_req_i) and
        ONCE_BETWEEN_AND LAST_ (t_req_i, t_gr_i, t_gr_i) ;
tel.
```

Figure 9.20. Vérification de la propriété d'alternance sur l'arbitre de bus en Lustre

```
node ENVIRONMENT (req, gr, a: bool^3) returns (Ck_1, Ck_2: bool);
var P, t_req, t_gr: bool^3;
let
-- Générateur de séquences
(Ck_1, Ck_2) = GEN (a);

-- Reconnaisseur de séquences
P = RECOGNIZER (req, gr);
assert P;
tel.

node GEN (a : bool) returns (Ck_1, Ck_2: bool);
let
Ck_1 = true  -> a and not pre (a);
Ck_2 = false -> not (a) and pre (a);
tel.

node RECOGNIZER (req, gr: bool^3) returns (P: bool);
var t_req: bool^3;
let
-- Transitions montantes des req_i et gr_i
t_req = R_EDGE (req, req);
t_gr  = R_EDGE (gr, gr);

-- Propriété d'environnement
P = ALWAYS_FROM_TO_ (req, t_req, t_gr);

P  = P[0] and P[1] and P[2];
tel.

node R_EDGE (init, a: bool) returns (r_edge: bool);

let

r_edge = init -> (a and (not pre (a)));

tel
```

Figure 9.21. L'environnement de l'arbitre arbitre de bus en Lustre

9.4.5.b Exemple 2

Dans cet exemple, nous examinons la conformité de la description de la réalisation de l'arbitre de bus, donnée au paragraphe 9.1.4 (figure 9.4), vis-à-vis de sa spécification donnée au paragraphe 9.2.4 (figure 9.10). Cette vérification est réalisée sous l'environnement de fonctionnement de l'arbitre. La description Lustre de cet environnement a été donnée en figure 9.21.

La spécification considérée de l'arbitre de bus n'a pas d'horloge explicite, alors que la réalisation a une architecture pipeline commandée par deux horloges exclusives et alternées. La comparaison des flots de sortie des deux machines associées à la spécification et à la réalisation nécessite une synchronisation de la machine spécification par rapport aux horloges de la réalisation. Cette synchronisation est réalisée à l'aide de l'opérateur de synchronisation par rapport à l'horloge CK_1 qui rythme les entrées de la réalisation. Du fait que les sorties de la réalisation sont cadencées sur l'horloge CK_2, la comparaison entre les flots de sortie des machines associées à la réalisation et à la spécification synchronisée est réalisée à des instants définis par cette horloge. Le programme de vérification est obtenu par instantiation des noeuds Lustre ARBITER_IMP (figure 9.4), ARBITER_SPEC (figure 9.10) et ENVIRONMENT (figure 9.21), dans le programme VERIFY donné en figure 9.20. Ce programme Lustre dénote la machine de vérification sur laquelle la vérification de conformité sera réalisée.

```
node VERIFY (a: bool; req: bool^3) returns (Flag: bool);
var criterion, Ck_1, Ck_2: bool; var ss_gr, s_gr, r_gr: bool^3;
let
-- Réalisation et spécification de l'arbitre
 r_gr = ARBITER_IMP (req, Ck_1, Ck_2);
 s_gr = ARBITER_SPEC (req);
-- Spécification synchronisée
 ss_gr = SYNC (s_gr, (true -> Ck_1));
-- Environnement
(Ck_1, Ck_2) = ENVIRONMENT (req, gr, a);
-- Critère d'observation
criterion = false -> pre (Ck_2);
-- Comparaison
Flag  = not (criterion) or COMP (3, ss_gr, r_gr);
tel.
```

Figure 9.22 La vérification de conformité de l'arbitre de bus en Lustre.

9.5 Conclusion

Nous avons montré dans ce chapitre que le langage Lustre permet de décrire les SMSS aux différents niveaux de conception (de la spécification à la réalisation), de spécifier les propriétés de sûreté complexes relatives aux comportements des SMSS et de décrire leurs environnements de fonctionnement. Nous avons ensuite montré que le langage Lustre permet de décrire un programme de vérification avec une seule sortie Flag, pour résoudre les deux problèmes de vérification de propriétés et de conformité. Ce programme dénote la machine de vérification que nous avons définie dans la deuxième partie de ce document. Sur cette machine, la vérification sera réalisée par l'outil de preuve Lesar qui sera présenté au chapitre suivant. Lesar évalue la formule "Flag =1" sur le modèle de machines d'états finis associé au programme de vérification en utilisant deux stratégies d'exploration du graphe d'états de la machine de vérification.

Chapitre 10

L'outil de vérification Lesar

L'outil de vérification Lesar [Glo 89, Rat 92] permet de vérifier que l'unique sortie de la machine d'états finis associée à un programme Lustre est toujours vraie. Dans cet outil, deux stratégies de vérification ont été implémentées correspondant à deux méthodes d'exploration du graphe d'états de la machine. Dans les deux stratégies, l'exploration du graphe est réalisée sans le construire (exploration "à la volée").

Dans ce chapitre, nous explicitons d'abord la méthode d'identification de la machine d'états finis associée à un programme Lustre opérant sur des flots booléens. Ensuite, les méthodes utilisées par l'outil Lesar pour explorer le graphe d'états de la machine associée au programme Lustre sont étudiées.

10.1 Obtention de la machine d'états finis

Après la phase de vérification statique d'un programme Lustre : analyse syntaxique, vérification de types, vérification d'inter-blocages et calcul d'horloge, le programme est traduit dans un format intermédiaire EC (de l'anglais, Expanded Code). L'obtention du modèle d'exécution associé à un programme Lustre s'effectue à partir du programme réécrit au format EC. Ces tâches sont réalisées par le compilateur Lustre [Pla 88, Ray 91]. L'obtention du modèle de machine d'états finis, est fondée sur une phase de normalisation et une phase d'identification. Ces deux phases sont décrites dans ce qui suit.

10.1.1 Normalisation du programme

La normalisation d'un programme Lustre réécrit au format EC [Ray 91] consiste en une transformation syntaxique de ce programme. Le but de cette opération est de remplacer les opérateurs temporels : "pre", "→", "when" et "current", par des constructions syntaxiques classiques. Cette transformation uniformise la notion de mémoire avec l'introduction de deux opérateurs binaires : *last* et *init*. L'introduction de l'opérateur *last* définit l'implémentation des opérateurs "pre", "when" et "current" et celle de l'opérateur *init* définit l'implémentation de l'opérateur d'initialisation "→". Après la normalisation, le programme Lustre peut être représenté par un nouveau réseau d'opérateurs.

Pour illustrer la normalisation de programmes, considérons le noeud Lustre (figure 10.1) décrivant un détecteur de fronts. Le nouveau programme obtenu après normalisation, est représenté par le réseau d'opérateurs de la figure 10.2.

