Nancy-Université
Université Henri Poincaré

AVERTISSEMENT

Ce document numérisé est le fruit d'un long travail approuvé par le jury de soutenance et mis à disposition de l'ensemble de la communauté universitaire élargie.
Il est soumis à la propriété intellectuelle de l'auteur au même titre que sa version papier. Ceci implique une obligation de citation et de référencement lors de l'utilisation de ce document.
D'autre part, toute contrefaçon, plagiat, reproduction illicite encourt une poursuite pénale.
La Bibliothèque a pris soin d'adresser un courrier à l'auteur dans lequel elle l'informe de la mise en ligne de son travail. Celui-ci peut en suspendre la diffusion en prenant contact avec notre service.

➢ Contact SCD Nancy 1 : theses.sciences@scd.uhp-nancy.fr

LIENS

Code de la Propriété Intellectuelle. articles L 122. 4
Code de la Propriété Intellectuelle. articles L 335.2- L 335.10
http://www.cfcopies.com/V2/leg/leg_droi.php
http://www.culture.gouv.fr/culture/infos-pratiques/droits/protection.htm

Département de formation doctorale en informatique
UFR STMIA

École doctorale IAEM Lorraine

Modélisation et développement formel de circuits électroniques

THÈSE

présentée et soutenue publiquement le 29 novembre 2006

pour l'obtention du

Doctorat de l'université Henri Poincaré – Nancy 1

(spécialité informatique)

par

Yann Zimmermann

Composition du jury

Président :	Monique Grandbastien	Professeur, Université Henri Poincaré Nancy 1
Rapporteurs :	Jacques Julliand	Professeur, Université Franche-Comté
	Laurence Pierre	Professeur, Université de Nice
Examinateurs :	Dominique Cansell	Maître de conférence, HDR, Université de Metz
	Stefan Hallerstede	Docteur, ETH Zurich
	Dominique Mery	Professeur, Université Henri Poincaré Nancy 1

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

Mis en page avec la classe thloria.

Résumé

Les systèmes électroniques sont de plus en plus complexes et les exigences de fiabilité vis-à-vis de ces systèmes sont de plus en plus importantes. Le défi est de continuer à développer des systèmes de plus en plus complexes tout en assurant la correction de ceux-ci. Les méthodes de correction par le test sont aujourd'hui dépassées par la complexité des systèmes. Nous proposons dans cette thèse d'utiliser la preuve et le raffinement pour assurer la correction d'un système. La correction par la preuve à l'avantage de ne pas être limitée par la complexité du système. Nous proposons d'utiliser la méthode B et son concept de raffinement pour simplifier le processus de modélisation et de preuve. À chaque étape du raffinement, des obligations de preuve sont générées par les outils supports pour assurer que le modèle plus concret est bien correct vis-à-vis du modèle abstrait. Cette méthode assure que l'implantation finalement obtenue est correcte vis-à-vis du premier modèle abstrait qui constitue la spécification. Nous avons commencé par une étude de cas réaliste choisie par un industriel (Volvo) consistant à modéliser un contrôleur d'accès à un bus série. Cette étude de cas nous a permis de dégager des règles de modélisation pour le développement de circuits électroniques par la méthode B. Cela nous a conduit à définir le langage BHDL correspondant au niveau synthétisable de B et des traducteurs de BHDL vers VHDL et SystemC ont été développés. Une étude théorique du langage BHDL a été faite en définissant deux sémantiques de ce langage et en prouvant la correction de la traduction de BHDL vers VHDL. Des travaux ont également été faits pour traduire BHDL vers ACL2.

Mots-clés: méthode B, circuits électroniques, raffinement

Abstract

Electronic systems become more and more complex and reliability requirements are more and more important. The challenge is to continue to develop more and more complex systems while ensuring correction of systems. Test-based methods are now overtaken by complexity of systems. We suggest using proof and refinement to ensure correction of systems. Proof-based methods are not limited by the complexity of systems. We suggest using the B method and its concept of refinement to simplify the process of modelling and proving. At each refinement step, proof obligations are automatically generated by tools to ensure that the concrete model is correct with respect to the abstract model. This method ensures that the final implementation is correct with respect to the initial abstract model which is the specification. We started by a realistic case study chosen by an industrialist (Volvo) consisting in modelling an access controller for a serial bus. This case study leaded us to define some modelling rules to develop electronic circuits using the B method. We have defined the BHDL language which is the synthesisable level of B and we have implemented translators from BHDL towards VHDL and SystemC. A theoretic study of BHDL has been done defining two semantics for this language and proving the correction of the translation from BDHL to VHDL. Some work has also been done to translate BHDL to ACL2.

Keywords: B method, electronic circuits, refinement

Remerciements

Je souhaite tout d'abord remercier les membres du Jury, qui ont permis la soutenance de ma thèse :

- Jacques Julliand et Laurence Pierre, pour avoir accepté la tâche difficile de relire le manuscrit. Merci pour leurs conseils.
- Monique Grandbastien et Dominique Mery pour m'avoir fait l'honneur de faire partie du jury de thèse.
- Un remerciement particulier à Stefan Hallerstede, pour avoir accepté de faire partie du jury de thèse et pour m'avoir aussi bien accueilli et encouragé en début de thèse dans la société KeesDA.

Je souhaite remercier grandement mon directeur de thèse, Dominique Cansell pour avoir accepté de diriger mon travail et m'avoir soutenu dans les moments difficiles, rendant ainsi possible la soutenance de ma thèse.

Je dois également remercier la société KeesDA qui a financé ma thèse et dans la quelle j'ai travaillé au quotidien. Je remercie le LORIA mon laboratoire d'accueil qui m'a accueilli régulièrement. Certains travaux effectués durant cette thèse ont été en partie financés grâce au projet européen PUSSEE (projet IST-2000-30103) ainsi que par l'ANVAR. Je remercie donc la commission européen et l'ANVAR pour les financements qu'ils ont apportés.

Table des matières

Introduction

Les systèmes électroniques sont de plus en plus présents dans notre vie quotidienne : appareils de télécommunication, électroménager, appareils audio et vidéo, systèmes embarqués dans les moyens de transport, etc. Ils deviennent également de plus en plus complexes, et, dans le même temps, les mauvais fonctionnements d'un système électronique coûtent de plus en plus cher. Que ce soit dans le domaine grand public où une erreur de conception peut amener au remplacement de milliers d'appareils, ou dans des domaines comme le transport où une erreur peut amener à la perte de vies humaines. Certains standards comme IEC 61508 [58] ou RTCA Do-254/EUROCAE ED-80 [81, 42], ont été développés pour faire face à ces problèmes, notre approche utilisant la méthode B peut être utilisée dans les parties de ces standards concernant la spécification et la validation. Le défi est de continuer à développer des systèmes de plus en plus en complexes tout en assurant la correction de ces systèmes.

Les méthodes de test ayant atteint leurs limites, les méthodes formelles sont maintenant nécessaires pour assurer la correction d'un système. La vérification formelle de circuits est souvent basée sur le model-checking qui est limité par le nombre d'états du système. Des méthodes symboliques comme l'évaluation symbolique de trajectoires [51] peuvent augmenter l'efficacité du model-checking. La preuve de théorème à l'avantage de ne pas être limitée par la taille de l'espace des états qui peut être inconnu (ou paramétré). Des exemples d'applications de prouveurs de théorèmes pour la vérification de systèmes électroniques sont ACL2 [19, 82], HOL [43] ou PVS [85].

1 Problématique

Il est maintenant devenu impossible de valider entièrement un circuit par le test : si on prend l'exemple d'un additionneur 32bits (2^{64} vecteurs de tests), il faudrait une ferme d'ordinateurs capable d'appliquer plus de 585 milliards vecteurs de test à la seconde pour que l'application de tous les vecteurs de test prenne moins d'une année. Et un additionneur n'est qu'un exemple très simple, les circuits sont aujourd'hui beaucoup plus complexes et vont le devenir encore plus dans les années à venir. Par exemple dans [61], Kuekes, Duncan et Williams expliquent comment il est possible d'augmenter encore plus la miniaturisation (et donc la complexité des systèmes sur une puce) des circuits en se passant des transistors, en les remplaçant par des "crossbar latch" qui n'utilisent que quelques atomes. La loi de Moore disant que la capacité des processeurs double tous les 18 mois ne semblent donc pas prête d'être infirmée.

Par ailleurs, il n'existe pas encore d'outil capable de faire de la synthèse haut niveau dans le cas général. La synthèse de haut niveau peut se comparer à la compilation dans le cadre des logiciels : le modeleur du système donne une description de haut niveau de son système et un outil support compile cette description vers un niveau d'implantation, le modeleur perdant ainsi, en tout ou partie, le contrôle de l'architecture finale de son système. Récemment, le langage SystemC a été augmenté d'une bibliothèque TLM (Transaction Level Model), permettant la description de systèmes à haut niveau mais il n'existe pas de synthétiseur capable de fournir une implantation du système à partir du niveau TLM. La transition doit donc se faire à la main.

La solution pour se détourner du test qui n'est plus une méthode valide pour la vérification fonctionnelle d'un système est de se tourner vers les méthodes formelles qui sont capables de faire face à la complexité d'un système.

Parmi les méthodes (et formalismes) utilisées dans la modélisation de matériel on peut citer le "model-checking" (par exemple basé sur les State Charts [39]), les Actions Systems [78, 77], la vérification (par

exemple ACL2 [14], Coq [32]), ou la transformation (par exemple Alpha [16]). La méthode B que nous utilisons propose la technique du raffinement. Le raffinement permet de modéliser un système pas à pas en commençant par une description très abstraite comportant peu de détails du système modélisé. Cette description abstraite est ensuite raffinée en ajoutant des détails. Par pas de raffinement successifs, on se rapproche peu à peu de l'implantation du système. Le raffinement a l'avantage de faire face aux systèmes complexes car il permet de diviser la preuve de correction en parties plus faciles à gérer.

Dans le développement de systèmes corrects, on applique des méthodes formelles pour obtenir la correction du système par preuve. L'effort additionnel nécessaire pour la modélisation du système et la preuve des propriétés dans le développement est habituellement payant dans les environnements où la sécurité est critique, ou bien quand le coût d'un échec est très grand. Les systèmes électroniques devenant de plus en plus complexes, le besoin en méthodes formelles se fait de plus en plus sentir.

2 Apport de la thèse

La méthode B [1] est l'une des méthodes formelles qui possède suffisamment d'outils support pour être utilisée en pratique aujourd'hui. Le domaine d'application de la méthode B est le développement de logiciels. Elle est utilisée dans [20] pour la vérification de circuits par traduction du code VHDL en B. Cette démarche ne permet pas la modélisation à haut niveau d'abstraction. Notre approche est inverse : partir d'un modèle B de haut niveau et produire du VHDL.

Plus récemment, la méthode B a aussi été utilisée dans le domaine du développement de systèmes [4, 3], donnant lieu au B événementiel. Certains efforts ont été faits depuis pour l'appliquer également au développement de matériel [18, 23], mais sans aller jusqu'à l'implantation. Notre but est de dégager les principes de modélisation qui permettent de modéliser des circuits électroniques synchrones et de les traduire vers des langages de description de circuits.

La méthode B a déjà fait ses preuves dans la conception de logiciels. Elle possède des outils supports comme atelierB, B4free et Btoolkit, et bientôt avec B# (projet RODIN) une boite à outils gratuite ce qui devrait favoriser l'enseignement de la méthode B et donc une meilleure connaissance de cette approche dans les milieux industriels.

Comme dans le cas de la bibliothèque TLM pour SystemC dont nous avons parlé plus haut, de plus en plus de langages proposent de faire de la modélisation système à des niveaux d'abstraction relativement hauts (pour une simulation plus rapide) et plus seulement de circuits électroniques au niveau de l'implantation. Les fournisseurs d'IP proposent maintenant des systèmes complets sur une puce (SOC-System On Chip). Comme nous l'avons dit, le passage de ce niveau système au niveau d'implantation se fait à la main, faute d'outils de synthèse de haut niveau. La notion de raffinement de modèles de hauts niveaux vers des modèles de bas niveaux est donc de plus en plus présente dans l'industrie de l'électronique et la méthode B gère formellement la notion de raffinement.

Nous résumons les apports de l'approche décrite dans cette thèse en six points.

1. Modélisation abstraite de systèmes électroniques.

   Faire des modélisations abstraites en B n'a rien de nouveau, mais faire des spécifications abstraites de systèmes électroniques est plus novateur. En effet les spécifications de circuits électroniques sont généralement données à un niveau très bas, en général au niveau de l'implantation. Nous avons montré par l'exemple (cf. chapitres 6 et 7) que nous pouvions créer une spécification abstraite à partir d'une spécification de départ de très bas niveau. Cette spécification abstraite est le départ d'une chaîne de raffinements.

2. Raffinement vers une implantation de circuits électroniques à partir d'une spécification abstraite.

   Là encore, faire du raffinement est une activité caractéristique de la méthode B, mais jusqu'ici les raffinements concernant des systèmes électroniques s'arrêtent en général au niveau des protocoles de communication, sans aller jusqu'à l'implantation. Nous montrons dans cette thèse comment il est possible d'utiliser le raffinement jusqu'au niveau de l'implantation, description à partir de laquelle il est possible de la traduire vers les langages habituels de descriptions de circuits, permettant par la suite de faire de la simulation ou de la synthèse.

3. Définition de règles de modélisation.

Nous avons défini des règles de modélisation de circuits en B événementiel. Ces règles sont issues de l'expérience, en particulier de l'étude de cas SAE J1708 présentée dans le chapitre 7.

En montrant de quelle manière on peut modéliser des circuits en B événementiel, nous donnons une sémantique de circuit au langage B, initialement conçu pour le développement de logiciels. Cette sémantique spécifique aux circuits électroniques synchrones se voit notamment dans la notion d'*ordonnancement* pour représenter l'architecture du système modélisé et la notion de *période* de cycle, permettant de modéliser les communications entre divers composants du système. Nous détaillerons ces notions dans le chapitre 5.

4. Définition du langage BHDL®[1], de sa sémantique et sa traduction vers d'autres langages (cf. chapitres 8 à 14).

   BHDL est le sous ensemble du langage B correspondant à une description implantable par circuit synchrone (comme l'est B0 pour le logiciel). Nous avons donné deux sémantiques formelles à ce langage (cf. chapitres 11 et 12) et définit la traduction vers d'autres langages de description de circuits comme VHDL et SystemC (cf. chapitre 14). Au lieu de définir directement la traduction vers ces langages, nous avons défini un langage de référence à partir duquel il est simple de traduire vers VHDL et SystemC. Ce langage de référence est suffisamment simple pour que nous ayons pu le doter d'une sémantique formelle et ensuite prouver la correction de la traduction à partir de BHDL vers ce langage de référence.

5. Possibilité de continuer le raffinement à partir d'une implantation.

   Lorsqu'on a obtenu un modèle implantable par un processus de raffinement, il est possible que le résultat ne satisfasse pas complètement les experts en systèmes électroniques. Dans ce cas il est possible de continuer le raffinement afin d'obtenir une autre implantation intégrant les remarques faites par les experts en électroniques.

6. Définition de la notion d'aplatissement d'un modèle BHDL.

   Un modèle BHDL aplati est un modèle dans lequel les nouvelles valeurs de chaque variable sont exprimées directement en fonction des entrées et des registres (sans utiliser de variable intermédiaire). Un tel modèle est constitué de substitutions composées en parallèle, chacune de ces substitutions ne modifiant qu'une seule variable. Cette notion d'aplatissement a été utilisée pour définir la traduction d'un modèle BHDL en ACL2.

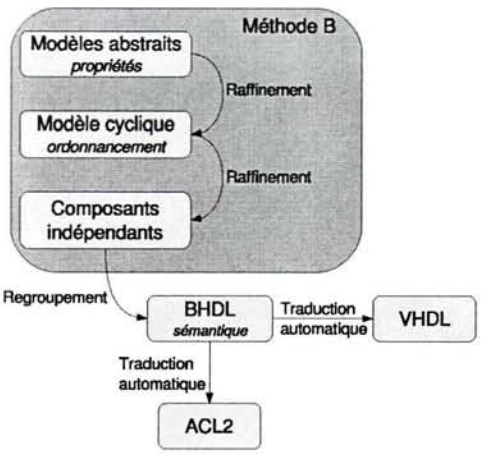
La figure 1 résume le processus de développement proposé dans cette thèse : le système est d'abord modélisé à des niveaux abstraits orientés par les propriétés du système. Le système est ensuite raffiné pour introduire le cycle d'horloge, pour cela nous montrons dans cette thèse comment modéliser un ordonnancement entre les événements du modèle pour modéliser le cycle du système et les communications entre les composants pendant un cycle. Le système est alors de nouveau raffiné de façon a séparer les composants en faisant en sorte que chaque composant ait son propre état qu'il est le seul à pouvoir modifier. Nous avons définit des règles de modélisation permettant de définir le niveau implantable d'un modèle B. Une fois ce modèle implantable obtenu, on peut regrouper les événements pour obtenir le modèle BHDL. Nous définissons ce langage dans ce document et nous lui donnons une sémantique formelle. Nous avons défini la traduction vers un langage cible (les traductions vers VHDL et SystemC ont été implantées). Nous avons donné une sémantique au langage cible et nous avons prouvé la traduction. Nous avons également défini et implanté une traduction de BHDL vers ACL2 de façon à pouvoir comparer un modèle BHDL avec un modèle BHDL : le modèle VHDL est aussi traduit vers ACL2 et on prouve une bisimulation entre les modèles en utilisant le prouveur d'ACL2.

3 Contexte de travail

Cette thèse a fait l'objet d'un contrat CIFRE entre la société KeesDA et le laboratoire Loria (équipe Mosel). Le travail quotidien s'est effectué dans l'entreprise KeesDA et des visites ont été rendues régulièrement au laboratoire Loria. Le début de cette thèse s'est également inscrit dans le cadre du projet PUSSEE (projet IST).

[1] BHDL est une marque déposée par KeesDA

FIG. 1 – Processus de développement



3.1 KeesDA

KeesDA est une petite société Grenobloise qui a choisi à partir de l'an 2000 de développer une technologie pour la conception prouvée de systèmes électroniques, (entre autres les systèmes sur une puce, SOC) en appliquant à la production du hardware les principes, jusque là mis en œuvre pour la génération de programmes, de la méthode B.

Un premier programme coopératif de développement a été soutenu par la commission européenne en 2002 et 2003, il s'agit du projet PUSSEE (voir paragraphe suivant) dont KeesDA assura la direction technique. Puis l'ANVAR, à son tour, manifesta son intérêt pour la technologie de KeesDA en lui octroyant une aide à l'innovation.

KeesDA, malgré sa taille très modeste, poursuit une importante activité de recherche en collaboration avec plusieurs universités européennes, en Angleterre, en Allemagne ou en France (comme le Loria). Ont contribué à cette recherche, outre son directeur scientifique, ancien directeur de recherches au CNRS, un docteur de l'université de Southampton et une docteur de l'université de Grenoble.