```
node EDGE_DETECT (e, Ck: bool) returns (s: bool);
var v1, v2: bool;
let
 v1 = current (e when true -> Ck);
 v2 = false -> pre (v1);
  s = e and (not v2);
tel.
```

Figure 10.1. Le programme Lustre décrivant un détecteur de fronts

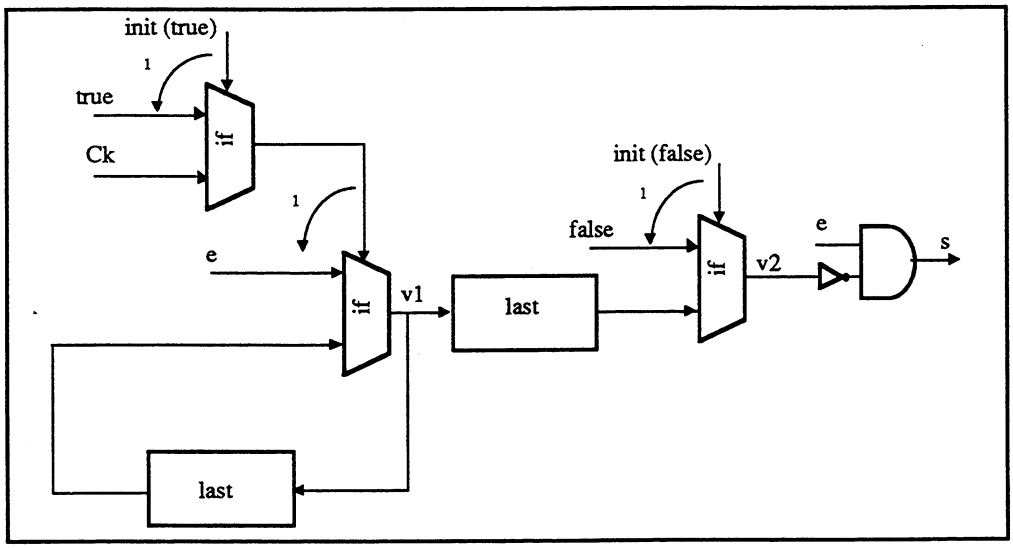


Figure 10.2. Un réseau d'opérateurs normalisé

10.1.2 Identification du modèle

Le parcours du réseau d'opérateurs du programme normalisé permet d'obtenir la machine d'états finis associée au programme. En effet, ce parcours du réseau permet d'extraire les variables d'états de la machine et les variables d'entrée. Les variables d'état sont toutes les expressions du type : init (E) et last (E), où E est une expression booléenne. Par exemple, dans le réseau de la figure 10.2, init (true), et last (v1) sont des variables d'états.

La machine d'états finis $M = (Q, E, S, \delta, \lambda, q_0)$ dénotée par un programme Lustre est identifiée de la façon suivante :

- Q est l'ensemble des états de la machine, où chaque état est défini par une configuration possible des variables d'état du programme ;
- E est l'ensemble des entrées ;
- S est l'ensemble des sorties ;
- δ est la fonction de transition, définie sur chaque variable d'état par l'expression booléenne se trouvant en paramètres des opérateurs *init* et *last* ;
- λ est la fonction de sortie, définie sur chaque sortie par l'expression booléenne évaluant cette sortie ;
- q_0 est l'état initial correspondant à la configuration initiale des variables d'état défini par les opérateurs *init*.

Lorsqu'un programme Lustre contient des assertions, la machine associée à ce programme se distingue par une fonction d'assertion α. Cette fonction correspond à l'expression booléenne "assertée". La fonction d'assertion α, est définie de l'ensemble $E \times Q$ dans $\{0, 1\}$.

10.2 Méthodes d'exploration de la machine de vérification

Rappelons d'abord que pour résoudre les deux problèmes de vérification de conformité et de vérification de propriétés, nous définissons un programme Lustre avec une seule sortie Flag. Ce programme de vérification dénote une machine de vérification $M_v = (Q, E, S, \delta, \lambda, q_0)$. Sur la machine M_v la vérification sera réalisée par l'outil de preuve Lesar en évaluant la formule "Flag = 1" sur la machine M_v. Lesar utilise deux stratégies d'exploration du graphe d'états : la méthode énumérative [Glo 89] et la méthode symbolique en arrière [Rat 92].

Dans cette section, nous décrivons les deux méthodes de parcours d'un graphe d'états utilisées par l'outil Lesar. Ensuite, nous donnons un ensemble de résultats expérimentaux.

10.2.1 La méthode énumérative

Le parcours du graphe avec cette méthode [Glo 89] revient à simuler le comportement des variables d'état de la machine de vérification en énumérant toutes les entrées possibles. A chaque état, en disposant des entrées et de la valeur courante des variables d'état , il est facile de déterminer l'ensemble des valeurs futures des variables d'état, c'est à dire les états successeurs de l'état courant. Si On identifie n variables d'état, Lesar aura à parcourir au plus 2^n états.

Toutes les variables d'état étant initialisées (les opérateurs temporels pre et current doivent être initialisés), Lesar dispose d'un état initial défini par les opérateurs *init* du programme normalisé pour commencer son parcours. Chaque état est repéré par un n-uplet de variables d'état. A partir de l'état initial q_0, on étudie toutes les transitions possibles en faisant varier les entrées. On obtient alors l'ensemble des états accessibles à partir de l'état initial. Sur ces transitions, on vérifie que la sortie Flag est vraie. Le processus est alors itéré pour chaque état successeur (parcours en profondeur d'abord). Si le programme de vérification contient des assertions, la machine de vérification associée au programme se distingue par sa fonction d'assertion α. La sortie Flag doit être vérifiée pour toute transition où la fonction d'assertion retourne la valeur booléenne 1. Si la fonction α retourne la valeur booléenne 0, tous les états successeurs de l'état courant sont considérés comme des états non accessibles. Dans le cas où l'assertion est non causale, c'est-à-dire qu'il n'existe aucune variable d'entrée telle que la fonction α retourne 1, la vérification est arrêtée en signalant que l'assertion du programme est non causale.

Lesar tient à jour une table d'états où il note les états visités et les états accessibles mais non visités. Le parcours est terminé lorsque tous les états accessibles ont été visités. Le nombre d'états étant fini, l'énumération prend un temps fini. Il faut noter que Lesar ne construit pas le graphe de la machine en entier mais il garde uniquement en mémoire les états visités et les états à visiter. La génération et la vérification de la machine d'états finis sont réalisées simultanément, d'où son nom de vérification "à la volée". En général, Lesar parcourt moins de 2^n états car tous les états accessibles ne sont pas forcément atteints à partir de l'état initial. L'algorithme énumératif de vérification "à la volée" est donné dans la figure 10.3.

Avec la méthode énumérative le problème de l'explosion du modèle se pose. En effet, cette méthode consiste à énumérer l'ensemble des états accessibles de la machine de vérification. De plus, dans le cas où les assertions sont non causales la vérification est interrompue dès que la

non causalité de l'assertion est détectée. Pour résoudre ces problèmes, une approche symbolique a été implanté. Celle-ci fera l'objet du prochain paragraphe.

```
• V = états visités
• A = états accessibles
• M_{V/α} = (Q, E, S, δ, λ, q0) = machine de vérification, où α est la
                                  fonction d'assertion