KeesDA a été invitée par les leaders industriels européens de l'électronique à être partenaire dans le projet intégré SPRINT et dans le projet "Open-TLM" du pôle d'excellence Minalogic.

3.2 Équipe MOSEL du laboratoire LORIA

L'équipe MOSEL [41] du laboratoire LORIA [65] fait de la recherche, du développement et de l'enseignement dans le domaine des méthodes formelles pour la conception et l'analyse de systèmes automatisés.

S'appuyant sur plus d'une dizaine d'années d'expérience concernant la description précise et l'analyse de systèmes, en utilisant différentes méthodes et différents outils, l'équipe préconise une approche basée sur le concept de raffinement, permettant de prouver des propriétés cruciales sur un modèle simple (abstrait), puis maintenues pendant la phase de développement (par raffinements).

Les projets industriels aident à valider les idées grâce à des études de cas réalistes et d'appliquer les concepts à des modèles exprimés dans les langages de description disponibles actuellement.

3.3 Projet PUSSEE

L'objectif du projet PUSSEE [80, 69] était d'introduire la preuve formelle de propriétés systèmes en même temps qu'une méthodologie modulaire de conception de systèmes et de réutilisation de composants existants. L'objectif étant de réaliser cela en combinant UML et la méthode B.

Ce projet a réuni plusieurs acteurs européens du domaine de la conception de systèmes électroniques comme Volvo (Suède), Nokia (Finlande), Intracom (Grèce), ClearSY (France), l'université de Southampton (Angleterre), l'université de Parderborn (Allemagne) et KeesDA. La méthodologie et les outils

développés ont été validés sur applications industrielles : un terminal mobile de communication sans fil, un composant de cryptage pour des transmissions sécurisées sur internet et un contrôleur pour un réseau utilisé dans l'industrie automobile (cf. chapitre 7)

Le rôle de KeesDA dans ce projet a été de définir une méthodologie de développement de systèmes électroniques par raffinement depuis un niveau abstrait correspondant à la spécification du système jusqu'au niveau de l'implantation. KeesDA a également développé deux traducteurs permettant de traduire le niveau d'implantation B vers SystemC et VHDL au niveau synthétisable. C'est cette approche qui est développée dans cette thèse.

4 Plan général de la thèse

Ce document est divisé en cinq parties. La première partie décrit le contexte dans lequel s'inscrit cette thèse. Nous expliquons dans le premier chapitre ce que sont les circuits électroniques synchrones et donnons trois exemples de modélisation d'un même exemple dans trois langages différents de façon à montrer la diversité qui existe dans les langages de modélisation. Le deuxième chapitre donne l'état de l'art en ce qui concerne les méthodes formelles utilisées pour la conception ou la vérification de circuits électroniques. Nous donnons quelques exemples d'applications concernant le model-checking et la preuve de théorème pour la conception de circuits corrects. La méthode B est expliquée plus en détail dans le troisième chapitre. Nous y donnons la définition des substitutions et des événements, qui sont la base du langage B. Nous terminons sur un exemple de modélisation de circuit de façon à introduire la façon dont on peut modéliser des circuits électroniques synchrones en B.

La façon dont on modélise des circuits synchrones est faite de manière détaillée dans la deuxième partie. Le quatrième chapitre donne brièvement les différents principes à mettre en œuvre pour la modélisation de circuits en B. Le cinquième chapitre explique en détail comment obtenir un modèle B implantable d'un circuit électronique. Nous donnons dans les deux chapitres suivants des études de cas illustrant les principes énoncés auparavant. La première étude concerne un circuit dont la spécification est donnée par une équation récurrente (somme de convolution) alors que la deuxième étude de cas concerne un contrôleur de bus série utilisé dans l'industrie automobile. Cette étude de cas a été choisie par Volvo dans le cadre du projet PUSSEE.

La troisième partie est consacrée au langage BHDL dont la définition est donnée dans le huitième chapitre. Dans ce chapitre nous définissons en particulier les notions d'ensemble support en écriture et d'ensemble support en lecture. Le neuvième chapitre introduit les substitutions généralisées telles qu'elles sont définies dans la méthode B classique sur laquelle est fondé le langage BHDL. Nous donnons la sémantique du langage de deux façons (par prédicat et par modèle ensembliste), ces sémantiques servent de base à la définition des sémantiques de BHDL. Le dixième chapitre est une introduction qui montre comment on peut définir une sémantique de trace pour le langage BHDL. Le onzième chapitre donne une sémantique du langage BHDL basée sur les prédicats alors que le douzième chapitre donne une sémantique de BHDL basée sur des relations ensemblistes. La première sémantique respecte l'architecture de la description BHDL et peut donc être utilisée pour définir la traduction vers d'autres langages. La deuxième a l'avantage d'être compositionnelle sur la majeure partie du langage et permet donc de prouver des propriétés par induction. Le treizième chapitre explique comment on peut introduire des importations de composants existants en BHDL en se basant sur la sémantique relationnelle. Le quatorzième chapitre donne les règles de traduction de BHDL vers un langage de description de circuits. Le langage cible est un langage inventé (auquel nous donnons une sémantique) dont les concepts sont proches de VHDL et SystemC et qui nous sert de référence à partir duquel il est relativement facile de passer vers SystemC ou VHDL. Les règles données ne comportent pas les règles d'optimisation utilisées dans les traducteurs développés par KeesDA.

La quatrième partie concerne des travaux qui ont été effectués pendant la thèse et qui sont en relation avec elle. Le quinzième chapitre explique comment on peut vérifier un composant existant en ACL2 dans le but de le réutiliser dans un flot de conception B. Pour cela, nous décrivons des règles d'aplatissement d'un modèle BHDL de façon à le traduire vers ACL2. Le seizième chapitre montre que l'on peut modéliser les concepts de la modélisation transactionnelle en B. La modélisation transactionnelle est introduite depuis quelques années de façon à permettre la simulation d'un modèle (dit transactionnel) de haut niveau

d'abstraction, où les communications entres les différents éléments se fait par des transactions plutôt que par des protocoles. Ceci permet une simulation plus rapide des modèles. Ces modèles doivent cependant être ensuite raffinés pour obtenir une description implantable, la méthode B pouvant intervenir à ce stade.

La cinquième partie contient les annexes du document. La première annexe donne la grammaire du langage BHDL. La deuxième annexe donne la preuve de compositionnalité des ensembles supports. La troisième annexe donne la preuve de correction de la traduction de BHDL vers le langage de référence. La quatrième annexe donne la preuve de correction de l'aplatissement utilisé pour la traduction de VHDL vers ACL2. La cinquième donne une machine B utilisée prouvant une propriété utilisée dans l'étude de cas du contrôleur de bus série. La sixième annexe donne le modèle B de niveau implantable de l'étude de cas du contrôleur de bus série.

5 Publications relatives à cette thèse

Conférences internationales avec comité de lecture

Yann Zimmermann, Diana Toma - Component Reuse in B using ACL2, ZB2005, LNCS 3455, pages 280-299

Stefan Hallerstede, Yann Zimmermann - Circuit Design by Refinement in EventB, Proc. of FDL'04, 13 pages, 2004

Conférence nationale avec comité de lecture

Yann Zimmermann - Modélisation formelle de circuits électroniques en B événementiel, MAJECSTIC, 29-31 octobre 2003

Chapitre dans un livre

Yann Zimmermann, Stefan Hallerstede, Dominique Cansell - Formal modelling of electronic circuits using event-B, Case Study : SAE J1708 Serial Communication Link - pp 211-226

Dans le livre

UML-B - Specification for Proven Embedded Systems Design, J. Mermet, editor, KeesDA, Kluwer Academic Publishers, 2004

Dans une revue

Dominique Cansell, Stefan Hallerstede, Yann Zimmermann - Construction sûre de systèmes électroniques, revue Génie Logiciel n°69 Juin 2004, pp 38-44

Rapports techniques

Yann Zimmermann, Stefan Hallerstede, Dominique Cansell - Formal modelling of electronic circuits using event-B Case Study : SAE J1708 Serial Communication Link - Deliverable 4.2.4 projet PUSSEE - 86 pages - Janvier 2004

Yann Zimmermann - Modélisation des concepts TLM en B événementiel - rapport ANVAR - 19 pages - Octobre 2004

Yann Zimmerman, Diana Toma - Manuel de référence du langage BTLM - rapport ANVAR - 36 pages - Juillet 2005

Yann Zimmermann, Diana Toma - Règles de traduction de BTLM vers SystemC - rapport ANVAR - 28 pages - Août 2005

Yann Zimmermann - Etude de cas : console multimédia - rapport ANVAR - 49 pages - Décembre 2005

Première partie

Contexte

Chapitre 1

Circuits électroniques

L'objectif de chapitre est de présenter ce que sont les circuits électroniques, en particulier les circuits électroniques synchrones. Nous présentons également comment de tels circuits peuvent se modéliser en VHDL (langage standard de description de circuits), en BHDL (langage introduit dans cette thèse) et en ACL2 (langage formel).

Un circuit électronique est constitué de composants connectés entre eux. Ces composants peuvent être élémentaires (portes logiques, à base de transistors), ou des composants plus complexes. Ces composants sont connectés par des fils sur lesquels se propagent des *signaux* électriques. Sur un fil donné le signal se propage toujours dans la même direction. Habituellement le signal qui se propage ne peut se trouver que dans un nombre fini d'états. Classiquement, le signal ne peut se trouver que dans deux états : un état de haute tension noté '1' (ou *true*) et un état de basse tension noté '0' (ou *false*), les portes logiques se comportant alors comme des fonctions booléennes. Cependant on trouve aussi des circuits utilisant des logiques à trois valeurs ou plus, dans ce cas on utilise des logiques associées. En regardant le circuit comme une boîte noire, certains signaux entrent dans le circuit, ce sont les *entrées* et certains signaux sortent du circuit, ce sont les *sorties*. Les entrées et les sorties prenant des valeurs booléennes, le circuit peut être modélisé comme une fonction booléenne. Suivant cette fonction, le circuit peut être *combinatoire* (les sorties sont entièrement définies par les entrées au même instant) ou *séquentiel* (les sorties peuvent dépendre le l'historique des entrées). Les circuits séquentiels comportent des éléments de mémoire et sont dits *synchrones* ou *asynchrones* suivant le type de mémoires utilisées. Dans ce document, nous nous intéressons uniquement aux circuits synchrones (ou combinatoires si le circuit ne possède aucune mémoire).

1.1 Circuits combinatoires

Un circuit combinatoire est un circuit ne possédant aucun élément de mémoire. Ainsi à un instant donné, les sorties sont fonctions des entrées au même instant.

Un circuit combinatoire (ou circuit logique) est physiquement réalisé par un assemblage de portes logiques. Les portes logiques correspondent aux fonctions booléennes de base : ET, OU, OU exclusif, NON, NON OU, NON ET. Sauf la porte logique NON qui ne prend qu'une entrée et renvoie une sortie, toutes les autres portes logiques listées précédemment prennent deux entrées en renvoient une sortie. Ces portes logiques correspondent à des opérateurs booléens dont nous donnons les tables de vérité ci-dessous.

ET	0	1
0	0	0
1	0	1

NON ET	0	1
0	1	1
1	1	0

OU	0	1
0	0	1
1	1	1

NON OU	0	1
0	1	0
1	0	0

OUX	0	1
0	0	1
1	1	0

NON	
0	1
1	0

Pour chaque sortie o_i, un circuit combinatoire définit une fonction booléenne f_i qui relie o_i aux entrées

$i_1, ..., i_n$.

$$o_i = f_i(i_1, ..., i_n)$$

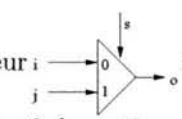
Le délai de propagation des signaux électriques des entrées jusqu'aux sorties n'est pas nul et peut être calculé par les outils de synthèse de circuits. Le délai δ calculé correspond au chemin le plus long du circuit. Plus il y a de portes logiques entre les entrées et les sorties, plus le délai de propagation est long. Pendant ce délai, les sorties sont instables et ne doivent pas être utilisées.

$$o_i(t) = f_i(i_1(t - \delta), ..., i_n(t - \delta))$$

Un circuit combinatoire ne contenant pas d'élément de mémoire, l'architecture résultant des connections entre les composants du circuit ne contient pas de cycle. Un cycle dans l'architecture introduirait un court circuit (ou éventuellement une mémoire en tenant compte du délai de propagation). Les cycles dans un circuit combinatoire ne sont pas autorisés.

L'architecture d'un circuit combinatoire peut être modélisée par un graphe acyclique dirigé (DAG). Les nœuds du graphe sont les composants électroniques de base ou d'autres circuits combinatoires (sous-composants). Les arcs dirigés du graphe sont les fils du circuit qui relient les sorties des composants aux entrées d'autres composants. La direction des arcs indiquent le sens de propagation des signaux électriques le long des fils. L'absence de cycle dans le circuit correspond à l'absence de cycle dans le graphe dirigé.

Dans ce document nous adoptons quelques notations graphiques pour représenter un circuit comme un graphe :

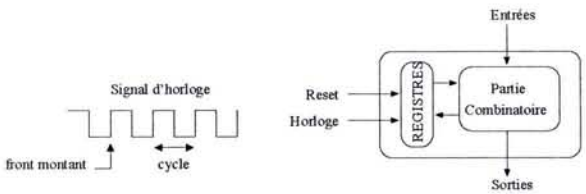
- arcs dirigés
  - ⟶ est un fil, la flèche indique le sens de propagation du signal électrique,
  -  est une connexion directe de plusieurs fils, tous les fils portent le même signal électrique, tous les fils doivent avoir la même direction.
- nœuds
  - représente un opérateur *Op* qui transforme le signal d'entrée vers le signal de sortie, il peut y avoir plusieurs entrées et plusieurs sorties. Un opérateur peut ne pas avoir d'entrée et par exemple produire constamment le même signal en sortie
  - Le multiplexeur  est un opérateur qui a trois entrées et une sortie. Lorsque l'entrée s porte la valeur 0, la sortie o est connectée à l'entrée i, sinon la sortie est connectée à l'entrée j.

Chaque fois qu'une entrée change, les nouvelles valeurs des signaux se propagent dans le circuit combinatoire et finalement (après un délai de propagation) jusqu'aux sorties. Ainsi, d'une manière générale, dans un circuit combinatoire les sorties ne sont stables que si les entrées sont stables.

1.2 Circuits synchrones

Un circuit synchrone est un circuit dans lequel les éléments de mémoire sont des *flip/flops*. Il existe plusieurs sortes de flip/flop, le principe général étant que les flip/flop sont commandés par un signal d'*horloge*. Nous appelons *registre* un flip/flop qui est sensible au front montant du signal d'horloge : la valeur mémorisée change uniquement lorsque le signal d'horloge passe de l'état *bas* à l'état *haut*. Entre deux fronts montants du signal d'horloge, la sortie d'un registre (c'est-à-dire la valeur mémorisée) ne change pas et les modifications de l'entrée du registre ne sont pas prises en compte avant le prochain front montant de l'horloge. Le signal d'horloge est cyclique : l'intervalle de temps entre deux fronts montants est constant (cf. figure 1.1). Cet intervalle est appelé un cycle d'horloge.

FIG. 1.1 – Le signal d'horloge et vue schématique d'un circuit synchrone (modèle de Mealy)



Tous les registres sont supposés être commandés par la même horloge et donc évoluer en même temps, de façon *synchrone*. Certains registres peuvent aussi avoir une autre entrée, appelée signal de *réinitialisation asynchrone* qui permet de réinitialiser la valeur mémorisée par le registre, indépendamment de la valeur de l'horloge. Ce signal est modifié par l'environnement du circuit. Il doit être mis en état *haut* pour initialiser le circuit lorsque l'ensemble du système démarre. Lorsque l'initialisation du système est terminée, ce signal doit rester indéfiniment au niveau *bas* (à moins que le système ait besoin d'être réinitialisé).

Notre but est de modéliser des circuits électroniques synchrones. Ce type de circuit possède une entrée cyclique, l'horloge. Le circuit peut être vu comme séparé en deux parties : une partie combinatoire, dans laquelle les signaux se propagent, et des registres qui mémorisent une valeur pendant un cycle d'horloge.

Il s'en suit que le circuit a lui-même un comportement cyclique. Lorsque l'horloge commence son cycle, les registres prennent une nouvelle valeur. Les entrées ne changent pas au même moment que le coup d'horloge (on obtiendrait un comportement chaotique du circuit). Lorsque les entrées changent, les nouveaux signaux se propagent dans le circuit. Ainsi, il est inutile de modéliser ce qui s'est passé entre la mise à jour des registres et le changement des entrées. Les signaux se propagent dans la partie combinatoire du circuit avant la fin du cycle (la fréquence de l'horloge doit être réglée pour cela). Certains signaux sont envoyés en sortie, d'autres en entrée des registres. Dès le début du cycle suivant, les registres prennent les nouvelles valeurs, les entrées changent ...

1.3 Modèle mathématique des circuits synchrones

Un modèle usuel pour donner une représentation mathématique d'un circuit synchrone est celui de la *machine de Mealy* [68].

Un circuit synchrone peut se diviser en deux parties (cf. figure 1.1) : une partie combinatoire d'une part et des registres d'autre part. Les registres mémorisent des valeurs pendant un cycle d'horloge. Lorsqu'il y a un front montant de l'horloge, les registres prennent de nouvelles valeurs et restent inchangés jusqu'au prochain front montant d'horloge. L'ensemble des registres constitue l'état de la machine de Mealy. La partie combinatoire est la connexion des différents éléments logiques qui ne contiennent pas de mémoires. La partie combinatoire est reliée aux registres en leur fournissant des entrées et en lisant leurs sorties.

Dans une machine de Mealy, à un instant donné, les sorties sont des fonctions des entrées et de l'état de la machine au même instant. Éventuellement, la partie combinatoire peut être telle que les sorties ne dépendent que de l'état de la machine, dans ce cas on obtient une *machine de Moore*. Cela peut être utile si on souhaite s'assurer que les sorties soient réellement stables pendant tout un cycle d'horloge.

Une machine de Mealy peut être définie par trois fonctions :

- la fonction *reset* qui donne l'état initial de la machine de Mealy,
- la fonction *next* qui calcule la valeur du nouvel état (pour un nouveau cycle d'horloge) à partir de l'état courant et des entrées,
- et la fonction c qui calcule les valeurs des sorties à partir de l'état et des entrées.