debut
V = ∅
A = {q0}
   tant que A ≠ ∅ faire
                  debut
                    Soit q ∈ A
                    V = V ∪ {q}
                    A = A \{q}
                    si ∄ e ∈ E | α(e,q) = 1
                    alors retourner (assertion non causale)
                    sinon faire pour tout e de E
                            si Flag = λ(e, q) = 0 et α(e,q) = 1
                            alors retourner (SMSS incorrect)
                            sinon si α(e,q) = 1
                                  alors A = A ∪ {δ(q)} \V
                  fin
   retourner (SMSS correct)
fin
```

Figure 10.1. Algorithme énumératif de vérification "à la volée"

10.2.2 La méthode symbolique "en arrière"

Le vérificateur symbolique [Rat 92] est inspiré d'un algorithme de génération de modèle minimal de machine d'états finis [BFH 92]. Le principe de l'algorithme consiste à distinguer deux états, si et seulement si, le calcul des sorties ou des successeurs le nécessite ; d'ou son nom "algorithme dirigé par la demande". L'élément de base est une classe d'états caractérisée par une fonction booléenne. Au début, tous les états sont considérés équivalents. Au cours de la génération du graphe d'états, on est amené à diviser une classe si elle contient des états pour lesquels le calcul des sorties ou des successeurs diffèrent. Lorsque toutes les classes d'états accessibles depuis l'état initial deviennent stables, c'est-à-dire tous les états de chaque classe sont équivalents, le graphe d'états minimal est généré.

L'application de l'algorithme de génération du graphe d'état minimal au cas du programme de vérification avec une seule sortie Flag consiste à calculer les états de la machine associée au programme de vérification, à partir desquels il est possible d'atteindre la classe d'états caractérisée par "Flag = 0". La méthode de vérification est alors décomposée en deux phases :

(1) élimination des états caractérisés par "Flag = 0", puis au fur et à mesure élimination de leurs états prédécesseurs et,

(2) vérification que l'état initial n'a pas été éliminé.

Outre l'avantage d'opérer sur des classes d'états, la méthode en arrière permet de résoudre le problème des assertions non causales. En effet, un parcours en arrière permet d'éliminer les états qui mènent vers des états pour lesquels il n'existe aucune valeur d'entrée telle que la fonction d'assertion retourne la valeur vraie puis, au fur et à mesure leurs états prédécesseurs seront aussi éliminés. Ceci correspond a éliminer tous les états de transition du reconnaisseur de séquences menant uniquement vers son état puits (chapitre 6, § 6.2.1 et chapitre 9, § 9.4.1).

Le choix effectué pour l'implémentation de la méthode en arrière pour la représentation et la manipulation des fonctions booléennes caractérisant les classes d'états est la technique des graphes de décision binaires, ou BDDs (de l'anglais, Binary Decision Diagrams) [Ake 78, Bry 86]. Plus de détails sur la technique de l'implémentation de la méthode se trouvent dans [Rat 92].

10.2.3 Résultats expérimentaux

Nous donnons dans ce paragraphe les résultats expérimentaux que nous avons obtenus avec les deux stratégies de vérification implémentées dans l'outil Lesar : la méthode énumérative (Lesar 1) et la méthode symbolique en arrière (Lesar 2). Les tables 10.1 et 10.2 donnent le nombre d'états accessibles de la machine de vérification (donné par Lesar 1) et le temps CPU de vérification sur un SUN 4.

La table 10.1 illustre les résultats de la vérification de propriétés sur différents SMSS. Ces SMSS sont un module de synchronisation (Sync), un circuit à tolérance de pannes (Voteur), l'arbitre de bus présenté en chapitre 9 (§ 9.1.4), le circuit MinMax (chapitre 9, § 9.1.2) et une carte d'interface de bus conçu chez BULL [LJ 90] sur laquelle nous avons vérifié différentes propriétés (CLM1 à CLM7).

La table 10.2 donne les résultats obtenus pour la vérification de conformité de réalisations de circuits vis-à-vis de leurs spécifications fonctionnelles complètes. Ces circuits sont un vérificateur de code BCD [TP 90], un démodulateur, des compteurs, le circuit MinMax [BT 90] et l'arbitre de bus (chapitre 9, § 9.1.4).

SMSS	états accessibles	Lesar 1 (temps CPU)	Lesar 2 (temps CPU)
Sync	202	1,4s	11s
Voteur	430	9s	25s
Arbitre	3200	59s	2mn
MinMax2	5360	72s	17s
CLM1	143 390	78mn	55s
CLM2	143 390	100mn	3mn 46s
CLM3	159 477	100mn	spaceout*
CLM4	185 294	122mn	3mn 15s
CLM5	185 294	128mn	spaceout
CLM6	239 798	133mn	spaceout
CLM7	346 811	267mn	55s

Table 10.1. Vérification de propriétés

* spaceout : supérieur à 10 Mbytes

SMSS	états accessibles	Lesar 1 (temps CPU)	Lesar 2 (temps CPU)
BCD	64	0,6s	2s
Démodulateur	101	0,7s	1,7s
Compt4	153	1,2s	2mn 49s
Arbitre	841	13,5s	4,3s
MinMax2	1221	37s	40s
Compt8	4355	33s	8mn

Table 10.2. Vérification de conformité

10.2.4 Discussion

Ces résultats expérimentaux montrent que les deux méthodes sont complémentaires. D'une part, la vérification d'une propriété temporelle sur la carte CLM (CLM7) est réalisée en 267 minutes par Lesar "énumératif", alors que la même vérification est réalisée en 55 secondes par Lesar "symbolique". D'autre part, la vérification des compteurs (Compt4 et Compt8) est réalisée plus rapidement par Lesar "énumératif" que par Lesar "symbolique". Le plus souvent, la méthode symbolique est performante. En effet, la complexité de la représentation de la fonction caractéristique d'une classe d'états avec les BDDs ne dépend pas du nombre d'états. Un simple BDD peut représenter une classe contenant beaucoup d'états. Dans d'autres cas, lorsqu'un BDD complexe représente une classe avec peu d'états, la méthode énumérative est alors plus performante, c'est le cas des compteurs. Même si la méthode énumérative est linéaire par rapport au nombre d'états et de transitions, cette méthode est coûteuse en temps [SF 86]. Inversement, la méthode symbolique est coûteuse en espace mémoire. En effet, comme il a été mentionné dans [Cou 92, Rat 92] les opérateurs sur les BDDs génèrent souvent beaucoup de résultats intermédiaires qui sont consommateurs de mémoire. Toutefois, il faut indiquer que les résultats expérimentaux sont obtenus avec 10 Mbytes de mémoire. Ce qui est peu comparativement au nombre de Mbytes utilisé par d'autres systèmes de vérification. Signalons enfin, que pour obtenir un compromis temps / mémoire, une approche hybride à été proposé dans [Rat 92] fondée sur une interprétation différente des BDDs.

10.3 Conclusion

Nous avons présenté dans ce chapitre les deux méthodes utilisées par l'outil de preuve Lesar pour réaliser la vérification sur le modèle de machine d'états finis associée au programme Lustre de vérification. Nous avons ensuite montré sur un ensemble de résultats expérimentaux que ces deux méthodes sont complémentaires. Le résultat de la vérification élaboré par l'outil Lesar est binaire : soit le SMSS vérifie sa spécification, soit il ne la vérifie pas. En plus de sa réponse binaire, Lesar fournit un chemin de l'état initial vers un état où Flag vaut la valeur booléenne 0. Nous montrerons dans le chapitre 11 qu'un tel chemin nous permet de réaliser un diagnostic. Dans ce chapitre 11, nous abordons d'une manière générale le problème du diagnostic des erreurs de conception des SMSS.