Le comportement du circuit est cyclique, commandé par le signal d'horloge. A chaque cycle la nouvelle valeur de l'état est calculée, les entrées sont lues, les sorties sont calculées en utilisant les nouvelles entrées et l'état courant (et non pas le nouvel état). En notant les entrées par i, les sorties par o et l'état par s,

la machine de Mealy peut être modélisée par l'équation ci-dessous. Pour un élément x, la notation x_k se réfère à la valeur de x au k-ième cycle.

$$s_0 = reset \quad \wedge \quad \forall k \geq 0 \quad (o_k, s_{k+1}) = (c(i_k, s_k), next(i_k, s_k))$$

1.4 Circuits asynchrones

Un circuit asynchrone est un circuit séquentiel qui n'est pas contrôlé par un signal d'horloge. Les mises à jours des éléments de mémoires ne sont donc pas synchronisés et les valeurs mémorisées peuvent être modifiées dès que l'entrée est disponible et que la sortie a déjà été utilisée.

Le désordre engendré par cette perte de cohésion complique certes la conception des puces, mais le flux de données traitées augmente. En effet dans un circuit synchrone la fréquence de l'horloge est calculée par rapport au chemin le plus long existant dans le circuit. Ainsi, l'ensemble du circuit, en particulier les parties rapides du circuit, est limité par la partie la plus lente du circuit.

Les avantages potentiels des circuits asynchrones sont notamment des vitesses de fonctionnement supérieures, une consommation électrique réduite (les circuits de synchronisation pouvant consommer jusqu'à 35% de l'énergie nécessaire pour le fonctionnement d'un processeur [10]) et des interférences radio négligeables.

1.5 Langages de description de circuits

Les trois principaux langages de description de circuits (HDL - Hardware Description Language) sont VHDL, Verilog et SystemC. Nous donnons ci-dessous un bref historique de ces trois langages [73, 74, 75].

VHDL (VHSIC Hardware Description Langage) [92, 54], est un langage de description de matériel, c'est-à-dire un langage utilisé pour décrire un système numérique matériel, comme, par exemple, un flip-flop (bascule D) ou un microprocesseur. Il peut modéliser un système par n'importe quelle vue, structurelle ou comportementale, à tous les niveaux de description. De plus il peut servir non seulement à simuler un système mais aussi à le synthétiser, c'est-à-dire être transformé par des logiciels adaptés (synthétiseurs) en une série de portes logiques prêtes à être gravées sur du silicium. Le langage standard IEEE VHDL a été développé par le Groupe d'Analyse et de Standardisation VHDL (VASG, pour "VHDL Analysis and Standardization Group"). Larry Saunders est le coordinateur de VASG. La société CLSI (CAD Langage Systems Inc.), représentée par le Docteur Moe Shahdad et M. Erich Marschner a préparé une série d'analyses et de recommandations dont a été tirée en Février 1986 la version 7.2 de VHDL, point de départ du futur standard. La collaboration de CLSI au projet était financée par un contrat passé avec l'Air Force Wright Aeronautical Laboratories, représentée par le Docteur John Hines. Le standard définitif a été adopté vers le milieu de l'année 1987 et la dernière version date de 2002.

Verilog [30] a été inventé par Gateway Design Automation Inc. aux alentours de 1984. C'était un langage propriétaire, inspiré d'un autre HDL (HiLO) et du langage C. Gateway a aussi produit le premier simulateur Verilog en 1985, ainsi que son successeur Verilog-XL, toujours utilisé. En même temps, la société Synopsys développe le premier synthétiseur, travaillant à partir de sources Verilog. En 1990, la société de CAO Cadence rachète Gateway. Elle décide de rendre les spécifications de Verilog publiques, qui deviennent un standard IEEE en 1995 (IEEE Std. 1364-1995 [55]). Dans le même temps, plusieurs sociétés développent des simulateurs et des synthétiseurs, dont VCS le premier simulateur-compilateur (en comparaison, Verilog-XL est un interpréteur). Une extension de Verilog, appelée SystemVerilog est actuellement introduite sur le marché. Elle cherche à modéliser les systèmes à un niveau encore plus abstrait qu'actuellement, pouvant modéliser non seulement des systèmes matériels mais aussi des systèmes logiciels (OS, ...) ainsi que des mélanges des deux.

SystemC [87] est, comme Verilog et VHDL, un langage de description de matériel. C'est le fruit de contributions de plusieurs sociétés. En 1989, Synopsys met son outil commercial Scenic dans le domaine libre, et crée la version 0.9 de SystemC. Une première contribution de Frontier Design donne lieu à la version 1.0, et une autre de CoWare aboutit en 2000 à la version 1.1, première version officielle de SystemC. L'OSCI (Open SystemC Initiative) est alors créé, rassemblant une multitude de sociétés et de laboratoires de recherche. Cette organisation est en charge de diffuser, promouvoir et rédiger les

spécifications de SystemC. Les spécifications de SystemC ont été étendues en 2001 à la modélisation de systèmes abstraits (de très haut niveau, avant partitionnement matériel / logiciel), aboutissant à la version 2.0. La version actuelle stable est la 2.1 et est devenu le standard IEEE 1666 en Décembre 2005 [56].

1.6 Le niveau "RTL"

Le niveau RTL (Register Transfert Level) est le niveau auquel l'ensemble du circuit est décrit par des registres et des portes logiques. On peut aussi utiliser des composants plus complexes (comme des additionneurs, multiplicateurs ...) mis à disposition par des bibliothèques de composants et dont les outils de synthèse sont capables de les transformer automatiquement en portes logiques.

Le niveau RTL est le niveau de description cible pour tout développement d'un circuit électronique, qu'il soit obtenu automatiquement (par des outils de synthèse de haut niveau) ou à la main.

1.7 Le niveau "Netlist"

Le niveau de description d'un circuit par *netlist* est le niveau de description physique du circuit. A ce niveau, le circuit est décrit par un réseau de portes logiques et de transistors. Ce niveau de description physique n'est généralement pas obtenu à la main mais pas des outils de synthèse à partir du niveau RTL.

1.8 Trois styles différents de modélisation

Pour donner une idée des différentes façons dont un circuit électronique peut être modélisé, nous donnons ci-dessous trois modèles d'un même exemple. Nous choisissons l'exemple d'un compteur qui compte de 0 à 7 décrit à un niveau très bas. Cet exemple sera repris pour illustrer la modélisation de circuit en B dans la section 3.5 page 48 puis dans le chapitre 15 page 247 pour illustrer la méthode définie par Diana Toma pour prouver l'équivalence de deux circuits en ACL2.

1.8.1 VHDL

Un des langages de description de systèmes électroniques est VHDL (VHSIC – Very High Speed Integrated Circuit – Hardware Description Language) [54]. Nous nous focalisons sur le sous ensemble de VHDL qui modélise les circuits électroniques au niveau transfert de registre, c'est-à-dire le niveau auquel les circuits sont décrits en utilisant des signaux et des registres.

Une description VHDL se constitue d'une liste de processus concurrents. Le processus de base est l'affectation concurrente de signal $s \Leftarrow E$, s est le signal et E l'expression qui est affectée au signal. Le point virgule ";" dénote la composition concurrentielle : $t \Leftarrow s; s \Leftarrow E$ est l'affectation concurrente de l'expression s au signal t et de E à s. En comparaison avec des langages comme C ou Pascal, l'opérateur ";" n'a ni la même sémantique qu'en B, ni la même sémantique que $\|$. Il signifie que t est connecté à s *et* que s est connecté à E : ainsi t est aussi, indirectement, connecté à E. Écrire, "A ;B" ou "B ;A" est équivalent.

On obtient un processus plus complexe en utilisant un bloc d'opérations séquentielles. Dans chaque processus, les opérations sont exécutées séquentiellement et il est possible d'utiliser des variables locales. Les processus et les affectations de signaux sont concurrents. La sémantique de la concurrence est généralement donnée en utilisant les δ-délais (cf. [44] par exemple). Le principe consiste à appliquer de façon répétitive les processus et les affectations de signaux jusqu'à atteindre un point fixe.

Notons qu'il ne faut pas confondre les variables et les signaux, ce sont deux types d'objet différents. Les signaux correspondent généralement aux fils du circuit alors que les variables sont utilisées dans les processus comme un moyen de programmer une fonctionnalité. Lorsqu'on affecte une valeur à une variable, la valeur de la variable change immédiatement (comme c'est l'habitude dans les langages de programmation). Quand on affecte une valeur à un signal, la valeur du signal ne change pas immédiatement,

ce sont les futures valeurs du signal qui sont modifiées. Par exemple, on peut écrire s ⇐ '1' after 10ns, '0' after 30ns, ce qui signifie que la valeur du signal sera '1' après un délai de 10ns, puis '0' après 30ns (c'est-à-dire 20ns après le premier changement). La modification d'un signal peut également dépendre d'un événement, par exemple on peut écrire le multiplexeur s ⇐ '1' when t='0' else '0', ce qui signifie que le signal s portera la valeur '1' chaque fois que le signal t porte la valeur '0' et s porte la valeur '0' dans les autres cas.

FIG. 1.2 – Description VHDL d'un compteur

```
-- interface
entity counter is
    port (
        clock : in std_logic;
        reset : in std_logic;
        rst : in std_logic;
        alm3 : out std_logic);
end;
-- architecture
architecture tab of counter is
-- type du registre
     type tc_type is
    array (0 to 7) of std_logic;
-- déclarations des signaux
    signal gd : std_logic;
    signal tc_0 : tc_type; ...
begin
-- processus qui modélise les registres
process (clock, reset) begin
    if reset = '1' then
        tc_0 ⇐ "1000 0000"
    elsif clock'EVENT and clock='1' then
        tc_0 ⇐ tc_1;
    end if;
end process;

-- Partie Combinatoire

-- La sortie alm3 est connectée à
-- la dernière cellule du registre tc_0
        alm3 ⇐ tc_0(7);

-- Le premier bit de tc_6 est '1',
-- les autres bits sont à '0'
        tc_6(0) ⇐ '1';
    GEN1 : for k in 1 to 7 generate
        tc_6(k) ⇐ '0';
    end generate;

-- gd vaut '1' si maximum non atteint
        gd ⇐ not (tc_0(7));

-- tc_10 est tc_0 décalé à droite
        tc_10(0) ⇐ '0';
    GEN2 : for k in 1 to 7 generate
        tc_10(k) ⇐ tc_0(k - 1);
    end generate;

-- tc_8 est tc_10 sauf si maximum atteint
        tc_8 ⇐ tc_10 when gd='1'
                else tc_0;
-- Le registre est
-- réinitialisé (tc_6) si rst='1'
-- décalé à droite si maximum non atteint
-- inchangé sinon
        tc_1 ⇐ tc_6 when rst='1'
                else tc_8;
end;
```

La partie combinatoire du circuit est modélisée par une liste d'affectations concurrentes de signaux et un registre est modélisé en utilisant deux signaux et un processus dont le déclenchement est sensible aux signaux *clock* (signal d'horloge) et *reset* (signal de réinitialisation asynchrone). Un des deux signaux modélisant un registre porte la valeur courante du registre (le fil de sortie du registre) et l'autre porte la valeur suivante du registre telle que spécifié par la partie combinatoire.

Comme exemple, nous donnons sur la figure 1.2 le code VHDL d'un compteur. Ici le compteur n'est pas implanté par un nombre entier et un additionneur pour l'incrémenter. Nous utilisons un vecteur de bits (tc) qui contient un jeton (c'est-à-dire qu'un seul des bits du vecteur vaut '1' à un instant donné). A chaque cycle, (à moins que rst soit positionné à '1'), le jeton passe à la cellule suivante (modélisé dans le code par GEN2 et tc_10(0) ⇐ '0'). Lorsque le jeton a atteint la dernière cellule, il ne bouge plus jusqu'à ce que le signal rst soit positionné à '1'. Dans ce cas, le jeton retourne à la première cellule (modélisé dans le code par tc_6(0) ⇐ '1' et GEN1). Les signaux tc_0 et tc_1 sont respectivement la sortie et l'entrée du registre implantant le compteur. Les autres signaux tc_x sont des signaux intermédiaires. La sortie $alm3$ est une alarme qui est positionnée à '1' lorsque le jeton a atteint la dernière cellule : lorsque le compteur

a atteint son maximum. C'est pourquoi le signal de sortie *alm3* est connecté à la dernière cellule de *tc_0*. Nous reprenons ci-dessous les différents éléments de la description VHDL.

La déclaration de l'interface du circuit se fait en utilisant la structure entity. Elle contient la liste des ports du circuit, avec leurs types. Les entrées sont indiquées en utilisant le mot clef in et les sorties en utilisant le mot clef out.

```
entity counter is
    port (
        clock : in std_logic;
        reset : in std_logic;
        rst : in std_logic;
        alm3 : out std_logic);
end;
```

L'interface est composée de trois entrées (clock, reset et rst) et une sortie (alm3). Nous utilisons le type std_logic pour dire que ces valeurs d'interface peuvent prendre les valeurs '0' ou '1'. En fait le type std_logic contient sept autres valeurs que nous n'utilisons pas ici. L'entrée *clock* correspond à l'horloge et l'entrée *reset* au signal de réinitialisation asynchrone. L'entrée *rst* peut être utilisée pour remettre le compteur à zéro. Si *rst* vaut '0', le compteur est incrémenté par 1 à chaque cycle d'horloge. La sortie *alm3* vaut '1' lorsque le compteur à atteint sa valeur maximale.

La modélisation des registres se fait par un processus. Le fait que les signaux *clock* et *reset* soient mis entre parenthèses à côté du mot clef process signifie que le processus est sensible à ces deux signaux. C'est-à-dire qu'à chaque fois que l'un de ces deux signaux changera de valeur, le processus se déclenchera.

```
process (clock, reset) begin
    if reset = '1' then
        tc_0 <= "1000 0000"
    elsif clock'EVENT and clock='1' then
        tc_0 <= tc_1;
    end if;
end process;
```

Si le signal de réinitialisation asynchrone *reset* vaut '1' alors le tableau *tc_0* est réinitialisé : le jeton retourne dans la première cellule. L'expression clock'EVENT and clock='1' est l'expression utilisée pour détecter qu'il y a un front montant d'horloge : clock'EVENT signifie qu'il y a eu un changement de la valeur de l'horloge et clock='1' que la nouvelle valeur est '1'. Ainsi, lorsqu'il y a un front montant d'horloge, la valeur du tableau *tc_0* change pour prendre celle de *tc_1* : *tc_1* est l'entrée du registre et *tc_0* est la sortie du registre. On remarque que *tc_0* ne peut donc changer que lorsqu'il y a une réinitialisation ou un front montant d'horloge. La valeur de l'entrée du registre *tc_1* est spécifiée par la partie combinatoire du circuit (partie droite sur la figure 1.2)

Dans la partie combinatoire, toutes les affectations de signaux sont concurrentes, elles peuvent être écrites dans n'importe quel ordre.

L'affectation alm3 <= tc_0(7) signifie que la sortie *alm3* est connectée à la dernière cellule du tableau *tc_0* (qui est la sortie du registre).

Le reste des affectations concurrentes a pour but de finalement spécifier la nouvelle valeur pour *tc_0*, l'entrée du registre. Pour cela nous utilisons des signaux intermédiaires. Le signal *tc_6* correspond à un tableau réinitialisé : la première cellule (*tc_6*(0)) contient le jeton et toutes les autres sont vides (contiennent '0').

```
    tc_6(0) <= '1';
GEN1 : for k in 1 to 7 generate
    tc_6(k) <= '0';
end generate;
```

Le signal *tc_10* correspond à un tableau décalé à droite d'une cellule par rapport à *tc_0* : le jeton est décalé d'une cellule dans le tableau. Remarquons que dans le cas où le jeton se trouvait dans la dernière cellule, *tc_10* se retrouve vide de jeton.

```
    tc_10(0) ⇐ '0';
  GEN2 : for k in 1 to 7 generate
    tc_10(k) ⇐ tc_0(k - 1);
  end generate;
```

Le reste des affectations consiste à faire le choix entre les trois tableaux tc_0, tc_6 et tc_10 pour savoir lequel sera l'entrée du registre tc_1. La condition gd vaut 1 lorsque le maximum du compteur n'est pas atteint (not (tc_0(7))). On utilise un autre signal intermédiaire tc_8 qui est le tableau inchangé si le maximum du compteur est atteint ou qui vaut tc_10 (c'est à dire le tableau tc_0 décalé à droite) sinon. Enfin, la dernière affectation dit que l'entrée du registre sera tc_6 (remise à zéro du compteur) si le signal rst est reçu, et tc_8 sinon.

```
gd ⇐ not (tc_0(7));
tc_8 ⇐ tc_10 when gd='1' else tc_0;
tc_1 ⇐ tc_6 when rst='1' else tc_8;
```

1.8.2 ACL2

ACL2 est un prouveur de théorèmes basé sur la logique du premier ordre avec égalité et induction. Il possède un grand degré d'automatisation et possède des bibliothèques réutilisables de définitions de fonctions et de preuves de théorèmes [59]. ACL2 est également un langage de programmation basé sur Common Lisp. Ainsi, les modèles ACL2 sont à la fois exécutables et prouvables. Avant d'investir du temps dans la preuve, il est donc possible de vérifier le modèle sur quelques vecteurs de test. ACL2 a déjà été utilisée pour la vérification de systèmes électroniques [60].

Modèle ACL2 d'un modèle VHDL

On peut obtenir un modèle ACL2 automatiquement à partir d'un modèle VHDL en utilisant une méthode basée sur la simulation symbolique développée par l'équipe VDS du laboratoire TIMA [88]. Le modèle est simulé pour un cycle d'horloge – ce qui correspond en fait à plusieurs δ-cycles VHDL – de façon à extraire la fonction de transition d'un cycle à l'autre pour chaque sortie et chaque variable d'état du modèle. Le corps d'une fonction de transition est une expression conditionnelle, une expression arithmétique ou une expression booléenne. Les fonctions sont traduites en Lisp et utilisée pour définir la machine de Mealy de la description VHDL initiale. Les opérations VHDL sur les vecteurs de bits ou les booléens sont remplacées par les opérations correspondantes définies et prouvées être correctes en ACL2.

En plus de toutes ces fonctions, les informations de typage concernant les entrées et les variables d'états sont traduites en Lisp. Pour cela deux prédicats sont créés : hyp-input (input), qui établit le type de chaque entrée, et hyp-st (st), qui établit le type de chaque variable d'état.

L'état de la machine de Mealy est constitué de toutes les mémoires internes et de toutes les sorties du modèle. Un cycle d'horloge est modélisé par une fonction sim-step qui prend en paramètre les entrées du modèle et l'état de la machine au cycle d'horloge k, et produit l'état de la machine au cycle d'horloge $k+1$. Le corps de la fonction sim-step est la composition de toutes les fonctions de transition obtenues par simulation symbolique.

Sur la figure 1.3, nous donnons la fonction sim-step correspondant à l'exemple VHDL précédent, implantant un compteur.

Une fonction "nextsig_X" décrit le comportement du signal X pendant un cycle d'horloge. Par exemple, nous donnons le corps de la fonction "nextsig_alm3" sur la figure 1.3.