Chapitre 11

Diagnostic

Nous avons vu au chapitre précédent que l'outil de vérification Lesar permet de démontrer qu'une description d'un système matériel satisfait sa spécification. Le résultat fourni par l'outil Lesar pour toute vérification est binaire : le système matériel satisfait ou ne satisfait pas sa spécification. Lorsque le SMSS ne satisfait pas sa spécification, Lesar démontre seulement l'existence d'une erreur. Un concepteur utilisant un tel outil de vérification, souhaite de plus connaître la cause de telles erreurs. Une aide à la localisation de ces erreurs sera utile.

D'un point de vue pratique, les méthodes choisies pour le diagnostic doivent être facilement utilisables et très proches des méthodes qu'un concepteur a l'habitude d'utiliser. De plus, il est souhaitable que ces méthodes soient définies dans le même environnement que celui qui est utilisé pour la description et la vérification des SMSS. Dans ce contexte, nous présentons ici trois méthodes, fondées sur le langage Lustre et l'outil de vérification Lesar, permettant la réalisation d'un diagnostic :

- l'exécution du programme de vérification,
- l'utilisation des assertions du langage Lustre,
- l'utilisation d'un analyseur logique virtuel.

11.1 Exécution du programme de vérification

En plus de sa réponse binaire, l'outil de vérification Lesar fournit un chemin de l'état initial vers un état où la sortie Flag de la machine d'observation vaut la valeur booléenne 0. Ce chemin appelé *séquence d'erreur* est constitué d'une suite de valeurs des variables d'entrée et des variables d'état. La version énumérative de Lesar arrête son parcours de graphe d'états associée à la machine d'observation dès que la sortie Flag est fausse sur une transition. Le chemin, partant de l'état initial, menant à cet état est donné (figure 11.1a). Dans la version symbolique, Lesar arrête le processus de vérification lorsque l'état initial de la machine de vérification est contenu dans la classe d'états où la sortie Flag est fausse ou dans les états prédécesseurs de cette classe. A partir de l'état initial, une séquence d'erreur de longueur minimale est générée progressivement (figure 11.1b). Signalons toutefois que la génération de cette séquence est possible grâce à la mémorisation de toutes les fonctions caractéristiques des classes d'états prédécesseurs de la classe où la sortie Flag est fausse.

Cette séquence d'erreur fournie par l'outil Lesar permet de réaliser un diagnostic en deux étapes, comme suit :

- (1) sélection de toutes les variables susceptibles d'être à l'origine de l'erreur dans la description ou la spécification du SMSS, puis spécification de celles-ci comme des variables de sortie dans le programme de vérification,

- (2) exécution du programme de vérification sur la séquence d'erreur à l'aide du compilateur du langage Lustre.

La comparaison des résultats obtenus avec le comportement attendu de l'ensemble des variables sélectionnées permet d'aider à localiser la source de l'erreur. La localisation de l'erreur est facilitée par une interface permettant la visualisation des sorties retournées par le programme de vérification, durant son exécution, sous forme d'un chronogramme.

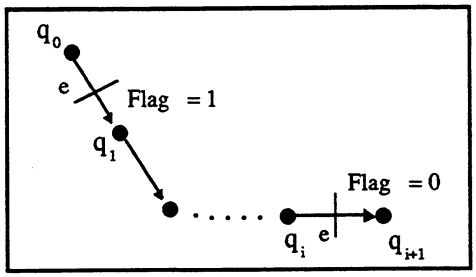


Figure 11.1a. Séquence d'erreur fournie par Lesar "énumératif"

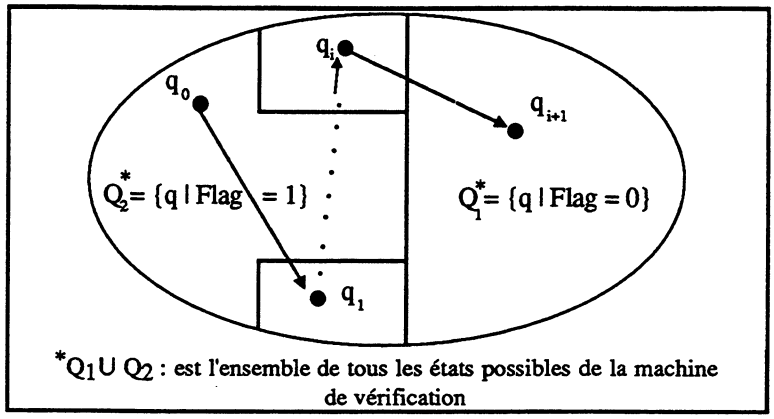


Figure 11.1b. Construction de la séquence d'erreur par Lesar "symbolique"

11.2 Technique des assertions

Au moyen d'une assertion, on peut spécifier un comportement restreint de l'environnement. Ceci permet de définir des sous-ensembles de $(E \times S)^*$, où E et S sont respectivement les ensembles d'entrée et de sortie du SMSS. A chaque sous-ensemble, les blocs fonctionnels correspondants sont activés. En particulier, pour les SMSS ayant une partie contrôle et une partie opérative distinctes, cette technique permet la localisation de l'erreur dans le bloc responsable de la partie opérative.

Exemple. Pour illustrer l'utilisation de cette technique, nous donnons dans la figure 11.2 une réalisation possible du processeur du signal MinMax présenté en chapitre 9 (§ 9.1.2). Rappelons que ce processeur reçoit en entrée des données "In" ainsi que trois signaux de contrôle : Reset, Enable et Clear. Il fournit en sortie des données "Out" dépendant des entrées et des signaux de contrôle. La réalisation que nous donnons du MinMax a une structure pipeline à deux étages commandés par deux horloges : Ck_1 et Ck_2.

La restriction des signaux de contrôle : Reset, Enable et Clear au langage $(0\ 1\ 0)^*$ permet d'inactiver le module *last_value*. Lorsque cette restriction est définie sur le langage $((1 + 0)\ 0\ 0)^*$ le module *mean_value* est inactivé. Cette restriction à ces langages des signaux de contrôle est définie au moyen des assertions comme suit :

- `assert (not Reset and Enable and not Clear)` spécifie le langage $(0\ 1\ 0)^*$ et,
- `assert (not Enable and not Clear)` spécifie le langage $((1 + 0)\ 0\ 0)^*$

Lors de la vérification, le calcul des sorties ne tient pas compte du module inactivé. Ceci peut permettre de localiser une erreur sur un module. Les figures 11.3a et 11.3b illustrent

l'utilisation de cette technique sur l'exemple du MinMax. Notons que cette technique nous a permis de localiser une erreur dans un module comparateur "COMP_max".

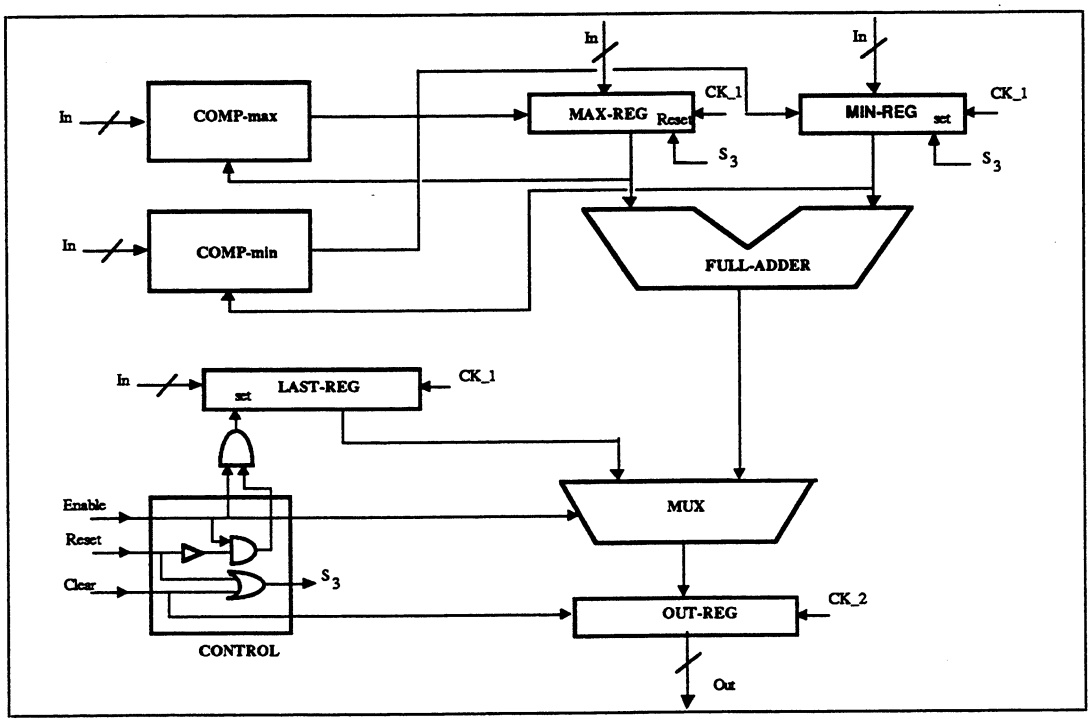


Figure 11.2. Une réalisation du processeur MinMax

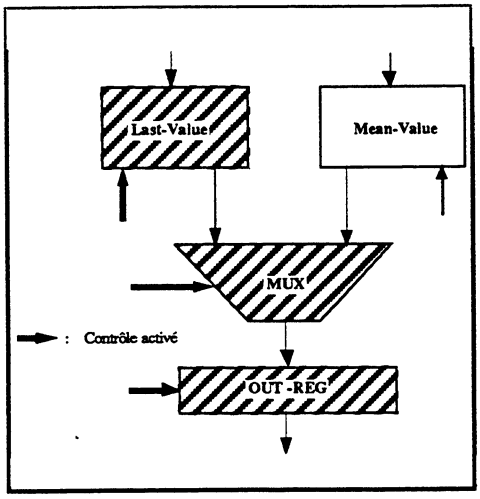


Figure 11.3a.

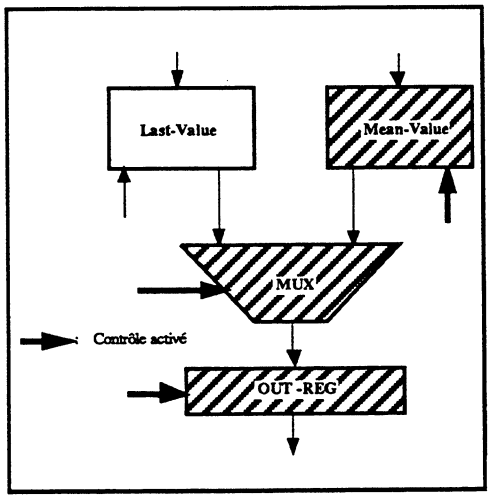


Figure 11.3b.

Figure 11.3. Utilisation de la technique des assertions pour le diagnostic du MinMax

11.3 Un analyseur logique virtuel

Cette méthode est inspirée d'une des méthodes utilisée par les concepteurs de systèmes matériels pour l'identification et la localisation des erreurs de conception [ZL 79]. Il s'agit de l'utilisation d'un analyseur logique permettant la visualisation des signaux du système matériel sous forme de traces. Un tel analyseur permet au concepteur de suivre l'évolution du système de manière à détecter les anomalies de fonctionnement.

La méthode de diagnostic que nous proposons ici consiste à vérifier exhaustivement sur la description du système matériel des propriétés temporelles de manière à localiser la source d'erreur. Le choix des propriétés temporelles est réalisé par le concepteur. En effet, le type de comportement à vérifier aidant à déterminer l'origine de l'anomalie nécessitera une intervention humaine. La spécification des propriétés à vérifier est facilitée par l'utilisation d'une bibliothèque d'opérateurs temporels donnée en annexe. Rappelons qu'une propriété temporelle s'exprime à l'aide de paramètres temporelles d'intervalle d'observation, de contexte et de réaction (chapitre 3).

En cas d'échec de la vérification d'une propriété choisie pour le diagnostic, on passe à la deuxième étape qui consiste à faire varier les paramètres temporels de la propriété. Par exemple, la variation de l'intervalle d'observation peut permettre de déterminer l'intervalle temporel dans lequel le comportement du circuit ne vérifie pas la propriété. Ceci peut aider à localiser le bloc fonctionnel du SMSS responsable de l'anomalie. A chaque étape, l'exécution du programme de vérification sur la séquence d'erreur se fera à l'aide du compilateur Lustre. Une interface permet la visualisation des traces d'exécution. Ce fonctionnement correspond à celui de l'analyseur logique classique.

11.4 Conclusion

Nous avons proposé, dans ce chapitre, des méthodes visant à aider le concepteur à réaliser un diagnostic d'une erreur détectée lors de la vérification d'un SMSS. Ces méthodes sont définies dans le même environnement de description et de vérifications des SMSS. Elles sont fondées sur le langage Lustre et sur l'outil de vérification associé Lesar. Ces méthodes offrent seulement une assistance à la localisation des erreurs de conception. Une intervention humaine reste le facteur déterminant permettant une localisation précise de l'anomalie. L'idéal serait un outil de preuve permettant de démontrer et de localiser l'erreur. Une première amélioration envisageable serait de déterminer le chemin physique associé à la séquence d'erreur. En effet, le modèle sur lequel s'effectue la vérification est déterminé à partir du programme réécrit au format EC qui perd la dénomination des variables du programme source.

Conclusion et perspectives

L'approche de vérification fonctionnelle des systèmes matériels séquentiels synchrones que nous avons développée dans ce document permet l'expression et la résolution des problèmes de vérification qu'un concepteur a à affronter d'une manière simple. Cette approche est fondée sur la vérification d'une formule sur un modèle de machine d'états finis, dans laquelle la description d'un système matériel à un niveau d'abstraction donné, sa spécification et son environnement ont pour modèle une machine d'états finis. L'expression du problème de vérification se ramène à définir une machine d'observation ayant une unique sortie. Il s'agira alors de vérifier que cette machine d'observation génère le langage 1^*.

Nous avons ensuite présenté l'application du langage Lustre et de son outil de vérification associé Lesar développés initialement pour la programmation et la vérification des systèmes réactifs. Nous avons montré que le langage Lustre permet de décrire le programme de vérification dénotant la machine d'observation sur laquelle la vérification sera réalisée par l'outil de preuve Lesar. Lesar utilise deux stratégies de vérification pour explorer le graphe d'états d'une machine : une méthode énumérative et une méthode symbolique. Nous avons montré que ces deux méthodes sont complémentaires. En effet, même si la méthode énumérative est coûteuse en temps, la vérification avec la méthode symbolique peut ne pas aboutir du fait qu'elle est coûteuse en espace mémoire. Notons ici que l'obtention du résultat d'une vérification pour un concepteur est importante, même si celle-ci est coûteuse en temps.

Cette étude nous a permis de mettre en relief un réel environnement de vérification fonctionnelle des systèmes matériels séquentiels synchrones pouvant être intégré dans une chaîne de CAO (figure 12). Dans cet environnement, le langage de description des systèmes matériels (aux différents niveaux de conception) et de spécification des fonctionnalités à vérifier sera un langage du type Lustre ayant une sémantique formelle en termes de machine d'états finis. Dans

le but de faciliter la tâche de description du concepteur, il serait intéressant que le langage utilisé pour la description et la vérification des systèmes matériels ait une notion de décomposition hiérarchique permettant la définition d'une bibliothèque de systèmes et d'opérateur temporels de base. Tout problème de vérification se ramène ici à définir un programme de vérification ayant une unique sortie. Ce programme dénote une machine d'états finis d'observation sur laquelle la vérification sera réalisée. Cette vérification consistera à explorer le graphe d'états de la machine d'observation et à vérifier que la machine génère le langage 1^* sur son unique sortie. Plusieurs stratégies d'exploration d'un graphe d'états peuvent alors être intégrées dans un même outil : énumérative [Glo 89, Hol 89], pseudo-énumérative [Rat 92], symbolique en avant [Cou 91, BCM 90, GDN 90], symbolique en arrière [Rat 92]. Pour toute vérification, on pourra lancer en parallèle les différentes options : la première stratégie donnant le résultat arrêtera le processus de vérification. Le problème du diagnostic sera aussi pris en compte dans l'environnement de vérification. En effet, un concepteur souhaite, en plus de la démonstration de l'existence d'une erreur, une aide à la localisation de la source d'erreur. Les différentes méthodes et améliorations que nous avons présentées et discutées au chapitre 11 pourront être intégrées dans ce cadre de vérification.

Les concepteurs de systèmes matériels sont habitués à spécifier le fonctionnement de leurs systèmes par un ensemble de chronogrammes. De ce fait, il serait intéressant d'intégrer à l'environnement de vérification un outil permettant des spécifications sous forme de chronogrammes. Dans ce but, nous avons proposé dans [BT 92, Lau 90], une méthode permettant le passage systématique d'un chronogramme vers une propriété temporelle. Notons ici que des travaux ont été menés dans ce sens aux laboratoires Fujitsu [Fuj 90] ainsi qu'au centre de recherche BULL [AL 91].

Finalement, il serait intéressant d'appliquer l'approche que nous avons développé dans cet ouvrage à un environnement VHDL, du fait que le langage VHDL [VHDL 87] est actuellement considéré comme une norme industrielle. Ce langage a été conçu pour la description et la simulation des systèmes matériels. Mais, il est difficile de lier l'ensemble du langage VHDL à un modèle qu'on peut manipuler formellement. Toutefois, des sous-ensembles VHDL permettant des manipulations formelles (ayant une sémantique en termes de machines d'états finis) ont été définis par différentes équipes de recherches [BPS 92, DBJ 92]. Notons ici qu'une étude réalisée dans notre équipe de recherche avec la collaboration du centre de recherche de BULL [CE 91] a permis de définir une méthodologie de preuve de propriétés dans un environnement VHDL.

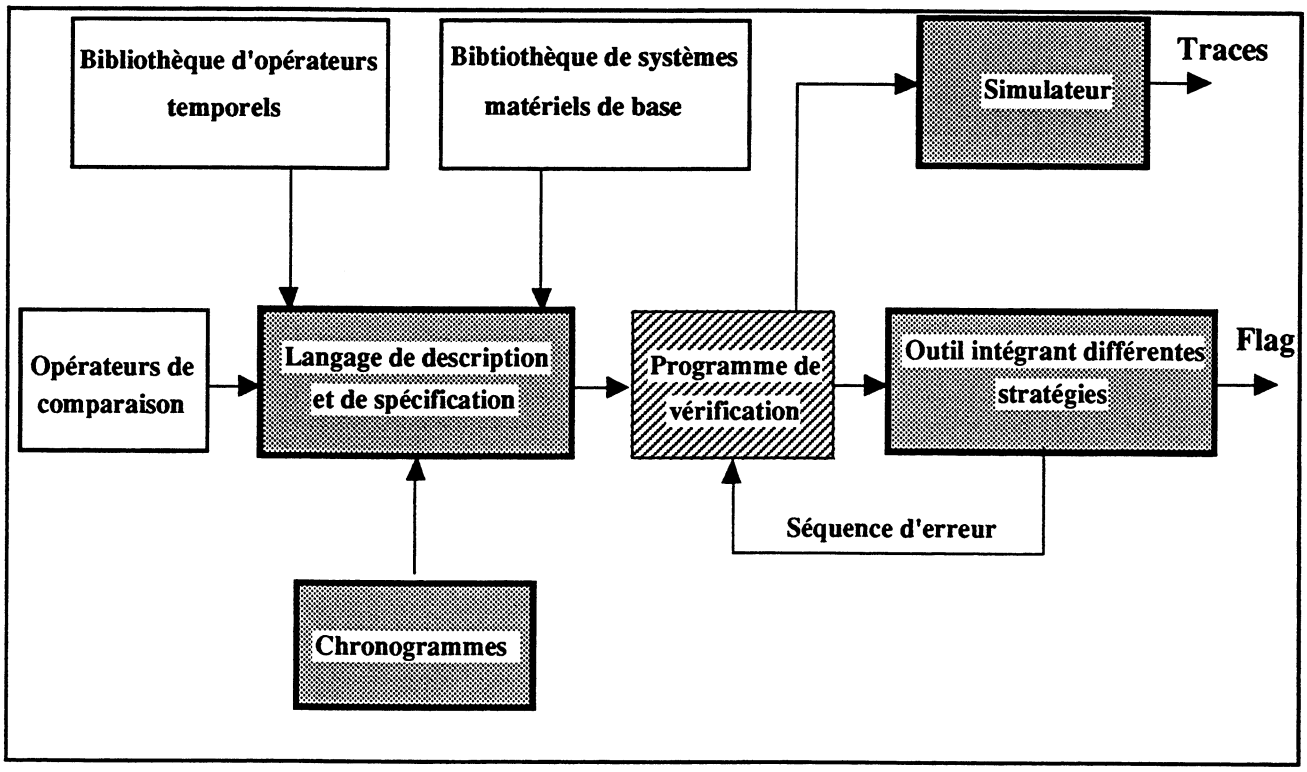


Figure 12. Un environnement de vérification fonctionnelle des systèmes matériels synchrones

Bibliographie

[AEF 90] E. Audureau, P. Enjalbert, L. Farinas del Cerro. *Logique Temporelle : Sémantique et Validation de Programmes Parallèles.* Edition MASSON, Paris, 1990.

[Ake 78] S. B. Akers. *Binary Decision Diagrams.* In IEEE Transaction on Computer, C-27(6), 1978.