L'état général de la machine est défini comme une fonction récursive qui prend une séquence d'entrées l-input et un état initial st, et retourne l'état obtenu après avoir consommé toutes les entrées. La séquence l-input représente la liste de valeurs numériques ou symboliques que le modèle prend en entrée à chaque cycle d'horloge :

l-input = (inputs_cycle-1 inputs_cycle-2 ... inputs_cycle-k)

Lorsque la liste d'entrées est vide (vérifié par la fonction ACL2 atom), le calcul est terminé et la fonction retourne l'état st. Sinon, lorsque la liste d'entrées n'est pas vide, l'état suivant est calculé et st

FIG. 1.3 – Modèle ACL2 du compteur

```
(defun vhdl-counter (l-input st )
   (if (atom l-input) st
      (vhdl-counter (cdr l-input) (vhdl-sim-step (car l-input) st ) )))

(defun vhdl-sim-step (in st )
...
   (list (nextsig_tc_0 reset tc_1)
         (nextsig_tc_1 reset rst tc_1)
         (nextsig_tc_6)
         (nextsig_tc_8 reset tc_1)
         (nextsig_tc_10 reset tc_1)
         (nextsig_gd reset tc_1)
         (nextsig_alm3 reset tc_1))))

(defun nextsig_alm3 (reset tc_1) (nth 7 (if (equal reset 1) (list 1 0 0 0 0 0 0 0) tc_1)))
...
```

est mis à jour en appelant la fonction sim-step. Comme nous l'avons déjà mentionné, le modèle est à la fois prouvable et exécutable.

La fonction vhdl_counter, donnée sur la figure 1.3, modélise en ACL2 la fonction de la machine d'état correspondant à l'exemple VHDL donné dans la section précédente.

1.8.3 BHDL

Le langage BHDL sera décrit plus en détail dans le reste de ce document mais nous donnons ici un exemple à titre de comparaison avec VHDL et ACL2. Nous donnons le modèle BHDL du même exemple du compteur sur la figure 1.4. La machine $PARAM$ définit le type (i.e. l'ensemble) TYPETC comme une fonction $0..MAX \rightarrow BOOL$ et la constante MAX égale à 7 (nombre maximum sur 3 bits).

Dans un modèle BHDL, le circuit est représenté par un événement. Nous ne différencions pas les registres et les fils dans le modèle BHDL, ils sont tous représentés par des variables. La distinction se fera lors de la traduction vers un langage de description de circuits.

Les entrées, les sorties et les variables du modèle sont respectivement déclarées dans les clauses INPUTS, OUTPUTS et VARIABLES de la machine BHDL. Les types de tous ces éléments sont donnés dans la clause INVARIANT, le type BOOL est déclaré dans la machine $BHDL$ (clause SEES). La clause INITIALISATION décrit la manière dont les variables sont réinitialisées lorsque le signal de réinitialisation asynchrone est activé. Des substitutions sont données pour les entrées et les sorties dans la clause INITIALISATION car toute variable doit être initialisée en B. Cependant ces substitutions sont non déterministes car l'activation du signal de réinitialisation n'a pas de conséquence directe sur les entrées et les sorties. Il a en fait une conséquence indirecte sur les sorties lorsque celles-ci dépendent de registres qui sont réinitialisés. Cette dépendance est modélisée dans la clause OPERATIONS. Cette clause est la clause principale, elle décrit le circuit. Elle est composée d'un seul événement qui décrit l'évolution du circuit pendant un cycle d'horloge. Les entrées sont lues et les sorties sont écrites.

Une description complète de la façon dont ce modèle est obtenu est donnée dans la section 3.5 page 48. Pour l'affectation du tableau tc nous utilisons des λ-expressions puisque le tableau est modélisé par une fonction en B.

En comparant le modèle BHDL avec les modèles VHDL ou ACL2 ont peut constater que la description BHDL est plus compacte. D'autre part il n'est pas fait distinction entre registres et fils, ce sont tous des variables en BHDL. L'avantage est qu'on ne manipule qu'un seul type d'élément. L'inconvénient est que cela peut nécessiter un travail intellectuel de la part de la personne en charge de la modélisation si elle souhaite maîtriser quels doivent être les registres et quels doivent être les fils du circuit. On peut dire

FIG. 1.4 – Modèle BHDL d'un compteur

```
MACHINE
  counter
SEES
  BHDL, PARAM
INPUTS
  rst
OUTPUTS
  alm3
VARIABLES
  tc
INVARIANT
  rst ∈ BOOL ∧ alm3 ∈ BOOL ∧ tc ∈ TYPETC
INITIALISATION
  rst :∈ BOOL||
  alm3 :∈ BOOL||
  tc := λk.(k = 0|TRUE) ∪ λk.(k : 1..MAX|FALSE)
OPERATIONS
  BEGIN
    alm3 := tc(MAX)
    ;
    IF rst = TRUE THEN
      tc := λk.(k = 0|TRUE) ∪ λk.(k : 1..MAX|FALSE)
    ELSE
      IF tc(MAX) = FALSE THEN
        tc := λk.(k = 0|FALSE) ∪ λk.(k : 1..MAX|tc(k − 1))
      END
    END
  END
END
```

qu'une description BHDL se situe entre la description VHDL et la description ACL2 : elle est à la fois fonctionnelle mais permet aussi d'avoir une idée de l'architecture. Elle est fonctionnelle car les expressions utilisées (comme les λ-expressions) tiennent plus de la programmation fonctionnelle que de la description de circuits. Elle donne une idée de l'architecture en observant les compositions qui sont faites entre les substitutions simples : par exemple en observant l'imbrication des *IF*s (compositions conditionnelles) ou l'utilisation du ; (composition séquentielle). Ce n'est pas le cas dans cet exemple, mais on peut aussi utiliser une composition parallèle, ce qui permet d'identifier des blocs de code qui peuvent être implantés séparément.

1.8.4 Comparaison des trois styles de modélisation

Ces trois styles de modélisation ont chacun leurs avantages et leurs inconvénients. Le langage VHDL a été créé spécialement pour la modélisation de circuits, il permet donc de modéliser différents types de circuits, y compris des circuits qui ne sont pas synchrones. Par contre on ne peut pas faire de preuves directement sur un modèle VHDL. La correction sur un modèle VHDL ne peut être obtenue que par un test exhaustif.

Une description ACL2 a l'avantage d'être à la fois prouvable et exécutable. Le prouveur possède déjà un nombre de bibliothèques réutilisables, facilitant ainsi la preuve. Cependant , l'utilisation du langage Lisp rend difficile la lecture d'un modèle dans ce langage et la preuve est réservée aux experts en ACL2.

Par rapport à VHDL, le langage BHDL est synchrone et ne peut donc pas modéliser tous les types de circuits. On ne peut modéliser que les circuits synchrones. L'écriture d'une description BHDL nécessite

cependant une expertise dans la méthode B, puisque le modèle est obtenu à la suite d'un processus de raffinement B. On ne peut pas faire de preuve directement sur le modèle BHDL, seulement sur les modèles B dont il est issu.

1.9 Conclusion

Nous avons introduit dans cette section les principaux concepts liés aux circuits électroniques. Nous avons expliqué ce qu'est un circuit combinatoire, un circuit synchrone ou un circuit asynchrone. Nous avons donné un bref rappel historique des trois principaux langages de descriptions de circuit que sont VHDL, Verilog et SystemC. Enfin nous avons donné un exemple de circuit que nous avons modélisé de trois façons différentes (en VHDL, ACL2 et BHDL) de façon à comparer ces approches différentes de la modélisation de circuits.

Chapitre 2

Méthodes formelles

L'objectif de ce chapitre est de présenter différentes méthodes formelles utilisées aujourd'hui dans le développement ou la vérification de systèmes électroniques.

Les méthodes formelles sont des techniques, en général basées sur la logique, qui ont été mises au point afin de raisonner de façon rigoureuse sur des descriptions de programmes ou de systèmes électroniques. Elles donnent un cadre formel dans lequel la correction du système modélisé peut être assurée. Au cours des années, différentes approches ont été développées.

Une approche formelle peut s'appuyer sur trois aspects :

- Un langage doté d'une sémantique formelle bien définie pour décrire un modèle du système étudié de façon non ambiguë. La modélisation peut se faire de façon abstraite pour établir une spécification sans entrer dans les détails de d'implantation, ou de façon plus concrète pour décrire des algorithmes. Les différences entre un langage de programmation ou de description de circuit et un langage formel sont : d'une part l'existence d'une sémantique formelle bien définie du langage et d'autre part l'existence d'outils, ou au minimum de théories, permettant de tirer partie de l'aspect formel du langage. Par exemple, bien qu'on puisse donner des sémantiques formelles à des langages de programmation, ceux-ci sont généralement plus orientés vers une approche d'efficacité du logiciel obtenu (utilisation de pointeurs en C++, outils d'optimisation du code, compilateurs dédiés à un processeur donné, types de données simples...) alors qu'un langage formel doit donner la possibilité de raisonner de manière formelle sur lui. En conséquence de quoi, les langages formels, bien que pouvant souvent manipuler des objets mathématiques beaucoup plus complexes qu'un langage de programmation, sont souvent plus "simples" de façon à permettre un raisonnement humain plus aisé et un support plus facile par des outils logiciels.

- Un langage pour exprimer des propriétés sur le système étudié. Ces langages sont généralement basés sur la logique mais cherchent parfois à s'en éloigner pour ne pas trop "effrayer" les utilisateurs potentiels. Le type de propriété qui peut être exprimé dépend du langage utilisé. Par exemple la logique classique peut permettre de spécifier des propriétés invariantes (propriétés qui doivent être valides en tout point de l'évolution du système) alors que des logiques plus expressives comme LTL ou CTL peuvent permettre de spécifier des propriétés temporelles (propriétés dont la validité à un instant donné dépend de l'état dans lequel se trouve le système à cet instant).

- Une méthode, une théorie, pouvant définir (1) de quelle manière écrire des modèles, (2) quelles propriétés peuvent être écrites sur ces modèles et (3) de quelle manière elles peuvent être vérifiées. Il faut également des outils supportant cette méthode : gestion de la bibliothèque de modèles, génération des propriétés de bonne formation des modèles, outils permettant la vérification des propriétés (prouveur, model-checker, ...). Sans outil support, il n'y a pas de passage à l'échelle possible.

2.1 Langages de modélisation et d'expression de propriétés

Il existe de très nombreux langages dans lesquels ont peut exprimer des modèles de circuits électroniques. Certains langages sont plus formels que d'autres. Le langage le plus répandu est VHDL ; dans le même style on peut citer Verilog, Handle-C [66] ou SystemC. Bien que SystemC ne soit qu'une bibliothèque ajoutée au dessus de C++, ces quatre langages sont dédiés à la modélisation de circuits électroniques, en proposant toutes les primitives permettant la modélisation des objets physiques d'un circuit électronique.

Dans les langages plus formels on peut citer Lustre, Esterel ou les StateCharts [39]. On peut également citer ACL2 [14, 82, 19] qui a entre autre été utilisé pour la vérification d'un algorithme de division proche de celui implanté dans les processeurs AMD. Au même titre qu'ACL2, d'autres langages, comme Coq [32], HOL [43] ou PVS [85] ont été utilisés pour la modélisation et la vérification de circuits bien que ce ne soient pas des langages dédiés à cela.

Le langage B a déjà été utilisé, avant cette thèse, pour modéliser des circuits électroniques. Il a été utilisé par exemple par Abrial dans [4] pour modéliser un arbitre et un contrôleur de feu rouge, par Cansell et Mery dans [26] pour modéliser un additionneur et un multiplicateur. La thèse de Cyril Proch [79], réalisée en parallèle à celle-ci, a également consisté à utiliser la méthode B pour la modélisation de systèmes électroniques, notamment pour la production d'un outil de mesure pour une norme (ETSI TR 101 290) dans le domaine de la télévision numérique terrestre. On peut également citer les travaux de Boulanger, Aljer, Mariano, Devienne et Tison dans [20, 12] ainsi que les travaux de Sørensen dans [84]. Dans le premier cas, le principe est de prendre des bibliothèques de base de VHDL et de les traduire en machine B ; des circuits plus complexes sont alors obtenus en combinant des éléments de plus bas niveau. C'est l'approche opposée à celle que nous avons adoptée puisque nous partons d'une spécification abstraite et raffinons le modèle pour obtenir une description implantable. Dans le deuxième cas, la modélisation B est aussi conçue à partir des concepts de VHDL mais à un niveau d'abstraction plus élevé, en utilisant des *processus concurrents*, modélisés par des machines indépendantes, et des *événements* (au sens VHDL). Le style de modélisation est de décrire un ensemble de processus concurrents, avec la restriction qu'une variable ne peut être écrite que par un seul processus ; chaque processus étant un programme séquentiel développé de manière traditionnelle en B. Des bibliothèques sont ajoutées pour décrire plus facilement des concepts liés à la modélisation de circuits (comme les signaux ou les événements).

On peut classifier les langages d'expression de propriétés en deux grandes catégories : les langages basés sur la logique du premier ordre ou d'ordre supérieur, et les langages basés sur des logiques temporelles.

Le langage choisi pour exprimer les propriétés est généralement directement lié au choix du langage choisi pour la modélisation du circuit. Les outils supports déterminent quels langages de propriétés sont utilisés pour un langage de modélisation donné. La plupart des langages de modélisation cités plus haut utilise des langages classiques de la logique temporelle (LTL, CTL) ou non (logique des prédicats). Parmi les langages d'expressions de propriétés développés pour les circuits électroniques, on peut citer PSL/Sugar [53] développé par IBM ou le langage *e* [52] développé par Verisity (maintenant partie de Cadence).

2.2 Outils supports

Les trois principales techniques de vérification sont l'équivalence de modèles, le model-checking et la preuve de théorème. Le model-checking consiste à vérifier une propriété donnée par énumération des états accessibles du système. La preuve de théorème se base sur une axiomatisation de la logique à partir de laquelle on essaie de dériver la preuve de la propriété souhaitée (de façon automatique ou avec l'aide de l'utilisateur).

2.2.1 Équivalence de modèles

Dans le flot traditionnel de conception d'un circuit électronique, un circuit est d'abord modélisé dans un langage comme VHDL à des niveaux relativement élevés d'abstraction (comparable à du logiciel) puis raffiné (le plus souvent informellement) jusqu'au niveau RTL. À partir de ce niveau on utilise un outil de synthèse automatique pour produire un "netlist" du circuit. Ce netlist peut ensuite être optimisé,

certaines parties peuvent être remplacées automatiquement par des composants plus efficaces, et peuvent même être éventuellement manipulées à la main. Il se pose alors la question de la correspondance entre la description RTL initiale et la description par netlist finale qui servira à fabriquer le circuit physique.

L'utilisation des outils automatiques devrait donner une bonne confiance dans l'équivalence entre la description RTL et la description netlist, mais on sait que les programmes informatiques ne sont pas exempts de bugs. Des outils ont donc été mis au point pour permettre la vérification d'équivalence entre deux modèles, ils se basent généralement sur les BDDs ou les solveurs SAT. Les deux principaux outils sont *Conformal* de Cadence [22] et *Formality* de Synopsys [86].

2.2.2 Model-checking

Une approche théorique du model-cheking est la suivante. À partir de la description initiale du système, celui-ci est décrit sous forme d'un graphe orienté dont les nœuds sont les états accessibles et les transitions représentent les évolutions possibles du système d'un état vers un autre. La négation de la propriété que l'on souhaite vérifiée est elle-même représentée par un graphe orienté, ce graphe représentant l'ensemble des évolutions du système qui satisfont la négation de la propriété. La vérification de la validité de la propriété, consiste à calculer le produit cartésien synchrone des deux graphes précédemment construits. Si le langage reconnu par le produit cartésien des deux graphes est vide, cela prouve qu'il est impossible de trouver une évolution du système qui satisfasse la négation de la propriété à valider, la propriété est donc vérifiée. Si le langage reconnu par le produit cartésien synchrone des deux graphes n'est pas vide, la propriété n'est pas vérifiée et les mots de ce langage sont des contre-exemples de la propriété.

En pratique, il n'est pas possible de construire réellement le graphe du système car il y a une explosion exponentielle du nombre des états accessibles, et donc de la taille du graphe. Les techniques de model-cheking consistent en pratique dans une énumération astucieuse (selon les algorithmes) de l'espace des états accessibles, permettant de limiter les ressources en mémoire et en temps de calcul. L'efficacité des méthodes basées sur le model-checking reste en général dépendante de la taille de l'espace des états accessibles (qui doit être fini) et trouve donc ses limites dans les ressources en mémoire de l'ordinateur. Des techniques d'abstraction (éventuellement guidées par l'utilisateur) peuvent être utilisées pour améliorer l'efficacité des algorithmes.

Les propriétés qui sont vérifiées par model-checking sont souvent des propriétés temporelles. Il existe différentes logiques temporelles, chacune ayant un pouvoir d'expression plus ou moins important, et chacune mobilisant des ressources plus ou moins importantes pour le model-checking.

SMV

Le système SMV [67] est un outil de vérification de systèmes d'états finis par rapport à une spécification écrite dans la logique temporelle CTL. Le langage d'entrée de SMV permet la description de systèmes d'états finis qu'ils soient synchrones ou asynchrones. Le langage permet des descriptions hiérarchiques et la définition de composants réutilisables. Comme le langage est destiné à la description de machines d'états finis, les types de données sont également finis : booléens, scalaires, vecteurs de longueur fixe, et des types structurés. La logique CTL permet l'expression d'une riche classe de propriétés temporelles de manière concise, incluant des propriétés de sûreté, de vivacité et d'absence de blocage. SMV utilise l'algorithme de model-checking symbolique basé sur les OBDD (Ordered Binary Decision Diagrams). A titre d'illustration, on donne ci-dessous un petit exemple exprimé dans le langage de SMV.

```
MODULE main
VAR
  request : boolean;
  state : {ready,busy};
ASSIGN
  init(state) := ready;
  next(state) := case
                   state = ready & request : busy;
                   1 : {ready,busy};
                 esac;
```

```
SPEC
  AG(request -> AF state = busy)
```

La partie `VAR` déclare l'ensemble des variables avec leurs types. La partie `ASSIGN` donne l'initialisation et la fonction de transition de la machine. La partie `SPEC` donne les propriétés temporelles que l'on souhaite vérifier sur le modèle. La ligne `init(state) := ready` définit l'initialisation de la variable `state` qui est initialisée à la valeur `ready`. L'expression `next(state)` introduit la fonction de transition de la variable `state`. La valeur de l'expression `case` est déterminée par la première expression sur la partie droite d'un `:`. Ainsi, si `state=ready` et que la variable `request` est vraie alors la valeur de l'expression est `busy`. Sinon la valeur de l'expression est `{ready,busy}`. Lorsqu'on affecte un ensemble à une variable, le résultat est non déterministe. Notons que la variable `request` n'est pas affectée dans ce programme. Ceci laisse le système SMV libre de choisir n'importe quelle valeur, ce qui donne à cette variable la caractéristique d'une entrée non contrainte du système.