[AL 91] C. Antoine, B. Le Goff. *Timing Diagrams for Writing and Checking Logical and Behavioral Properties of Integrated Systems.* Correct Hardware Design Methodologies, P. Camurati and P. Prinetto Editors, Elsevier Science Publishers, 1991.

[Bar et al 85] M. R. Barbacci et al. *ADA as a Hardware Description Language : an Initial Report.* In 7th International Conference on Computer Hardware Description Languages (CHDL), Tokyo, August 1985.

[BC 86] M. C. Browne, E. M. Clarke. *Automatic Circuit Verification Using Temporal Logic: Two New Examples.* Formal Aspects of VLSI Design, G. J. Milne and P. A Subrahmanyam Editors, Elsevier Science Publishers, 1986.

[BCM 90] S. Burch, E. M. Clarke, K. L. McMillan *Symbolic Model Checking: 10^{20} States and Beyond.* In Proc. of LICS, 1990.

[Ber 89] G. Berry. *Real Time Programming: Special Purpose or General Purpose Languages.* In IFIP World Computer Congress, San Fransisco, 1989.

[BFH 90] A. Bouajjani, J. C. Fernandez, N. Halbwachs. *On the Verification of Safety Properties*. Technical Report SPECTRE, IMAG, Grenoble, France, March 1990.

[BFH 92] A. Bouajjani, J. C. Fernandez, N. Halbwachs. *Minimal State Graph Generation*. To appear in Science of Computer Programming, 1992.

[BM 79] R. S. Boyer, J. S. Moore. *A Computational Logic*. Academic Press, 1979.

[BMP 83] M. Ben-Ari, Z. Mana, A. Pnueli. *The Temporal Logic of Branching Time*. Acta Informatica, 20 (3), 1983.

[Boc 82] G. V. Bochmann. *Hardware Specification with Temporal Logic: An Example*. IEEE Transactions on Computer, vol.c-31, n°3, March 1982.

[Boo 67] T. L. Booth. *Sequential machines and Automata Theory*. J. Wiley & Sons Editors, New York, 1988.

[BPPC 89] D. Borrione, J. L. Paillet, L. Pierre, H. Collavizza. *Modélisation Fonctionnelle et Preuve de Circuits Digitaux*. Techniques et Science Informatique, vol 8, N° 6, 1989.

[BPS 92] D. Borionne, L. Pierre, A. Salem. *Formal Verification of VHDL Descriptions in the PREVEAL Environment*. In IEEE Design & Test of Computer, vol 9, N° 3, Juin 1992.

[BRSW 87] R. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, A. Wang. *MIS: Multiple Level Logic Optimization System*. In IEEE Transaction on Computer-Aided-Design, Novembre 1987.

[Bry 86] R. E. Bryant. *Graphed Based Algorithms for Boolean Functions Manipulation*. In IEEE Transaction on Computer, C-35(8), August 1986.

[BT 90] B. Berkane, G. Thuau. *Application du Langage Lustre à la Vérification de Conformité. : un Recueil d'Exemples*. Rapport interne, Juin 1990.

[BT 92] B. Berkane, G. Thuau. *Extraction des Propriétés Temporelles d'un Chronogramme*. Rapport interne, Mars 1992.

[CBM 89] O. Coudert, C. Berthet, J. C. Madre. *Verification of Sequential Machines Using Boolean Functional Vectors.* In Proc. of the IFIP International Workshop, Applied Formal Methods for Correct VLSI Design, Leuven, Belgium, November 1989.

[CCBM 91] F. Corela, R. Camposano, R. Bergamashi, M. Mayer. *Verification of Synchronous Sequential Circuits Obtained from Algorithmic Specification* . In Proc. of the International Workshop on Formal Methods in VLSI Design, Miami, USA, January 1991.

[CDCT 92] C. Courcoubetis, D. Dill, M. Chatzaki, P. Tzounakis. *Verification with Real Time COSPAN.* In Proc. of the Fourth Workshop on Computer-Aided Verification, Montreal, CANADA, June 1992.

[CE 91] P. Cavenel. X. Epineuse *Validation de Propriétés de Circuits Digitaux dans un Environnement VHDL.* Projet 3ème année ENSERG-ENSIMAG, Juin 1991.

[CES 83] E.M. Clarke, E. A. Emerson, A. P. Sistla. *Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specification: a Practical Approach.* Tenth ACM Symposium on Principles of Programming Languages, Austin, USA, 1983.

[CGPS 91] P. Camurati, M. Gilli, P. Prinetto, M. Sonza Reorda. *The Product Machine and Implicit Enumeration to Prove FSMs Correct.* In Correct Hardware Design Methodologies, P. Camurati and P. Prinetto Editors, Elsevier Science Publishers, 1991.

[CM 90a] O. Coudert, J. C. Madre. *Verifying Temporal Properties of Sequential Circuits without Building their State Diagrams.* In Computer-Aided Verification, E.M. Clarke and R. P. Kurshan Editors, DIMACS series, June 1990.

[CM 90b] O. Coudert, J. C. Madre. *A Unified Framework for the Formal Verification of Sequential Circuits.* In Proc. of ICCAD, Santa Clara, USA, November 1990.

[Coh 87] A. Cohn. *A Proof of Correctness of VIPER Microprocessor: the First Level.* In VLSI Specification, Verification, and Synthesis, G. Birtwistle and P. A Subrahmanyam Editors, Elsevier Science Publishers, 1987.

[Cou 91] O. Coudert. *SIAM : une Boite à Outils pour la Preuve Formelle de Systèmes séquentiels.* Thèse, Ecole Nationale Supérieure des Télécommunications, Octobre 1991.

[CP 88] P. Camurati, P. Prinetto. *Formal Verification of Hardware Correctness: Introduction and Survey of Current Research.* In IEEE Computer, vol 21, N° 27,, July 1988.

[CPHP 87] P. Caspi, D. Pilaud, N. Halbwachs, J. Plaice. *Lustre : a Declarative Language for Programming Synchronous Systems.* In 14th ACM Symposium on Principles of Programming languages, January 1987.

[Cra 85] M. Crastes. *Spécification et Simulation Fonctionnelles de Circuits Complexes.* Thèse, Institut Polytechnique de Grenoble, Novembre 1985.

[CSSB 92] M. Chiodo, T. R. Shiple, A. Sangiovanni-Vincentelli, K. Brayton. *Automatic Reduction in CTL Compositional Model Checking.* In Proc. of the Fourth Workshop on Computer-Aided Verification, Montreal, CANADA, June 1992.

[Dar 79] J. A. Darringer. *The Application of Program Verification Technique to Hardware Verification.* In Proc. of the 16th ACM/IEEE Design Automation Conference (DAC), 1979.

[DBJ 92] A. Debreil, C. Berthet, A. Jerraya. *Symbolic Computation of Hierarchical and Interconnected FSMs.* In J. Mermet (ed.), VHDL for Simulation, Synthesis and Formal Proofs of Hardware, Kluwer Academic Publishers, 1992.

[DM 92] P. Déverchère, J. C. Madre. *Functional Abstraction and Formal Proof of Digital Circuits.* In Proc. of the European Conference on Design Automation, Brussels, Belgium, March 1992.

[FF 89] M. Fujita, H. Fujisawa. *Specification, Verification and Synthesis of Control Circuits with Propositional Temporal Logic.* In Proc. of the 9th IFIP on Computer Hardware Description Languages, Washington DC, USA, June 1989.

[Fuj 90] M. Fujita. *Logic Verification with Binary Decision Diagrams.* Séminaire, Grenoble, Mai 1990.

[GDN 90] A. Ghosh, S. Devadas, A. R. Newton. *Verification of Interacting Sequential Circuits.* In 27th ACM/IEEE Design Automaton Conference (DAC), Orlando, USA, June 1990.

[GK 91] C. H. Golaszewski, R.P. Kurshan. *Task_Driven Supervisory Control of Discrete Event Systems.* In DIMACS Series in Discrete Mathematics and Theorical computer Science, Vol. 3, 1991.

[Glo 89] A-C. Glory. *Vérification de Propriétés de Programmes Flots de Données Synchrones.* Thèse, Université Josephe Fourier de Grenoble, Décembre 1989.

[Gor 83] M. Gordon. *LCF-LSM.* Technical Report, University of Cambrige, Computer Laboratory, N° 41, 1983.