La spécification du système apparaît comme une formule CTL sous le mot clef `SPEC`. Le système SMV vérifie que tous les états initiaux du modèle satisfassent la spécification. Dans notre exemple, la spécification est que, invariablement (`AG`), si `request` est vraie alors, inévitablement (`AF`), la variable `state` sera `busy`.

RuleBase

RuleBase [17] est un outil de vérification formelle développé par le laboratoire de recherche d'IBM à Haifa. Il s'agit d'une version augmentée de l'outil SMV et de son moteur de vérification. RuleBase supporte plusieurs langages de description de circuits comme VHDL ou Verilog. Le modèle de l'environnement est écrit dans le langage de RuleBase, un dialecte de SMV.

Le model-checker SMV utilise CTL comme langage de spécification alors que RuleBase utilise le langage de spécification Sugar, permettant aux modeleurs de circuits qui ne sont pas experts en CTL d'exprimer des propriétés. Sugar est construit au dessus de CTL et inclut des constructeurs supplémentaires.

Lustre

Lustre [46] est un langage flot de données synchrone : les variables et les expressions sont considérées comme représentant les séquences de valeurs qu'elles prennent pendant le temps d'exécution du programme et les opérateurs Lustre opèrent sur ces séquences. La nature synchrone du langage vient de l'hypothèse que toutes les variables et expressions du programme prennent leurs n-ième valeurs au même instant. Un programme Lustre est un ensemble d'équations du type `X=E` où `X` est une variable et `E` une expression ; cette expression signifie que les séquences de valeurs pour `X` et `E` sont les mêmes, c'est-à-dire qu'à chaque cycle du programme, `X` et `E` prennent la même valeur. Le langage Lustre propose également deux opérateurs particuliers sur les séquences :

- un opérateur permettant de se référer à la valeur précédente d'une expression, c'est l'opérateur `pre(E)`
- un opérateur permettant de donner une valeur initiale à une séquence et de donner le reste de sa séquence, c'est l'opérateur `E -> F` où `E` est la valeur initiale de la séquence et `F` le reste de la séquence (on a $(E \rightarrow F)_0 = E_0$ et pour tout i supérieur ou égal à 1 $(E \rightarrow F)_i = F_i$).

On donne ci-dessous un exemple d'utilisation de ces deux opérateurs. Cette déclaration définit un nœud (nom d'un "module" en Lustre) qui retourne *true* chaque fois que son paramètre booléen passe de *false* à *true*.

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

Pour illustrer la façon d'exprimer des propriétés temporelles, nous prenons l'exemple d'un "demi-tour" pour un métro utilisé par Halbwachs, Lagnier et Ratel dans [47]. Une section de demi-tour est composée

de trois rails A, B et C, et d'un commutateur S permettant de connecter le rail B soit à A soit à C. Pour faire demi-tour, un métro arrive par le rail A, entre dans le rail B (à condition que le commutateur soit bien positionné), puis, une fois le commutateur de nouveau bien positionné, passe du rail B au rail C. On définit neuf variables :

- `ack_AB`, resp. `ack_BC`, indique si le commutateur connecte A à B, resp. B à C.
- `on_A`, `on_B` et `on_C` qui sont trois capteurs, un pour chaque rail
- `do_AB` et `do_BC` qui sont les commandes pour le commutateur
- `grant_access` et `grant_exit` qui sont les commandes pour gouverner les feux pour les métros. Le premier permet au métro de passer du rail A au rail B et le deuxième de passer de rail B au rail C. Dans les deux cas, on doit garantir que le commutateur est bien positionné.

Nous donnons ci-dessous le nœud Lustre `UMS` modélisant cet exemple. Nous ne détaillons pas plus les explications sur les expressions utilisées, rappelons qu'il n'y a aucun ordre entre les équations d'un nœud.

```
node UMS(on_A,on_B,on_C,ack_AB,ack_BC: bool)
        returns (grant_access,grant_exit, do_AB,do_BC: bool);
var empty_section, only_on_B: bool;
let
  grant_access = empty_section and ack_AB;
  grant_exit = only_on_B and ack_BC;
  do_AB = not_ack_AB and empty_section;
  do_BC = not_ack_BC and only_on_B;
  empty_section = not(on_A or on_B or on_C);
  only_on_B = on_B and not(on_A or on_C);
tel
```

Les propriétés temporelles que l'on souhaite vérifier sur un programme Lustre sont elles mêmes exprimées comme un programme Lustre. Il s'agit en fait de les exprimer comme des conditions booléennes qui doivent être toujours vraies. Pour simplifier l'écriture de ces propriétés on peut définir des nœuds Lustre qui correspondent aux opérateurs de base permettant d'exprimer des propriétés temporelles. Nous en donnons cinq ci-dessous.

- Le nœud `implies` implante l'implication logique ordinaire :

```
node implies(A, B: bool) returns (AimpliesB: bool);
let
  AimpliesB = not A or B;
tel
```

- Le nœud `after` retourne la valeur *false* jusqu'à ce que son entrée soit *true* et reste ensuite à *true* pour toujours :

```
node after(A: bool) returns (afterA: bool);
let
  afterA = false -> pre(A or afterA);
tel
```

- Le nœud `always_since` a deux entrées et retourne *true* si et seulement si la première entrée a valu continuellement *true* depuis la dernière fois que la seconde entrée ait valu *true* :

```
node always_since(B,A: bool) returns (alwaysBsinceA: bool);
let
 alwaysBsinceA = if A then B
              else if after(A) then B and pre(alwaysBsinceA)
              else true;
tel
```

- Le nœud `once_since` a deux entrées et retourne *true* si et seulement si sa première entrée a valu au moins une fois *true* depuis la dernière fois que sa seconde entrée ait valu *true* :

```
node once_since(C,A: bool) returns (onceCsinceA: bool);
let
  onceCsinceA = if A then C
                else if after(A) then C or pre(onceCsinceA)
                else true;
tel
```

- Le nœud `always_from_to` prend trois paramètres booléens A, B, C, et retourne une sortie X telle que X est toujours vraie si et seulement si la propriété suivante est vérifiée :

```
node always_from_to(B,A,C: bool) returns (X: bool);
let
 X = implies(after(A), always_since(B,A) or once_since(C,A));
tel
```

Avant de donner le programme Lustre pour la vérification du demi-tour du métro, il nous faut modéliser l'environnement. Ceci se fait toujours en Lustre en faisant des hypothèses sur l'environnement (hypothèses nécessaires pour garantir la correction du programme Lustre) en utilisant le mécanisme d'assertion. Par exemple on dit que le commutateur ne peut pas connecter à la fois A avec B et B avec C en écrivant :

```
assert not(ack_AB and ack_BC)
```

En utilisant les principes que nous avons évoqués précédemment, nous pouvons écrire le programme Lustre qui sert à la vérification :

```
node UMS_verif(on_A,on_B,on_C, ack_AB,ack_BC: bool)
         returns(property: bool);
var
  grant_access,grant_exit: bool;
  do_AB,do_BC: bool;
  no_collision,exclusive_req: bool;
  no_derail_AB,no_derail_BC: bool;
  empty_section, only_on_B: bool;
let

  empty_section = not(on_A or on_B or on_C); only_on_B = on_B and not(on_A or on_C);

-- ASSERTIONS
  assert not(ack_AB and ack_BC);
  assert always_from_to(ack_AB,ack_AB,do_BC)
         and always_from_to(ack_BC,ack_BC,do_AB);
  assert empty_section -> true;
  assert true -> implies(edge(not empty_section), pre grant_access);
  assert true -> implies(edge(on_C), pre grant_exit);
  assert implies(edge(not on_A),on_B);
  assert implies(edge(not on_B), on_A or on_C);

-- UMS CALL
  (grant access,grant_exit,do_AB,do_BC) =
            UMS(on_A,on_B,on_C,ack_AB,ack BC);

-- PROPERTIES
  no collision= implies(grant_access,empty_section);
  exclusive req = not(do_AB and do_BC);
  no_derail_AB = always_from_to(ack_AB, grant_access, only_on_B);
  no_derail_BC = always_from_to(ack_BC, grant_exit, empty_section);
  property = no_collision and exclusive_req and no_derail_AB and no_derail_BC;
tel
```

Le problème de vérification se résume alors à prouver que la sortie `property` du programme `UMS_verif` est toujours vraie, sous les conditions des assertions données. Deux moteurs de vérifications ont été implantés et intégrés dans un outil de vérification appelé Lesar [90]. Le premier énumère explicitement les états atteignables, comme cela se fait dans les model-checkers traditionnels. Le deuxième moteur travaille symboliquement : on part d'une formule booléenne F_0, caractérisant l'ensemble des états où la sortie est vraie (en Lustre c'est l'expression de `property`), et on calcule itérativement F_1, F_2, ..., F_n, où F_{i+1} caractérise l'ensemble des états qui conduisent à F_i. Si l'algorithme converge (car l'ensemble des états est fini) vers une formule F alors la propriété est vraie.

On peut également faire de la preuve de théorème en utilisant l'outil GLOUPS [89] qui utilise PVS ou de l'interprétation abstraite avec l'outil Nbac [91].

2.2.3 Preuve de théorème

La preuve de théorème s'appuie sur une logique définit de manière axiomatique et pourvue de règles d'inférences à partir de ces axiomes, permettant de déduire des propriétés plus complexes à partir des axiomes de base. En général, un prouveur de théorème définit des règles d'inférence supplémentaires travaillant sur des objets plus complexes que ceux définis par les axiomes de la logique sous-jacente. Par exemple AtelierB possède un ensemble de règles d'inférence sur les ensembles, évitant ainsi d'avoir à définir les ensembles à partir des axiomes, facilitant ainsi les preuves.

Réaliser une preuve avec un prouveur de théorème consiste à lui fournir un certains nombres d'hypothèses et une conclusion à prouver sous ces hypothèses. Les preuves peuvent se faire de deux façons, soit de manière automatique, soit de manière interactive. La preuve automatique consiste simplement à donner l'ordre au prouveur de théorème de tenter de prouver la conclusion sous les hypothèses données sans l'action de l'utilisateur. C'est donc la solution la plus simple pour l'utilisateur. Malheureusement la preuve automatique échoue souvent. Rappelons que, par exemple, la logique classique n'est pas décidable (à partir du moment où elle inclue l'arithmétique, ce qui est le cas en général) : c'est-à-dire qu'il n'existe pas d'algorithme permettant de décider si une conjecture est vraie ou fausse. D'une manière ou d'une autre, il est donc souvent nécessaire d'avoir une interaction entre le prouveur de théorème et l'utilisateur. D'une manière générale, cette interaction peut se faire de deux façons : soit en définissant des lemmes intermédiaires plus simples à démontrer et que le prouveur de théorème pourra réutiliser, soit en demandant à l'utilisateur, par l'intermédiaire d'une interface de preuve, d'effectuer lui-même certains pas de la preuve. Même dans le dernier cas, il est rare que l'utilisateur ait à effectuer la preuve en entier : après avoir décomposé le théorème en théorèmes plus simples, il suffit souvent de faire un appel au prouveur automatique pour terminer la preuve.

Coq

Coq est un outil de preuve développé à l'INRIA-Rocquencourt [57]. Il propose un langage typé très riche et très expressif, une logique constructive d'ordre supérieur et la possibilité d'extraire automatiquement des programmes fonctionnels à partir de certaines parties des preuves. Il a été utilisé par exemple dans [31, 76] pour prouver la correction d'un multiplicateur. Une vérification d'un multiplicateur au niveau vecteur de bits peut être trouvée dans [13]. Dans [32], les auteurs essaient d'exploiter la possibilité d'extraction de programme à partir des preuves sur une étude de cas concernant un comparateur de deux nombres entiers. Nous reprenons cet exemple pour montrer le style de modélisation d'un circuit électronique en Coq.

La section `dependent_lists` définit la notion de liste dépendante d'un ensemble `A`. Notons qu'en Coq, la lettre `O` correspond à l'entier 0 et l'expression `(S n)` signifie le successeur de n. La définition d'une liste se fait de manière inductive en définissant les deux constructeurs `nil` et `cons`. Le constructeur `nil` construit la liste vide (liste de longueur 0). Le constructeur `cons` prend un élément de l'ensemble `A` et une liste de longueur n pour renvoyer une liste de longueur $n+1$.

```
Section dependent_lists .
Variable A:Set.
Inductive list :nat->Set:= nil:(list O)|
                           cons:(n:nat)A->(list n)->(list (S n)).
```

```
...
End dependent_lists.
```

La section `numerals` définit un ensemble de termes permettant la représentation des nombres. Si par exemple on choisi $BASE = 2$, la représentation des nombres est binaire. Un nombre (type `num`) est une liste de chiffres (type `digit`). La fonction `Val`, définie inductivement, associe l'entier qui correspond à une liste de chiffres donnée.

```
Section numerals.
Require dependent_lists.

Definition BT:={b:nat|(lt O b)}.
Variable BASE:BT.
Definition base:=(Inj nat [b:nat](lt O b) BASE).
Definition digit:={x:nat|(lt x base)}.
Definition val:digit->nat:=(Inj nat [x:nat](lt x base)).
Definition num:=(list digit).
Local Cons:=(cons digit).
Local Nil:=(nil digit).

Fixpoint Val[n:nat;X:(num n)]:nat:=<[m:nat]nat>Case X of
(*X=Nil*)          O
(*X=(Cons p d D)*) [p:nat][d:digit][D:(num p)]
                       (plus (mult (val d) (exp base p)) (Val p D))
end.
...
End numerals.
```

La section `systolic_lists` donne une définition générique pour la connexion de cellules identiques ayant chacune quatre ports (entrées/sorties). On remarqua qu'il n'y a pas de distinction faite entre les entrées et les sorties des cellules connectées dans cette définition. On peut interpréter cette définition inductive de la façon suivante : la base est une liste vide (`(nil B)` et `(nil C)`), la partie inductive dit que d'ajouter une cellule avec les ports (a b c a1) à une liste de connexions qui a les ports (a1 lb lc a') produit une liste de connexions dont les ports sont (a b@lb c@lc a'). Nous avons utilisé abusivement la notation e@l pour désigner l'ajout d'un élément e en tête d'une liste l.

```
Section systolic_lists.
Variables A,B,C:Set.
Variable cell:A->B->C->A->Prop.

Inductive connection :(n:nat)A->(list B n)->(list C n)->A->Prop:=
C_O :(a:A)(connection O a (nil B) (nil C) a) |
C_Sn:(n:nat)(a,a1,a':A)(b:B)(c:C)(lb:(list B n))(lc:(list C n))
 (cell a b c a1)->(connection n a1 lb lc a')->
 (connection (S n) a (cons B n b lb) (cons C n c lc) a').

End systolic_lists.
```

On définit l'ensemble `order` constitué de trois éléments : L pour "inférieur", E pour "égal" et G pour "supérieur". La fonction `comparaison` compare deux nombres entiers. Elle utilise la fonction prédéfinie `Lt_eq_Gt`.

```
Inductive order:Set:=L:order|E:order|G:order.

Definition comparison:=[v1,v2:nat]<order> Case (Lt_eq_Gt v1 v2) of
          [_(lt v1 v2)]          L
          [_ v1=v2]              E
          [_(gt v1 v2)]          G                        end.
```

La section `comparator` définit la spécification (`Specif`) et l'implantation (`Comparator`) du comparateur. La spécification utilise directement des nombres entiers alors que l'implantation travaille sur des listes (`Num`) de chiffres et une connexion de n cellules comparatives de bases (`cell`).

```
Section comparator.

(*system of numeration*)
Variable BASE: BT.
Local Digit:=(digit BASE).
Local ValB:=(Val BASE).
Local Num:=(num BASE).

(*semantics of the cells*)
Local f_cell:order->Digit->Digit->order:=
[o,x,y]<order>Case o of
(*o=L*) L
(*o=E*) (comparison (valB x) (valB y))
(*o=G*) G                                    end.

Definition cell:order->Digit->Digit->order->Prop:=
[o, x, y, o'] o'=(f_cell o x y).

(*structure of the comparator*)

Local Connection:=(connection order Digit Digit cell).
Local Comparator:=[n:nat][o:order][X,Y:(Num n)](Connection n E X Y o).


(*behaviour of the comparator*)

Local Specif:(n:nat)(inf n)->(inf n)->order:=[n,X,Y]
(comparison (val_inf n X) (val_inf n Y)).
```

Le théorème de correction de l'implantation vis-à-vis de la spécification s'écrit sous la forme donnée ci-dessous.

```
Remark correctness:(n:nat)(X,Y:(Num n))(o:order)
(Comparator n o X Y)-> o=(Specif (exp base n) (Val_bound n X) (Val_bound n Y)).
```

PVS

PVS (Prototype Verification System), développé par SRI International [62], consiste en un langage de spécification intégré avec un outil support et un prouveur de théorème qui sont distribués gratuitement pour un usage non commercial. Le langage de spécification de PVS est une logique d'ordre supérieur avec typage. Son prouveur de théorème est à la fois interactif et très mécanisé : l'utilisateur choisi chaque pas de preuve qui doit être appliqué et PVS les applique, affiche le résultat et attend une autre commande. PVS se différencie d'autres prouveurs interactifs par la puissance des pas de preuve : on peut invoquer des procédures de décision pour l'arithmétique ou l'égalité, de simplification propositionnelle basé sur des BDDs, d'induction et d'autres larges unités de déduction. Les papiers [72, 85, 33] présentent comment utiliser PVS pour la vérification de circuits électroniques. Dans [34], Cyrluck, Rajan, Shankar et Srivas nous montre comment utilisé PVS pour la vérification d'un microprocesseur. Nous reprenons un des exemples qui sont donnés par Srivas, Rueß et Cyrluk dans [85] pour donner une brève présentation de la modélisation en PVS. Cet exemple est un additionneur conçu à partir de cellules réalisant l'addition sur 1 bit.

La théorie `full_adder` définit, par la fonction `FA`, un additionneur 1-bit qui prend en entrée 2 bits plus une retenue entrante et renvoie en sortie la somme de ces 3 bits plus une retenue sortante. La notation `[# carry, sum: bit #]` dénote un type aggloméré (type "record") avec les deux champs `carry` et

sum. La théorie `full_adder` définit également un lemme attestant la correction de la fonction `FA` en disant que la somme des deux valeurs d'entrée `x`, `y` et de la retenue entrante `cin` est égale à la valeur de la retenue sortante multipliée par 2 plus la valeur de la somme de sortie.

```
full_adder: THEORY
BEGIN

  IMPORTING bitvectors@bit

  x, y, cin: VAR bit

  FA(x,y,cin): [# carry, sum: bit #] =
    (# carry := (x AND y) OR ((x XOR y) AND cin),
       sum := (x XOR y) XOR cin #)

  FA_corr: LEMMA
    sum(FA(x, y, cin)) = x + y + cin - 2 * carry(FA(x, y, cin))

END full_adder
```

Nous pouvons maintenant définir un additionneur N-bits en réutilisant l'additionneur 1-bit définit précédemment. Ceci est fait par la théorie `cpa` qui décrit l'implantation et le théorème de correction de l'additionneur. Cette théorie est paramétrée par la longueur `N` des vecteurs de bits d'entrée et de résultat.

Le bit de retenue qui se propage à travers les additionneurs 1-bit est défini récursivement par la fonction `nth_cin`. La fonction `cpa` définit la retenue sortante comme étant la `N`-ième retenue, et la `n`-ième somme de sortie est définie comme étant la somme de sortie du `n`-ième additionneur 1-bit appliqué aux `n`-ièmes bits des vecteurs de bits d'entrée et à la `n`-ième retenue.

```
cpa[N: posnat]: THEORY
BEGIN

  IMPORTING full_adder, bitvectors@bv[N], bitvectors@bv_nat

  xv, yv: VAR bvec[N]; n: VAR below[N]; j: VAR upto[N]

  nth_cin(j, xv, yv): RECURSIVE bit =
    IF j = 0 THEN 0
    ELSE carry(FA(xv(j - 1), yv(j -1),
                  nth_cin(j - 1, xv, yv))) ENDIF MEASURE j

  cpa(xv, yv): [# carry: bit, sum: bvec[N] #] =
    (# carry:= nth_cin(N, xv, yv),
       sum := LAMBDA n: sum(FA(xv(n), yv(n), nth_cin(n,xv,yv)))
     #)

  cpa_char: THEOREM
    sum(cpa(xv, yv)) = xv + yv - exp2(N) * carry(cpa(xv, yv))
END cpa
```

Le théorème `cpa_char` exprime la correction conventionnelle d'un additionneur au niveau du vecteur de bits, et relie le niveau porte de l'implantation (cf. additionneur 1-bit) avec le niveau fonctionnel.

2.3 Autres approches utilisant la méthode B

Les papiers [20, 12] décrivent une approche de la modélisation de circuits électroniques en utilisant la méthode B. Le langage utilisé n'est pas le B événementiel mais le B "classique" : une machine B contient

un ensemble de variables et définit un ensemble de fonctions qui peuvent être appelées de l'extérieur. La modélisation de l'environnement du circuit ne fait donc pas partie intégrante du modèle du circuit (à moins d'écrire une machine dédiée à la modélisation de l'environnement. Dans notre approche, présentée dans le reste de ce document, nous utilisons le langage B événementiel ; dans cette approche la modélisation de l'environnement est nécessaire car le système est considéré fermé et risque donc de se bloquer si l'environnement n'est pas modélisé.

L'approche présentée dans [20] est appliquée à des exemples de circuits combinatoires, et il est dit qu'un circuit synchrone est un circuit auquel on ajoute une entrée correspondant à l'horloge. Nous n'avons pas trouvé d'article montrant comment utiliser cette approche pour des circuits synchrones. Dans cette approche, une machine représente un composant électronique et chaque port (entrée/sortie) est modélisé par une variable locale à la machine. La relation entre les entrées et les sorties est modélisée par une relation mathématique. Cette relation se retrouve dans l'invariant de la machine. La propagation des signaux dans le composant est modélisée par un appel d'opération. Pour chaque sortie une opération de lecture est définie et pour chaque entrée une opération d'écriture. Pour définir l'évolution du système, on utilise une "macro" `Compute_`, définie dans la clause DEFINITIONS. Cette macro est utilisée dans l'invariant et dans les opérations d'écriture des entrées. Ainsi la sortie est mise à jour à chaque fois que l'on modifie une entrée.

A titre d'exemple nous donnons ci-dessous la modélisation d'un multiplexeur. La macro `Compute_` est définie par une formule logique, reliant les entrées principales `yy` et `zz`, l'entrée permettant le choix `xx` et la sortie `res`. Cette macro est ensuite utilisée dans l'invariant et dans les fonctions d'écriture des entrées `In_1` et `In_2`.

```
MACHINE
  B_Mux_0
DEFINITIONS
  Compute_(xx,yy,zz,res)==bool(((xx=FALSE)=>(res=yy)) & ((xx=TRUE)=>(res=zz)))
VARIABLES
  Select , in_1 , in_2 , out
INVARIANT
  Select:BOOL & in_1:BOOL & in_2:BOOL & out:BOOL & Compute_(Select,in_1,in_2,out)
INITIALISATION
  Select,in_1,in_2,out:(
    Select:BOOL & in_1:BOOL & in_2:BOOL& out:BOOL & Compute_(Select,in_1,in_2,out))
OPERATIONS
  In_1(val) =
    in_1,out :(
      in_1 : BOOL & in_1=val & out: BOOL & Compute_(Select,in_1,in_2,out))
  In_2(val) =
    in_2,out :(
      in_2 : BOOL & in_2=val & out: BOOL & Compute_(Select,in_1,in_2,out))
  Gate (val) =
    Select,out:(
      Select : BOOL & Select=val &out: BOOL & Compute_(Select,in_1,in_2,out))
  val <-- Out =
    val := out
END
```

Les auteurs de [20] ont défini ainsi la bibliothèque VHDL standard `STD_LOGIC_1164` qui contient la définition des bits, des vecteurs de bits et des opérations de base sur ces objets.

Il nous semble que cette approche trouve vite ses limites. Pour illustrer ce propos on peut citer le titre d'une section de [20] : "Modelling of complex circuit : a multiplexor".

Dans notre approche, un multiplexeur est modélisé par une composition conditionnelle (IF) et nous ne le raffinons pas jusqu'au niveau portes logiques. Il nous semble que l'approche présentée dans [20], consistant à traduire les composants de bases de VHDL, tient plus de la synthèse de circuits que de la

modélisation à proprement parler. Il existe déjà des synthétiseurs automatiques qui ne sont certes pas totalement fiables (d'où la nécessité de faire des vérifications d'équivalence entre les modèles avant et après synthèse), mais il ne nous semble irréaliste de vouloir réaliser des synthèses en B pour des composants réalistes comme des contrôleurs de bus, des microprocesseurs ou des SOCs.

Par ailleurs, dans cette approche, les propriétés sur le système doivent s'écrire au niveau portes logiques. Ceci peut-être intéressant pour vérifier la correction d'un montage original de portes logiques, mais on sait qu'il est très difficile d'écrire des propriétés système au niveau de l'implantation. Notre approche permet cela puisque le système est d'abord modélisé à un niveau abstrait puis raffiné vers une implantation qui sera synthétisée automatiquement.

Le langage B événementiel a été utilisé par Abrial dans [4] pour modéliser un arbitre et un contrôleur de feu rouge. L'approche présentée par Abrial est similaire à notre approche : elle consiste à modéliser le comportement du circuit par des événements et à modéliser l'environnement explicitement par un événement. Une variable est ajoutée au modèle pour effectuer un "ping pong" entre l'environnement et le circuit.

Cansell et Méry dans [26] ont modélisé un additionneur et un multiplicateur en utilisant le B événementiel. Dans ces exemples, le circuit est d'abord modélisé par un événement unique qui réalise l'expression arithmétique en un coup. Ensuite le système est raffiné en introduisant de nouveaux événements réalisant des opérations binaires les unes après les autres.

La thèse de Cyril Proch [79], réalisée en parallèle à celle-ci, a également consisté à utiliser la méthode B pour la modélisation de systèmes électroniques, notamment pour la production d'un outil de mesure pour une norme (ETSI TR 101 290) dans le domaine de la télévision numérique terrestre.

2.4 Conclusion

Nous avons exposé les principales méthodes formelles utilisées dans la vérification de circuits électroniques. La vérification d'équivalence de modèles et le model-checking ont l'avantage d'être entièrement automatiques, mais trouvent leurs limites dans la taille du modèle pour la vérification d'équivalence et la taille de l'ensemble des états pour le model-checking, bien que des techniques plus récentes comme le model-checking symbolique ou l'interprétation abstraite des modèles permettent d'améliorer les performances. Par ailleurs ces techniques interviennent à des niveaux d'abstraction des circuits en général relativement bas (niveau RTL). La preuve de théorème a l'inconvénient de ne pas être entièrement automatique, mais a l'avantage de ne pas être limité ni par la taille des modèles ni par le niveau d'abstraction du modèle. L'approche proposée dans ce document utilise la preuve de théorème. A des niveaux d'abstraction de hauts niveaux (spécification) la preuve est souvent automatique à 100% et pour les niveaux d'abstraction plus bas (implantation) ne descend généralement pas en dessous des 50% d'automatisation. Même si en théorie la preuve de théorème ne peut pas être automatique à 100% dans le cas général, pour des applications données, on peut envisager de développer des tactiques de preuves dédiées permettant d'obtenir de bonnes performances.

Chapitre 3

La méthode B événementielle

L'objectif de ce chapitre est de présenter la méthode B événementielle. Nous présentons la forme générale que prend un système B ainsi que la notion de substitution. Nous définissons également la notion de raffinement d'un modèle B. Nous donnons également un aperçu du langage BHDL, niveau d'implantation RTL pour le langage B. Nous illustrons les notions sur l'exemple d'un compteur.

A l'origine le langage B [1], inventé par J.R. Abrial et inspiré du modèle logiciel, permet de définir des modèles appelés *machines*. Une machine possède son propre état (un ensemble de variables) et un ensemble d'opérations (équivalent des fonctions d'un langage de programmation). Une opération étant définie par une substitution généralisée, ainsi que par ses paramètres d'entrée et de résultat. La sémantique d'une machine étant que ces opérations sont appelées par un opérateur extérieur. Pour faire un parallèle avec le monde du logiciel, une machine B peut être considérée comme une bibliothèque de fonctions, avec certaines restrictions permettant d'assurer la correction du modèle. A l'intérieur d'une machine on définit en particulier des invariants, qui sont des propriétés qui restent vérifiées quel que soit l'ordre dans lequel l'opérateur extérieur utilise les opérations de la machine.

Une machine B peut être raffinée vers une autre machine. Un raffinement de machine peut consister à raffiner les opérations de la machine (raffinement des substitutions généralisées définissant les opérations), à ajouter des nouvelles variables au modèle (ce qui s'appelle superposition) ou encore à cacher des variables. Dans le cas où des variables sont cachées , il faut fournir un invariant appelé *invariant de collage* qui permet de faire le lien entre l'ancien état (les variables qui ont été enlevées) et le nouvel état. Intuitivement, cet invariant de collage spécifie de quelle manière on retrouve les propriétés sur les anciennes variables à partir des variables actuelles. Cet invariant de collage doit être prouvé comme tout autre invariant.

La méthode B est la méthode qui définit formellement la relation de raffinement entre ces machines et définit les obligations de preuve permettant de garantir la correction des machines et de leurs raffinements. Des outils comme AtelierB [28], B-Toolkit [15] ou B4free [29] et Click'n'Prove [24], permettent de générer automatiquement les obligations de preuve et proposent des outils permettant d'effectuer les preuves.

Le B événementiel a été proposé en 1996 [9, 5] et a été utilisé pour la modélisation d'un système électronique dans [4]. Par rapport à une machine B, un système B événementiel ne peut utiliser qu'un nombre restreint de substitutions généralisées, mais c'est essentiellement l'interprétation du modèle qui change. Le modèle possède toujours un état défini par un ensemble de variables et un invariant sur cet état. Par contre les opérations sont remplacées par des événements. Un événement est défini par une garde et une substitution généralisée, et un événement ne comporte pas de paramètre d'entrée ou de résultat. L'interprétation du modèle n'est plus qu'un opérateur extérieur appelle des opérations mais que le modèle *évolue de lui-même*. La garde de l'événement est la condition sous laquelle l'événement *peut* se produire. Lorsque dans un état donné du système, les gardes de plusieurs événement sont vraies, *un seul* événement se produit et le choix de quel événement se produit est non déterministe. Cet événement fait évoluer le système selon la substitution, et de nouveau les événements dont les gardes sont vraies dans ce nouvel état peuvent se produire.

Pour comprendre la différence entre une machine B traditionnelle et un système événementiel, il faut considérer qu'une machine B traditionnelle est une collection d'opérations mises à disposition d'un

opérateur extérieur, et on ignore quelle séquence d'opérations sera appelée par l'opérateur et donc de quelle manière l'état de la machine va évoluer au cours du temps. Un système événementiel décrit l'évolution elle-même du modèle, on sait de quelle manière le système va évoluer, ou plus exactement de quelle manière le système *peut* évoluer (à cause du non déterminisme).

Dans cette perspective, un événement est l'observation du changement de l'état du système. Plus il y a d'événements dans le modèle du système, plus on observe de modifications du système. Le raffinement d'un système d'événements peut introduire de nouveaux événements et de nouvelles variables d'état. Le raffinement consiste alors à observer plus finement la façon dont le système évolue dans le temps et l'ajout de variables signifie qu'on observe plus de détails du système.

Ce changement d'interprétation implique que le système modélisé est un système *fermé*, il ne possède pas d'entrée ou de sortie lui permettant de communiquer avec son environnement. L'environnement doit donc faire partie intégrante du système [4] et être modélisé explicitement (au moins de manière abstraite). Loin d'être un inconvénient, ceci permet non seulement de modéliser le système lui-même mais également l'environnement dans lequel il évolue. Le modèle de cet environnement est une spécification de celui-ci et constitue l'ensemble des hypothèses qui sont faites pour la modélisation du système.

Ce changement d'interprétation conduit également à quelques changements dans la méthode. La notion de raffinement de substitution reste la même mais la notion de raffinement d'un système B événementiel est différente de la notion de raffinement d'une machine B "traditionnelle". D'une part, le B événementiel permet d'introduire de nouveaux événements lors d'un raffinement, c'est-à-dire des événements qui n'existent pas dans l'abstraction. Formellement, un tel événement qui est ajouté doit être un raffinement correct de la substitution *skip*, c'est-à-dire qu'il ne doit pas modifier les variables qui étaient déjà présentes dans les systèmes plus abstraits. Autrement dit, un nouvel événement ne peut modifier que les nouvelles variables introduites au même pas de raffinement. Le raffinement d'un événement existant peut consister à rendre sa garde plus forte ou à raffiner sa substitution. En plus des obligations de preuve liées à la correction de ces raffinements, la méthode B événementielle ajoute deux types d'obligation de preuve pour la correction du raffinement d'un système : d'une part les nouveaux événements ne doivent pas prendre le contrôle du système indéfiniment et d'autre part le système ne doit jamais se bloquer (il y a toujours un événement qui peut se produire).

Il n'existe pas réellement aujourd'hui d'outils permettant de prendre en charge spécifiquement le B événementiel. Il existe des outils permettant de traduire un système événementiel en machine traditionnelle en intégrant les nouvelles obligations de preuve. D'une manière générale, la syntaxe des modèles étant inchangée, on peut utiliser les outils existants pour prendre en charge un système B événementiel.

Dans la suite de ce chapitre nous présentons le langage B événementiel et nous donnons les obligations de preuve qui sont générées automatiquement pour assurer la correction d'un modèle B événementiel. Avant de définir le raffinement d'un système B et ses obligations de preuve associées, nous expliquons comment ce langage peut être utilisé pour la modélisation de circuits électroniques en prenant un exemple très simple de compteur. Nous finissons en continuant l'exemple du compteur jusqu'à obtenir une implantation. Nous montrons ensuite comment nous pouvons continuer à raffiner pour obtenir une nouvelle implantation.

3.1 Système B événementiel

Un système B événementiel est principalement constitué d'un état (un ensemble de variables) et une collection d'événements faisant évoluer cet état. L'état est contraint par un invariant que les événements doivent préserver. La description d'un événement se fait en utilisant une garde et une substitution généralisée. Dans la suite, nous décrivons la structure générale d'une description en B événementiel et les obligations de preuve qui y sont associées.

La figure 3.1 donne une vue d'ensemble de l'organisation d'un modèle B événementiel. La description est organisée en *clauses*. Une première clause permet de fixer le nom du modèle. La clause SEES donnent une liste de machines qui sont "vues". Lorsqu'une machine est vue par un modèle, le modèle peut utiliser les constantes et les assertions qui ont été prouvées dans cette machine.

La clause SETS permet de définir des ensembles particuliers appelés *ensembles abstraits*. Deux types de définitions existent dans cette clause. Soit on donne simplement le nom de l'ensemble, dans ce cas

FIG. 3.1 – Clauses d'un modèle en B événementiel

```
MODEL
  /* Nom du modèle */
SEES
  /* liste de machines qui sont "vues" */
SETS
  /* Définitions d'ensemble abstraits */
CONSTANTS
  /* Liste de nom de constantes */
PROPERTIES
  /* Propriétés des constantes */
VARIABLES
  /* Liste des variables d'état du modèle */
INVARIANT
  /* Invariant du système */
ASSERTIONS
  /* Des assertions à prouver */
INITIALISATION
  /* Initialisation des variables */
EVENTS
  /* Ensemble d'événements */
END
```

l'ensemble est défini et les seules hypothèses que l'on connaisse sur cet ensemble est qu'il n'est pas vide et qu'il est fini (mais son cardinal n'est pas spécifié). Dans cette clause on peut aussi définir des ensembles en extension : on donne le nom de l'ensemble et la liste de ses éléments entre accolades. Dans ce deuxième cas l'ensemble est défini comme contenant uniquement les éléments listés et les éléments sont définis et utilisables comme des constantes qui sont différentes les unes des autres. Deux ensembles abstraits ne peuvent pas avoir d'éléments en commun. Dans tous les cas, les ensembles abstraits sont utilisables comme des constantes. Deux ensembles abstraits sont toujours considérés comme étant de nature différente, par exemple on ne peut pas faire l'union de deux ensembles abstraits.

Si on veut définir des ensembles constants de manière non abstraite, on peut utiliser les clauses CONSTANTS et PROPERTIES. Ces deux clauses vont de paire, la première déclare la liste des constantes et la deuxième les propriétés de ces constantes. Les constantes peuvent être de toute nature (ensembles, éléments d'ensemble, fonctions ...). Il n'est pas nécessaire de spécifier les constantes de façon complète (par exemple on peut définir un entier sans spécifier qu'elle est sa valeur) mais au minimum le type de chaque constante doit être donné. Les prédicats écrits dans la clause PROPERTIES sont pris comme hypothèse dans toutes les preuves.

Les clauses VARIABLES et INVARIANT fonctionnent également ensemble. La clause VARIABLES donne la liste des variables du modèle et la clause INVARIANT donne les propriétés invariantes sur ces variables. La clause INVARIANT doit au moins contenir les types des variables introduites par le modèle. Lorsque le modèle est un raffinement, les anciennes variables qui ne sont pas retirées du modèle doivent figurées dans la clause VARIABLES mais il n'est pas nécessaire de rappeler leurs types dans la clause INVARIANT.