[Gor 86] M. Gordon. *Why Higher Order Logic is a Good Formalism for specifying and Verifying hardware.* In Formal Aspects of VLSI Design, G. J. Milne and P. A Subrahmanyam Editors, Elsevier Science Publishers, 1986.

[Gor 87] M. Gordon. *HOL a Proof Generating System for Higher Order Logic.* In VLSI Specification, Verification, and Synthesis, G. Birtwistle and P. A Subrahmanyam Editors, Elsevier Science Publishers, 1987.

[HCRP 91] N. Halbwachs, P. Caspi, P. Raymond, D. Pilaud. *The Synchronous Dataflow Programming Language LUSTRE.* In IEEE Special Issue on Synchronous Languages, Septembre 1991.

[HK 89] Z. Har'El, R. P. Kurshan. *Automatic Verification of Finite-State Concurent Systems.* In Proc. of Workshop on Automatic Verification Methods for Finite-State-Systems, Grenoble, June 1989.

[HLR 92] N. Halbwachs, F. Lagnier, C. Ratel. *An Experience in Proving Regular Network of Processes by Modular Model Checking.* In IEEE transactions on Software Engineering, Special Issue on the Specification and Analysis of Real-Time Systems, Septembre 1992.

[Hol 89] G. J. Holtzmann. *Algorithms for Automated Protocols Validation.* In Proc. of Workshop on Automatic Verification Methods for Finite_State Systems, Grenoble, June 1989.

[Hun 86] W. A. Hunt. *FM8501: A Verified Microprocessor.* IFIP Workshop, From HDL Description to Guaranteed Correct Circuit Designs, North-Holland Publishing, September, 1986.

[Kah 74] G. Kahn. *The Semantics of a Simple Language for Parallel Programming.* In IFIP 74, North Holland, 1974.

[Koh 70] Z. Kohavi. *Switching and Finite Automata Theory.* Computer Science Series, Mc Graw Hill, New York, USA, 1970.

[Kre 78] E. Kreyszig. *Advanced Engineering Mathematics* . J. Wiley & Sons Editors, New York, 1978.

[Kur 90] R. P. Kurshan. *Analysis of Discrete Event Coordination.* In Lecture Notes in Computer Science, Spring-Verlag, 1990.

[Lau 90] E. Laurent. *Du Chronogramme à la Propriété Temporelle.* Rapport de DEA, Institut National Polytechnique de Grenoble, Septembre 1990.

[LCGP 89] J. C. Laprie. B. Courtois, M. C. Gaudel, D. Powell. *Sûreté de Fonctionnement des Systèmes Infomatiques.* Edition Bordas, Paris, 1989.

[Leu 90] Proceeding. of the IFIP International Workshop, Applied Formal Methods for Correct VLSI Design, Leuven, Belgium, November 1990.

[Lev 91] B. Levy. *An Overview of Hardware Verification Using the State Delta Verification System* . In Proc. of the International Workshop on Formal Methods in VLSI Design, Miami, USA, January 1991.

[LJ 90] E. Laurent. E. Jolly. *Méthodes de Validation de Circuits : Application à une Interface de Bus.* Projet 3ème année, Ecole Nationale Supérieure d'Electronique de Radioélectricité de Grenoble, Juin 1990.

[Mad 90] J.C. Madre. *PRIAM : un Outil de Vérification Formelle de Circuits Intégrés Digitaux.* Thèse, Ecole Nationale Supérieure des Télécommunications, Juin 1990.

[Mah 90] M. Al Mahrouz. *Génération de Test Fonctionnel de Circuits Digitaux Décrits avec un Langage Declaratif : Lustre.* Thèse, Institut Polytechnique de Grenoble, Juillet,1990.

[Mar 86] S. Marine. *IRENE : un Langage pour la Description, Simulation et Synthèse Automatique du Matériel VLSI.* Thèse, Institut Polytechnique de Grenoble, Février 1986.

[MC 80] C. Mead, L.Conway. *Introduction to VLSI Systems.* Reading, Mass. Addison-Wesley, 1980.

[Mil 83] G. Milne. *Circal: A Calculus for Circuit Description.* Integration, the VLSI Jounal, 1(2, 3), 1983.

[Mou 92] L. Mounier. *Méthodes de Vérification de Spécifications Comportementales : Eude et Mise en Oeuvre.* Thèse, Institut Polytechnique de Grenoble, 1992.

[Nac 86] N.Nachtmann. *68000 : Anatomie d'un Super-Microprocesseur.* Publitronic, Septembre 1986.

[Pai 86] J. L. Paillet. *A Fonctional Model for Descriptions and Specifications of Digital Devices.* In Proc. of the IFIP International Working Conference, From HDL Descriptions to Guaranteed Correct Circuit Designs, Grenoble, September 1986.

[Par 81] D. Park. *Concurrency and Automata on Infinite Sequences.* In Peter Deussen Editor, Theoretical Computer Science. Volume 104 of Lecture Notes in Computer Science, Springer Verlag, Berlin, March 1981.

[Pie 90] L. Pierre. *Représentation Fonctionnelle et Preuve Automatisée de Circuits Digitaux.* Thèse, Université de Provence, 1990.

[Pla 88] J. A. Plaice. *Sémantique et Compilation de Lustre, un Langage Déclaratif Synchrone.* Thèse, Institut National Polytechnique de Grenoble, Mai 1988.

[QS 81] J. P. Queille, J. Sifakis. *Specification and Verification of Concurrent Systems in CESAR.* In Proc. of the Fifth International Symposium in Programming, 1981.

[Rat 92] C. Ratel. *Définition et Réalisation d'un outil de Vérification Formelle de Programmes Lustre : le Système Lesar.* Thèse, Université Joseph Fourier de Grenoble, Juillet 1992.

[Ray 91] P. Raymond. *Compilation Efficace d'un Langage Déclaratif Synchrone : le Générateur de Code Lustre V3.* Thèse, Institut Polytechnique de Grenoble, Novembre 1991.

[RH 91a] F. Rocheteau, N. Halbwachs. *Pollux, a Lustre Based hardware Design Environment.* Conference on Algorithms and Parallel VLSI Architectures, Deplasmolen, Château de Bonas, France, Juin 1991.

[RH 91b] F. Rocheteau, N. Halbwachs. *Implementing Reactive Programs on Circuits, a Hardware Implementation of Lustre.* Workshop on Real-Time: Theory and Practice, Deplasmolen, Netherlands, Juin 1991.

[Roc 92] F. Rocheteau. *Extension du Langage Lustre et Application à la Conception de Circuits : le langage Lustre V4 et le Système Pollux.* Thèse, Institut Polytechnique de Grenoble, Juin 1992.

[SF 86] K. J. Supowit, S. J. Friedman. *A New Method for Verifying Sequential Circuits.* In Proc. of the 23rd Design Automation Conference (DAC), 1986.

[She 84] M. Sheeran. *muFP, a Language for VLSI Design.* ACM Symposium on Lisp and Fonctional Programming, Austin, USA, 1984.

[Sze 68] H. Sze-Tsen. *Mathematical Theory of Switching Circuits and Automata.* University of California Press, Berkeley and Los Angeles, USA, 1968.

[TB 92] G. Thuau, B. Berkane. *Using the Language Lustre for Sequential Circuit Verification.* In Designing Correct Circuits, J. Staunstrup and R. Sharp Editors, Elsevier Science Publishers, 1992.

[TP 90] G. Thuau, D. Pilaud. *Using the Declarative Language Lustre for Circuit Verification.* In Designing Correct Circuits, G. Jones and M. Sheeran Editors, Springer-Verlag Publisher, 1990.

[VAD 91] F. Van Aelten, J. Allen, S. Devadas. *Verification of Relations between Synchronous Machines.* In Proc. of ICCAD, Santa Clara, USA, November 1991.

[VC 89] D. Verkest, L. Claesen. *Special Benchmark Session for Tautology Checking.* In Proc. of the IFIP International Workshop, Applied Formal Methods for Correct VLSI Design, Leuven, Belgium, November 1989.

[VHDL 87] ———, *IEEE Standart VHDL lanaguage Manual.* IEEE Standart 1076, 1987.

[ZL 79] R. Zaks, A. Lesea. *Techniques d'Interface aux Microprocesseurs.* Sybex Inc. Editors, Berkeley, 1979.

Annexe

Bibliothèque d'opérateurs temporels en LUSTRE

Evénements définis