La clause ASSERTIONS permet d'écrire des prédicats dont on souhaite prouver la véracité et que l'on pourra utiliser dans les preuves. Les propriétés peuvent porter aussi bien sur les variables que sur les constantes. On peut aussi utiliser cette clause pour prouver des propriétés qui ne sont pas nécessairement en correspondance avec un modèle, c'est-dire une machine ne contenant qu'une clause ASSERTIONS. Par exemple on peut faire une machine B ne contenant que des définitions de constantes (fonctions, ensembles ...) et une clause ASSERTIONS permettant d'utiliser les outils supportant la méthode B pour pouvoir prouver des propriétés sur les constantes.

La clause INITIALISATION spécifie par une substitution de quelle façon les variables du modèle sont

initialisées et la clause EVENTS contient la liste des événements du modèle.

3.1.1 Expressions

Le langage des expressions du langage B contient les expressions arithmétiques, les expressions booléennes et les expressions basées sur la théorie des ensembles (incluant les relations et les fonctions). Pour donner une idée du langage, on présente dans le tableau ci-dessous quelques notations avec leurs définitions. Pour une liste détaillée le lecteur se référera aux 200 premières pages de l'ouvrage de référence [1]. Dans le tableau qui suit, les notations r et q dénotent des relations binaires entre des ensembles E_1 et E_2 supposés connus. On utilise également un ensemble F qui est un sous-ensemble de E_1 ($F \subseteq E_1$) ainsi qu'un ensemble $G \subseteq E_2$.

Nom	Syntaxe	Définition
Domaine d'une relation	$dom(r)$	$\{x \mid x \in E_1 \wedge \exists y.(y \in E_2 \wedge (x \mapsto y) \in r)\}$
Relation inverse	r^{-1}	$\{y \mapsto x \mid (y \mapsto x) \in E_2 \times E_1 \wedge (x \mapsto y) \in r\}$
Image	$r[F]$	$\{y \mid y \in E_2 \wedge \exists x.(x \in F \wedge (x \mapsto y) \in r)\}$
Anti-restriction sur le domaine	$F \lhd r$	$\{x \mapsto y \mid (x \mapsto y) \in r \wedge x \notin F\}$
Surcharge	$r \lhd\!\!+ q$	$\{x \mapsto y \mid (x \mapsto y) \in E_1 \times E_2 \wedge$ $(((x \mapsto y) \in r \wedge$ $x \notin dom(q)) \vee (x \mapsto y) \in q)\}$
Restriction sur le codomaine	$r \rhd G$	$\{x \mapsto y \mid (x \mapsto y) \in r \wedge y \in G\}$
Fonctions partielles	$E_1 \nrightarrow E_2$	$\{f \mid f \in E_1 \leftrightarrow E_2 \wedge \forall(x,y,z).(x \mapsto y \in f$ $\wedge\, x \mapsto z \in f \Rightarrow y = z)\}$
Fonctions totales	$E_1 \rightarrow E_2$	$\{f \mid f \in E_1 \nrightarrow E_2 \wedge dom(f) = E_1\}$
Fonctions injectives partielles	$E_1 \rightarrowtail E_2$	$\{f \mid f \in E_1 \nrightarrow E_2 \wedge f^{-1} \in E_2 \nrightarrow E_1\}$
Fonctions injectives totales	$E_1 \rightarrowtail E_2$	$(E_1 \rightarrowtail E_2) \cap (E_1 \rightarrow E_2)$
Séquences finies sur E_1	$seq(E_1)$	$\bigcup n \cdot (n \in \mathbb{N} \mid (1..n) \rightarrow E_1)$
λ-expression	$\lambda k \cdot (k \in E \mid Exp(k))$	$\{x \mapsto y \mid x \in E \wedge y = Exp(x)\}$

3.1.2 Événements et substitutions

Le B événementiel se restreint à 3 formes d'événements. Ces formes séparent la garde (condition sous laquelle l'événement peut se déclencher) de la substitution (la modification appliquée lorsque l'événement se déclenche).

La garde est un prédicat qui sert de condition décrivant dans quelles situations l'événement a la possibilité de se déclencher. Si l'état du système est tel que plusieurs événements peuvent se déclencher, un seul événement se déclenche effectivement et modifie l'état du système. Le choix de l'événement qui se déclenche parmi tous les événements qui ont une garde s'évaluant positivement est non déterministe.

La substitution d'un événement exprime la façon dont le déclenchement de l'événement fait évoluer le système. Une substitution peut prendre trois formes.

1. La forme normale $x : Q(x_0, x)$ exprime que la variable x devient telle que le prédicat $Q(x_0, x)$ est vrai, où x_0 est la valeur *avant* de la variable x et x la nouvelle valeur (la valeur *après*). Par exemple, la substitution $x : (x = x_0 + 1)$ exprime que la variable x est incrémentée de 1.
2. La forme ensembliste $x :\in E(x)$, où $E(x)$ est un ensemble, exprime que la variable x est modifiée de façon à appartenir à l'ensemble $E(x)$. Par exemple $x :\in \mathbb{N}$ exprime que x devient un entier naturel. La forme normale correspondant à cette substitution ensembliste est $x : (x \in E(x_0))$
3. La forme simple $x := Exp$, où Exp est une expression signifie que la nouvelle valeur de x est Exp. Par exemple, la substitution $x := x+1$ incrémente la valeur de x. La forme normale correspondante est $x : (x = [x := x_0]Exp)$ où $[x := x_0]Exp$ est l'expression Exp dans laquelle toute les occurrences libres de x sont remplacées par x_0.

L'évolution du système par une substitution peut également s'exprimer par un prédicat dit *avant-après* que nous noterons BA (pour *before-after*). Ce prédicat relie la valeur des variables avant l'application d'une substitution aux valeurs après cette application. Pour des variables x, la valeur *avant* est notée x

et la valeur *après* est notée x'. Pour une substitution dont la forme normale est $x : Q(x_0, x)$, le prédicat *avant-après* est $BA(x, x')$ défini par $BA(x, x') \Leftrightarrow Q(x, x')$. Par exemple, le prédicat *avant-après* de la substitution $x := x + 1$ est $x' = x + 1$.

On présente ci-dessous les trois formes d'événements utilisées en B événementiel. Les notations $G(...)$ sont des prédicats dont les variables mis dans la parenthèse sont des variables libres. Les substitutions sont notées sous la forme normale, les variables entre parenthèses sont les variables qui sont utilisées par la substitution. Les variables x sont les variables d'état du système et *Event* est le nom donné à l'événement. Dans la deuxième forme (WHEN) le prédicat $G(x)$ correspond exactement à la garde de l'événement. Le prédicat et la substitution n'utilisent que des variables d'état du système. La troisième forme (ANY) est la plus générale. Elle permet d'introduire des variables locales à l'événement. Dans ce cas la garde n'est plus exactement G mais $\exists(l) \cdot G(x, l)$ c'est-à-dire que l'événement ne peut se déclencher que lorsqu'il est possible d'instancier les variables locales l de telle façon que le prédicat puisse s'évaluer positivement. La première forme est la plus simple, elle correspond à un événement qui n'a pas de garde (la garde est toujours vraie) et qui peut donc se produire à tout moment. On notera que la garde d'un événement peut éventuellement être toujours fausse, dans ce cas l'événement ne se produira jamais, c'est un moyen utilisé pour "supprimer" des événements d'un modèle. On définit également les prédicats *avant-après* d'un événement. Ce prédicat exprime la relation qui existe entre les valeurs des variables avant l'application de l'événement et les valeurs après. Pour l'initialisation on a un prédicat *après* qui donne les valeurs des variables suite à l'initialisation, on note se prédicat $Init(x)$

Initialisation	Prédicat *après*
INITIALISATION $x : Init(x)$	$Init(x)$

Événement	Prédicat *avant-après* : $BA(x, x')$
BEGIN $Q(x_0, x)$ END	$Q(x, x')$
WHEN $G(x)$ THEN $x : Q(x_0, x)$ END	$G(x) \wedge Q(x, x')$
ANY l WHERE $G(x, l)$ THEN $x : Q(x_0, x, l)$ END	$\exists l \cdot (G(x, l) \wedge Q(x, x', l))$

3.1.3 Obligations de preuve

On sépare les obligations de preuve en deux parties. D'une part les obligations de preuve permettant d'assurer la correction du modèle en lui-même. Il s'agit de vérifier que les invariants qui sont spécifiés sont bien des invariants du système. D'autre part il y a les obligations de preuve permettant d'assurer (le cas échéant) que le système est un raffinement correct du système abstrait spécifié dans la clause REFINES. Il peut y avoir d'autres obligations de preuve provenant de la clause ASSERTIONS, dans ce cas les obligations de preuve consistent à prouver les assertions avec comme hypothèses les propriétés de la clause PROPERTIES et les invariants du modèle.

Preuve des assertions

Une assertion est un théorème sur les variables et les constantes du système qui doit être prouvé sous l'hypothèse de l'invariant et des propriétés des constantes et des ensembles abstraits (clause PROPERTIES). En notant $I(x)$ l'invariant et $P(s, c)$ le contenu de la clause PROPERTIES pour chaque assertion $A(x, s, c)$ on doit prouver la propriété ci-dessous. Les variables du modèle sont x, les ensembles abstraits sont s et les constantes sont c.

$$P(s, c) \wedge I(x) \Rightarrow A(x, s, c)$$

Faisabilité d'une substitution d'un événement

Toute substitution d'un événement doit toujours être faisable, c'est-à-dire qu'à partir du moment où l'événement peut se déclencher (sa garde est vraie) alors la substitution doit donner un résultat. Plus concrètement, cela signifie que pour une substitution sous sa forme normale $x : Q(x_0, x)$ il doit exister une valeur pour x qui satisfasse $Q(x_0, x)$ pour tout état x_0 ayant été atteint par le système. La preuve de l'existence de x se fait sous l'hypothèse des invariants et des propriétés (clause PROPERTIES) sur les constantes et les ensembles abstraits.

En ce qui concerne l'initialisation, la faisabilité signifie que l'initialisation doit être capable de donner une valeur à chaque variable.

On résume les obligations de preuve liées à la faisabilité dans le tableau ci-dessous :

Événement	Faisabilité
INITIALISATION $x : Init(x)$	$P(s,c) \Rightarrow \exists x \cdot Init(x)$
BEGIN $Q(x_0, x)$ END	$P(s,c) \wedge I(x) \Rightarrow \exists x' \cdot Q(x, x')$
WHEN $G(x)$ THEN $x : Q(x_0, x)$ END	$P(s,c) \wedge I(x) \wedge G(x) \Rightarrow \exists x' \cdot Q(x, x')$
ANY l WHERE $G(x,l)$ THEN $x : Q(x_0, x, l)$ END	$P(s,c) \wedge I(x) \wedge G(x,l) \Rightarrow \exists x' \cdot Q(x, x', l))$

Correction de l'invariant

L'invariant du système ($I(x)$), prédicat sur l'état (x) du système, doit être satisfait quelle que soit l'évolution du système. L'invariant contient entre autre le type des variables. L'évolution du système étant modélisé par les événements, il faut s'assurer que tous les événements préservent l'invariant. Il faut aussi s'assurer que l'initialisation établit l'invariant. Pour s'assurer cela, les deux obligations de preuve suivantes sont générées automatiquement par les outils supportant la méthode B.

- La substitution de la clause INITIALISATION établit l'invariant. Si $x : Init(x)$ est la substitution de l'initialisation, la propriété suivante doit être prouvée :

$$Init(x) \Rightarrow I(x)$$

- Chaque événement (observable) doit préserver l'invariant. Le fait que l'événement soit observable, c'est-à-dire que sa garde est vraie, est contenu dans le prédicat *avant-après* qui contient la garde de l'événement. Donc si la garde d'un événement est fausse, la propriété de préservation de l'invariant ci-dessous est nécessairement vraie.

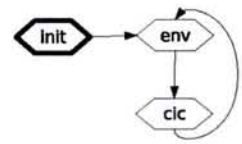
$$P(s,c) \wedge I(x) \wedge BA(x, x') \Rightarrow I(x')$$

3.1.4 Exemple de modèle B événementiel

A titre d'illustration, nous prenons dans ce chapitre un exemple de développement d'un circuit électronique en B. Pour cela, nous utilisons un exemple trivial qui n'aurait pas besoin normalement d'être autant détaillé. L'exemple est celui d'un compteur, qui, outre le signal d'horloge et le signal de réinitialisation asynchrone, possède une entrée pour remettre le compteur à zéro et une sortie qui est une alarme, activée lorsque le compteur atteint sa position maximale. Notons d'abord que les deux signaux "réinitialisation asynchrone" et "horloge" ne sont pas modélisés explicitement dans le modèle B. Le signal de réinitialisation asynchrone est activé une seule fois lors de la mise sous tension du système ; lorsqu'il est activé, les registres prennent leurs valeurs par défaut, nous modélisons son effet par la clause INITIALISATION du modèle B. L'horloge n'est pas modélisée explicitement non plus. Seul le comportement cyclique du système est modélisé. Dans cette spécification, ceci est représenté par deux événements dont les occurrences sont cycliques : on commence d'abord par l'événement *env* qui modélise l'environnement, puis l'événement *circ* qui modélise le circuit, ensuite l'événement *env* se déclenche à nouveau et

ainsi de suite. Ceci est représenté sur la figure 3.2. Chaque événement est représenté par un hexagone. Les transitions entre les événements montrent de quelle façon les événements peuvent se succéder. On se reportera à l'annexe G pour une description de ce type de graphe. Dans la suite nous donnerons toujours sur une figure le graphe de l'enchaînement des événements avant de présenter les événements eux mêmes, ceci permettant d'avoir une meilleure compréhension de l'évolution du système avant de lire les événements.

FIG. 3.2 – Graphe d'événements de la spécification d'un compteur



Dans notre modèle, nous utilisons un jeton pour ordonnancer les événements. Nous définissons l'ensemble abstrait $JET0$ qui est l'ensemble des jetons (en l'occurrence il n'y en a qu'un seul : la constante $jeton$) et la constante MAX qui est la valeur maximale du compteur. On remarquera que la valeur de MAX, la valeur maximum du compteur, n'a pas été fixée. Les preuves qui seront faites seront donc valables quel que soit la valeur de MAX.

SETS
$JET0 = \{jeton\}$
CONSTANTS
MAX,
PROPERTIES
$MAX \in \mathbb{N} \wedge$
$MAX > 0$

Le modèle contient trois variables (rst, alm et $compt$) correspondant à l'état du circuit plus 2 autres ($PENV$ et $PCIRC$) permettant l'ordonnancement entre les événements. La variable rst correspond à l'entrée du circuit, lorsque rst vaut $TRUE$ alors le compteur est remis à zéro. La variable alm est la sortie du circuit qui vaut $TRUE$ lorsque le compteur a atteint sa valeur maximale. La variable $compt$ correspond à l'état interne du compteur, elle mémorise la valeur du compteur. Les variables $PENV$ et $PCIRC$ sont des ensembles qui sont soit vides, soit égaux au singleton $\{jeton\}$. Autrement dit, elles contiennent ou non le jeton et servent à ordonnancer les deux événements.

VARIABLES
/* variables du circuit */
rst, /* entrée : reset synchrone */
alm, /* sortie : alarme */
$compt$, /* la variable compteur */

/* variables d'ordonnancement */
$PENV$,
$PCIRC$
INVARIANT
/* Typage des variables */
$rst \in BOOL \wedge$
$alm \in BOOL \wedge$
$compt \in NATURAL \wedge$

/* Typage des variables d'ordonnancement */
$PENV \subseteq JET0 \wedge$
$PCIRC \subseteq JET0 \wedge$

/* Quelques propriétés */
/* 1 */ $(PENV = \varnothing \vee PCIRC = \varnothing) \wedge$
/* 2 */ $(PENV = \{jeton\} \vee PCIRC = \{jeton\}) \wedge$
/* 3 */ $compt \in 0..MAX$
ASSERTIONS
$PENV \cup PCIRC = \{jeton\} \wedge$
$PENV \cap PCIRC = \varnothing$

Les deux assertions se prouvent en utilisant les invariants notés 1 et 2.

L'initialisation est appliquée au démarrage du système, le signal de réinitialisation asynchrone est activé. Elle doit contenir l'initialisation de l'entrée rst et de la sortie alm car toute variable doit être initialisée en B. Comme l'initialisation ne doit pas fixer de valeur pour ces variables, elles sont initialisées de manière non déterministe. La variable $compt$ est initialisée à 0 pour dire que le compteur démarre à zéro. Pour les variables d'ordonnancement, c'est $PENV$ qui contient le jeton à l'initialisation, ce qui signifie que c'est l'événement correspondant à l'environnement qui débutera juste après l'initialisation. On donne sur la figure 3.2 le graphe d'événements du modèle. Les événements sont représentés par des hexagones, les flèches représentent l'ordre dans lequel les événements peuvent se déclencher.