```
node RISING_EDGE (Init, A: bool) returns (Rising_edge: bool);
let
  Rising_edge = Init -> A and pre (not A));
tel.

node FALLING_EDGE (Init, A: bool) returns (Falling_edge: bool);
let
  Falling_edge = Init -> not (A) and pre (A));
tel.

node CYCLIC SIGNAL_N/M (static n: int; static m: int; a: bool)
                              returns (Cyclic signal_n/m: bool);
var B: bool^n, C: bool^m;
let
  B = false -> pre (a) | false^n-1 -> pre B[0 .. n-2];
  C = false -> pre (B[n-1]) | false^n-1 -> pre C[0 .. m-2];
  Cyclic signal_n/m = COMP (n, B, true^n) and COMP (m, C, false^m);
tel.

node COMP (static n:int; A, B: bool^n) returns (Flag: bool);
let
  Flag = with (n = 1) then (A[0] = B[0])
           else COMP (n/2, A[0 ..(n/2)-1], B[0 ..(n/2)-1]) and
                    COMP (n+1/2, A[n/2 ..n-1], B[n/2 ..n-1]);
tel.
```

Intervalles d'observation

```
node AFTER (A: bool) returns (After: bool);
let
  After = A -> A or pre (After));
tel.

node BEFORE (A: bool) returns (Before: bool);
let
  Before = not (After (A));
tel.

node FROM_TO_ (A, B: bool) returns (From_to: bool);
var Interval: bool;
let
  Interval = if A then true
                  else if B then false
                                 else (false -> pre interval);
  From_to_  = AFTER (RISING_EDGE (false, Interval)) and
              BEFORE (FALLING_EDGE (false, Interval));
tel.
```

Réactions de base

```
node ONCE_SINCE_ (A, B: bool) returns (Once_since: bool);
let
  Once_since = if B then A
                   else (A or pre (Once_since));
tel.

node ONLYONCE_SINCE_ (A, B: bool) returns (Onlyonce_since: bool);
let
  Onlyonce_since = if B then A
                        else (A and (false -> not pre (Onlyonce_since)) or
                             ((not A) and (true -> pre (Onlyonce_since));
tel.

node ALWAYS_SINCE_ (A, B: bool) returns (Always_since: bool);
let
  Always_since = not (ONCE_SINCE (not (A), B));
tel.

node NEVER_SINCE_ (A, B: bool) returns (Never_since: bool);
let
  Never_since  = ALWAYS_SINCE_ (not (A), B) ;
tel.
```

Opérateurs temporels de base

```
node ALWAYS (A: bool) returns (Always: bool);
let
  Always = A -> A and pre(Always);
tel.

node NEVER (A: bool) returns (Never: bool);
let
  Never = ALWAYS (not A);
tel.

node ONCE_FROM_TO_ (C, A, B: bool) returns (Once_from_to: bool);
let
  Once_from_to = not (B) or ONCE_SINCE_ (C, A);
tel.

node ALWAYS_FROM_TO_ (C, A, B: bool) returns (Always_from_to: bool);
let
  Always_from_to = not (B) or ALWAYS_SINCE_ (C, A);
tel.

node NEVER_FROM_TO_ (C, A, B: bool) returns (Never_from_to: bool);
let
  Never_from_to  = ALWAYS_FROM_TO_ (not C, A, B);
tel.

node ONCE_BETWEEN_ANDLAST_ (C, A: bool) returns
                                        (Once_between_andlast: bool);
let
  Once_between_andlast = not (A) or ONCE_SINCE_ (C, (True -> pre(A)));
tel.
```

Opérateurs temporels de base

```
node ONLYONCE_FROM_TO_ (C, A, B: bool) returns
                                    (Onlyonce_from_to: bool);
let
  Onlyonce_from_to = not (B) or ONLY ONCE_SINCE (C, A);
tel.

node NEVER_BEFORE_ (C, A: bool) returns (Never_before: bool);
let
  Never_before = not (BEFORE (A)) or  NEVER (C);
tel.
```

Résumé de thèse : La validation fonctionnelle d'un système matériel consiste à vérifier le système vis-à-vis de son fonctionnement attendu. Il existe deux façons de spécifier ce fonctionnement attendu. D'une part, la spécification peut être donnée sous forme d'une description fonctionnelle complète. D'autre part, l'expression de cette spécification peut être donnée sous forme d'un ensemble de propriétés temporelles critiques. Ces deux façons de spécifier les systèmes matériels ont donné lieu à deux problèmes de vérification. Notre domaine d'étude concerne les *systèmes matériels numériques séquentiels synchrones*. Ce document développe une approche de vérification unifiée, fondée sur le modèle de machines d'états finis, pour résoudre les deux problèmes de vérification sur ces systèmes. Dans cette approche, tout problème de vérification se ramène à définir une machine d'états finis sur laquelle la vérification sera réalisée. L'application du langage Lustre et de l'outil de vérification Lesar associé a été étudiée dans le but de valider cette approche. Dans cette application, la résolution des deux problèmes de vérification se ramène à définir un programme Lustre ayant une seule sortie. La vérification consiste à vérifier que cette sortie est la constante booléenne 1. Cette vérification est réalisée automatiquement par l'outil de vérification Lesar.

Mots-clés : systèmes matériels numériques séquentiels synchrones, machines d'états finis, vérification fonctionnelle, propriétés temporelles, environnement d'un système matériel, langage de description du matériel, outil de vérification, diagnostic.

Abstract: The functional verification of hardware design consists in checking that a given system operates as specified. There are mainly two kinds of high level specifications used in hardware design: (1) a complete specification given by a high abstraction level and (2) a partial high level specification made up of a set of properties that the system must satisfy. These two different kinds of hardware specifications give rise to two different verification problems. This thesis presents a unified framework to handle the two verification problems on synchronous sequential digital hardware systems. Each verification problem comes down to defining a verification machine with a single output Flag and to verifying that Flag is always true. The application of the langage Lustre and the verification tool Lesar is studied. Whithin this application, the designers can face their verification problems by defining a Lustre program with a single output flow. The verification comes down to checking that this output flow is the constant true. This verification is automatically performed by the verification tool Lesar.

Keywords: synchronous sequential digital hardware systems, finite state machines, functional verification, temporal properties, hardware system environment, hardware description langage, verification tool, diagnosis.