INITIALISATION
$rst :\in BOOL \|$
$alm :\in BOOL \|$
$compt := 0 \|$
$PENV := \{jeton\} \|$
$PCIRC := \varnothing$

L'événement env modélise l'environnement du circuit. Il est chargé de fournir les entrées pour le circuit. Dans notre exemple, l'événement consiste simplement à écrire la variable d'entrée rst. La modification de cette variable est non déterministe et peut à tout moment prendre n'importe quelle valeur booléenne, ceci signifie qu'il n'y a pas de contrainte sur cette entrée. Dans des cas plus complexes, on pourrait imaginer des contraintes sur les entrées en fonction des sorties du circuit par exemple.

env =
WHEN
$PENV = \{jeton\}$
THEN
$rst :\in BOOL \|$
$PENV := \varnothing \|$
$PCIRC := \{jeton\}$
END

Dans cette spécification, le circuit lui-même est spécifié par une formule logique mise dans la garde de l'événement. Si l'entrée rst est activée le compteur doit être remis à zéro. Sinon, si le compteur a atteint sa valeur maximale il y reste, sinon il est incrémenté. La sortie alm est déclenchée si le compteur a atteint sa valeur maximale. On remarquera qu'il y a un décalage entre la mise à jour du compteur et l'alarme (on utilise l'ancienne valeur du compteur, et non pas c, dans l'expression $bool(compt = MAX)$). Ainsi lorsque le compteur atteint sa valeur maximale, l'alarme n'est déclenchée qu'au cycle suivant.

circ =
ANY c WHERE
$PCIRC = \{jeton\} \wedge$
$c \in \mathbb{N} \wedge$
$(rst = TRUE \Rightarrow c = 0) \wedge$
$(rst = FALSE \Rightarrow c = min(MAX, compt + 1)$
THEN
$compt := c \|$
$alm := bool(compt = MAX) \|$
$PCIRC := \varnothing \|$
$PENV := \{jeton\}$
END

On peut calculer le prédicat *avant-après* de cet événement, on le note BAA :

$BAA(rst, alm, comp, PENV, PCIRC, rst', alm', comp', PENV', PCIRC')$
$\Leftrightarrow \exists c \cdot ($
$PCIRC = \{jeton\} \wedge$
$c \in \mathbb{N} \wedge$
$(rst = TRUE \Rightarrow c = 0) \wedge$
$(rst = FALSE \Rightarrow c = min(MAX, compt + 1)$
$\wedge$
$compt' = c \wedge$
$alm' = bool(compt = MAX) \wedge$
$PCIRC' = \varnothing \wedge$
$PENV' = \{jeton\}$
$)$

3.1.5 Exemple d'obligation de preuve

A titre d'illustration, nous montrons l'obligation de preuve générée automatiquement par les outils pour assurer la préservation de l'invariant noté 3. Nous devons prouver la propriété ci-dessous, où I est l'ensemble des invariants et P la clause PROPERTIES.

$$P \wedge I \wedge BAA \begin{pmatrix} rst, alm, comp, PENV, PCIRC, \\ rst', alm', comp', PENV', PCIRC' \end{pmatrix} \Rightarrow compt' \in 0..MAX$$

Nous avons en partie gauche de l'implication la formule BAA qui est une quantification existentielle sur une variable c. On peut donc exhiber une instance, notée c_1, de cette variable. Ce qui nous amène à l'obligation de preuve suivante.

$$P \wedge I \wedge \boxed{\begin{array}{l} PCIRC = \{jeton\} \wedge \\ c_1 \in \mathbb{N} \wedge \\ (rst = TRUE \Rightarrow c_1 = 0) \wedge \\ (rst = FALSE \Rightarrow c_1 = min(MAX, compt + 1) \\ \wedge \\ compt' = c_1 \wedge \\ alm' = bool(compt = MAX) \wedge \\ PCIRC' = \varnothing \wedge \\ PENV' = \{jeton\} \end{array}} \Rightarrow compt' \in 0..MAX$$

Comme nous avons $compt' = c_1$, l'obligation de preuve revient à montrer que $c_1 \in 0..MAX$. À partir de là, on peut faire un raisonnement par cas sur la variables rst. Si elle vaut $TRUE$, $c_1 = 0$ donc la propriété est vérifiée. Si rst vaut $FALSE$, on a dans P la propriété $MAX \in \mathbb{N} \wedge MAX > 0$ et dans I la propriété $compt \in 0..MAX$ (d'où on déduit $compt + 1 \in 1..MAX + 1$), on peut donc déduire que $min(MAX, compt + 1) \in 0..MAX$.

3.2 Raffinement d'un modèle B événementiel

La méthode permet de développer des systèmes de façon incrémentale en utilisant le raffinement. Le raffinement permet de commencer le développement par un modèle abstrait très simple du système. Ensuite, des détails sont ajoutés au fur et à mesure par raffinements successifs. Les modèles les plus abstraits constituent la spécification du système. Le long de la chaîne de raffinement, le système est modélisé de façon de plus en plus précise jusqu'à obtenir un niveau suffisamment concret correspondant à une implantation.

La figure 3.3 donne une vue d'ensemble d'un système raffinant un autre en B événementiel. La structure est à peu près la même que pour un système abstrait avec deux clauses supplémentaires. Une clause REFINES permet de spécifier quel modèle abstrait est raffiné. La clause VARIANT est utilisée lors d'un raffinement, lorsque le modèle ajoute de nouveaux événements à la description du système. Dans ce cas une expression arithmétique que chaque nouvel événement fait décroître doit être donnée dans cette clause.

Le raffinement d'un système consiste d'une part à raffiner l'état du système et d'autre part à raffiner l'ensemble des événements. Le raffinement de l'état se fait en ajoutant de nouvelles variables au modèle ou en enlevant d'autres. Un invariant appelé *invariant de collage* exprime la relation entre l'état du système abstrait (que nous notons v dans la suite) et l'état du système raffiné (noté w dans la suite) ; nous notons par $J(x, y)$ l'invariant de collage. Le raffinement de l'ensemble des événements permet de raffiner des événements existants ou à en ajouter d'autres. Raffiner un événement existant consiste à rendre sa garde plus forte (l'événement raffiné peut se produire moins souvent que l'événement abstrait) ou à raffiner la substitution elle-même. Ajouter un nouvel événement peut se faire à condition que ce nouvel événement soit formellement un raffinement de l'événement qui n'a pas de garde et qui a *skip* comme substitution. Autrement dit, la garde d'un nouvel événement peut-être celle qu'on veut mais la substitution ne peut modifier que les variables nouvellement introduites au même pas de raffinement (les anciennes variables peuvent être utilisées dans les expressions mais pas modifiées).

- L'initialisation peut être raffinée, notamment en enlevant les initialisations des variables retirées du modèle, par l'ajout de l'initialisation des nouvelles variables, ou en raffinant certaines des substitutions de l'initialisation. En supposant que l'état du système abstrait est x et son initialisation $x : init(x)$ alors que le raffinement a y comme état avec une initialisation $y : INIT(y)$ et $J(x, y)$ comme invariant de collage, on doit prouver la propriété ci-dessous qui indique que la nouvelle initialisation est cohérente avec l'ancienne.

$$INIT(y) \Rightarrow \exists x \cdot (init(x) \ \wedge \ J(x, y))$$

FIG. 3.3 – Clauses d'un raffinement en B événementiel

```
MODEL
  /* Nom du modèle */
REFINES
  /* Nom du modèle abstrait */
SEES
  /* liste de machines qui sont "vues" */
SETS
  /* Définitions d'ensemble abstraits */
CONSTANTS
  /* Liste de nom de constantes */
PROPERTIES
  /* Propriétés des constantes */
VARIABLES
  /* Liste des variables d'état du modèle */
INVARIANT
  /* Invariant du système */
VARIANT
  /* Variant du système */
ASSERTIONS
  /* Des assertions à prouver */
INITIALISATION
  /* Initialisation des variables */
EVENTS
  /* Ensemble d'événements */
END
```

- Un événement existant peut être raffiné en rendant sa garde plus forte. L'événement raffiné peut également modifier les nouvelles variables. La modification des anciennes variables ne peut se faire que dans les limites des modifications qui étaient déjà possibles dans l'événement abstrait (la nouvelle substitution doit être un raffinement de l'ancienne).

  Supposons que nous avons deux événements dont l'un raffine l'autre. L'événement abstrait a x pour état, $I(x)$ pour invariant et a $BAA(x, x')$ pour prédicat *avant-après*. L'événement raffiné a y comme état, $J(x, y)$ comme invariant de collage et a $BAC(y, y')$ pour prédicat *avant-après*. La propriété ci-dessous doit être prouvée.

$$I(x) \wedge J(x, y) \wedge BAC(y, y') \Rightarrow \exists x' \cdot (BAA(x, x') \wedge J(x', y'))$$

- De nouveaux événements peuvent être ajoutés mais ils doivent être des raffinements corrects de la substitution *skip* (la substitution qui ne modifie pas l'état).

$$I(x) \wedge J(x, y) \wedge BAC(y, y') \Rightarrow J(x, y')$$

En plus des obligations liées aux raffinements des événements eux-mêmes, un raffinement de système B événementiel doit respecter deux contraintes supplémentaires.

D'une part, les nouveaux événements ne doivent pas avoir la possibilité de prendre le contrôle du système indéfiniment, ils doivent toujours rendre le contrôle aux anciens événements au bout d'un temps fini. Cette contrainte vient de l'interprétation qui est faite de l'introduction des nouveaux événements : ils introduisent une vue plus précise du système, ce sont des événements qui se produisent *entre* les événements plus abstraits. Le comportement plus global, exprimé par les événements existants, doit être respecté et ne pas être empêché par les nouveaux événements. Ceci peut se rapprocher de la notion de "stuttering équivalence" [21] : si on choisit de ne pas observer les nouveaux événements, alors

l'enchaînement des événements de l'abstraction et du raffinement sont les mêmes (ou plus exactement *peuvent* être les mêmes, compte tenu du non déterminisme qui existe en B). Les outils existants ne proposent pas de moyen très pratique de prouver cela. Actuellement, il faut spécifier un *variant* : il s'agit d'une expression arithmétique dont il faut prouver qu'elle décroît à chaque occurrence d'un des nouveaux événements. Ce variant étant de type entier positif, on sait donc que la décroissance de ce variant ne peut pas être infinie. Cette méthode est formellement suffisante mais il peut être parfois difficile de trouver une expression correcte de variant, c'est pourquoi, pour ne pas compliquer les choses, il vaut mieux limiter au maximum le nombre de nouveaux événements qui sont introduits à chaque pas de raffinement (et faire plusieurs pas de raffinement successifs pour introduire plusieurs événements si nécessaire). Les obligations de preuve peuvent s'exprimer de la façon suivante. Le variant est noté $V(y)$ où y est l'état du système raffiné.

$$I(x) \wedge J(x,y) \Rightarrow V(y) \in \mathbb{N}$$

$$I(x) \wedge J(x,y) \wedge BAC(y,y') \Rightarrow V(y') < V(y)$$

D'autre part, un raffinement ne doit pas introduire de blocage dans le système : à chaque fois qu'un événement pouvait se produire dans le système abstrait, un événement doit pouvoir se produire dans le système concret (le même événement raffiné ou un autre). Cette nécessité provient du fait qu'on souhaite que l'abstraction corresponde bien à une spécification du système, y compris au niveau des blocages : si aucun blocage n'est possible dans l'abstraction (la spécification), aucun blocage ne doit être possible dans les raffinements (et finalement au niveau implantation). En notant par $\bigvee_C$ la disjonction de toutes les gardes des événements du système raffiné par and $\bigvee_A$ la disjonction de toutes les gardes abstraites, on peut exprimer cette condition par l'obligation de preuve suivante :

$$I(x) \wedge J(x,y) \wedge \bigvee_A \Rightarrow \bigvee_C$$

L'outil proposé actuellement par Clearsy permettant de transformer un système B événementiel en machine pouvant être prise en charge par les outils supportant la méthode B utilise une obligation de preuve légèrement différente. Cet outil considère qu'il ne doit en fait y avoir aucun blocage possible du système (y compris dans l'abstraction). L'obligation de preuve qui est générée est alors que la disjonction de toutes les gardes soit toujours évaluée positivement (on retire $\bigvee_A$ de l'obligation de preuve ci-dessus).

3.2.1 Exemple de Raffinement

Ce raffinement modifie un peu la garde de l'événement "circ" en remplaçant le calcul de c par un calcul un peu plus détaillé. Le reste du système est inchangé, sauf qu'il n'y a pas d'invariant. Dans l'événement abstrait, dans le cas où il n'y avait pas d'initialisation on calculait c comme étant le minimum de MAX et $compt + 1$. Maintenant nous détaillons le calcul en disant que si le maximum est atteint alors la valeur de c est MAX sinon sa valeur est incrémentée. Les deux écritures reviennent au même résultat, c'est ce que nous indique la preuve de correction du raffinement pour cet événement.

circ =
  ANY c WHERE
    $PCIRC = \{jeton\} \wedge$
    $c \in \mathbb{N} \wedge$
    $(rst = TRUE \Rightarrow c = 0) \wedge$
    $(compt = MAX \wedge rst = FALSE \Rightarrow c = MAX) \wedge$
    $(compt \neq MAX \wedge rst = FALSE \Rightarrow c = compt + 1)$
  THEN
    $compt := c \|$
    $alm := bool(compt = MAX) \|$
    $PCIRC := \varnothing \|$
    $PENV := \{jeton\}$
  END

On peut calculer le prédicat *avant-après* de cet événement, on le note BAC :

$$
\begin{array}{l}
BAC(rst, alm, comp, PENV, PCIRC, rst', alm', comp', PENV', PCIRC') \\
\Leftrightarrow \exists c \cdot (\\
\quad PCIRC = \{jeton\} \wedge \\
\quad c \in \mathbb{N} \wedge \\
\quad (rst = TRUE \Rightarrow c = 0) \wedge \\
\quad (compt = MAX \wedge rst = FALSE \Rightarrow c = MAX) \wedge \\
\quad (compt \neq MAX \wedge rst = FALSE \Rightarrow c = compt + 1) \\
\quad \wedge \\
\quad compt' = c \wedge \\
\quad alm' = bool(compt = MAX) \wedge \\
\quad PCIRC' = \varnothing \wedge \\
\quad PENV' = \{jeton\} \\
)
\end{array}
$$

Dans notre cas, la preuve raffinement pour le l'événement "circ" est alors la suivante. Comme les variables du raffinement sont les mêmes que celles de l'abstraction, nous les renommons en ajoutant un indice 1 (par exemple alm est renommé alm_1 dans le raffinement) et nous ajoutons l'invariant de collage $rst_1 = rst \wedge alm_1 = alm \wedge comp_1 = comp \wedge PENV_1 = PENV \wedge PCIRC_1 = PCIRC$.

$$
\begin{array}{l}
I(rst, alm, comp, PENV, PCIRC) \wedge \\
rst_1 = rst \wedge alm_1 = alm \wedge comp_1 = comp \wedge PENV_1 = PENV \wedge PCIRC_1 = PCIRC \wedge \\
BAC(rst_1, alm_1, comp_1, PENV_1, PCIRC_1, rst'_1, alm'_1, comp'_1, PENV'_1, PCIRC'_1) \\
\Rightarrow \\
\exists (rst', alm', comp', PENV', PCIRC') \cdot (\\
\quad BAA(rst, alm, comp, PENV, PCIRC, rst', alm', comp', PENV', PCIRC') \wedge \\
\quad rst'_1 = rst' \wedge alm'_1 = alm' \wedge comp'_1 = comp' \wedge PENV'_1 = PENV' \wedge PCIRC'_1 = PCIRC')
\end{array}
$$

Ce qui se simplifie dans la formule suivante.

$$
\begin{array}{l}
I(rst, alm, comp, PENV, PCIRC) \wedge \\
rst_1 = rst \wedge alm_1 = alm \wedge comp_1 = comp \wedge PENV_1 = PENV \wedge PCIRC_1 = PCIRC \wedge \\
BAC(rst_1, alm_1, comp_1, PENV_1, PCIRC_1, rst'_1, alm'_1, comp'_1, PENV'_1, PCIRC'_1) \\
\Rightarrow \\
BAA(rst, alm, comp, PENV, PCIRC, rst'_1, alm'_1, comp'_1, PENV'_1, PCIRC'_1)
\end{array}
$$

La formule BAC est une quantification existentielle sur une variable c. Comme elle se trouve en hypothèse de l'obligation de preuve (en partie gauche de l'implication) on peut exhiber une instance, disons c_1, de cette variable. La formule BAA est également une quantification existentielle sur une variable c. Il faut trouver la bonne instance pour c de façon à prouver l'obligation de preuve. On prouve que c_1 est une instance satisfaisante. Il nous faut alors prouver la propriété suivante.

$$
\begin{array}{l}
I(rst, alm, comp, PENV, PCIRC) \wedge \\
rst_1 = rst \wedge alm_1 = alm \wedge comp_1 = comp \wedge PENV_1 = PENV \wedge PCIRC_1 = PCIRC \wedge \\
PCIRC_1 = \{jeton\} \wedge \\
c_1 \in \mathbb{N} \wedge \\
(rst_1 = TRUE \Rightarrow c_1 = 0) \wedge \\
(compt_1 = MAX \wedge rst_1 = FALSE \Rightarrow c_1 = MAX) \wedge \\
(compt_1 \neq MAX \wedge rst_1 = FALSE \Rightarrow c_1 = compt_1 + 1) \\
\wedge \\
compt'_1 = c_1 \wedge \\
alm'_1 = bool(compt_1 = MAX) \wedge \\
PCIRC'_1 = \varnothing \wedge \\
PENV'_1 = \{jeton\} \\
\Rightarrow \\
PCIRC = \{jeton\} \wedge \\
c_1 \in \mathbb{N} \wedge \\
(rst = TRUE \Rightarrow c_1 = 0) \wedge \\
(rst = FALSE \Rightarrow c_1 = min(MAX, compt + 1) \\
\wedge \\
compt'_1 = c_1 \wedge \\
alm'_1 = bool(compt = MAX) \wedge \\
PCIRC'_1 = \varnothing \wedge \\
PENV'_1 = \{jeton\}
\end{array}
$$

En faisant des simplifications, la preuve de correction du raffinement revient à montrer la propriété suivante :

$$
\begin{array}{l}
(compt = MAX \wedge rst = FALSE \Rightarrow c_1 = MAX) \wedge \\
(compt \neq MAX \wedge rst = FALSE \Rightarrow c_1 = compt + 1) \\
\Rightarrow \\
(rst = FALSE \Rightarrow c_1 = min(MAX, compt + 1)
\end{array}
$$

3.3 Niveau d'implantation logiciel : B0

Le langage B permet d'écrire des modèles très abstraits (qui constituent la spécification du système) qui sont raffinés pas à pas pour finalement obtenir des modèles qui sont implantables.

Le sous langage B0 du langage B constitue la partie du langage qui est implantable en logiciel. À partir d'un système décrit en B0 (a priori obtenu par raffinement d'un modèle B abstrait), il existe des traducteurs automatiques, développés par la société Clearsy [27] vers des langages comme C ou ADA.

Dans ce langage B0, outre le fait que les substitutions doivent pouvoir être implantables (prises en charge par les traducteurs automatiques), la notion de boucle est introduite. Pour cela, une substitution WHILE est ajoutée au langage. Cette substitution ne peut être utilisée qu'au niveau de l'implantation, elle est interdite dans les niveaux au-dessus. Nous ne présentons pas cette substitution dans ce document car elle est typique d'une implantation par logiciel et nous nous focalisons sur la modélisation des circuits électroniques. Par ailleurs, un modèle B événementiel permet de faire des boucles sans utiliser cette substitution WHILE, il suffit pour cela de faire un événement qui peut se produire plusieurs fois de suite (il boucle sur lui-même). Les conditions de bonne formation d'un modèle B événementiel (en particulier le fait que de nouveaux événements ne peuvent prendre le contrôle indéfiniment) interdit qu'une boucle ne se termine jamais.

3.4 Niveau d'implantation matériel : BHDL

Comme dans le cas des logiciels, un sous langage de B, nommé BHDL, a été défini pour le niveau correspondant au niveau d'implantation de circuits électroniques. La définition de ce langage permet d'appliquer le flot de conception proposé par la méthode B à la conception de circuits, et plus généralement à des systèmes électroniques, pouvant éventuellement être constitués à la fois de logiciels et de circuits.

Une fois ce sous langage BHDL définit, on peut définir un processus de développement utilisant la méthode B pour le développement de circuits. Un modèle de processus de développement est résumé sur la figure 3.4. Généralement, la spécification initiale est fournie en langue naturelle. Puisqu'une telle spécification n'est pas formelle, elle peut être incomplète, inconsistante sur certains points ou le plus souvent ambiguë. Une première étape consiste à développer des modèles B abstraits de cette spécification. Ces modèles introduisent au fur et à mesure, par le bais du raffinement, les détails de la spécification, sans se soucier de la manière dont le système sera implanté.

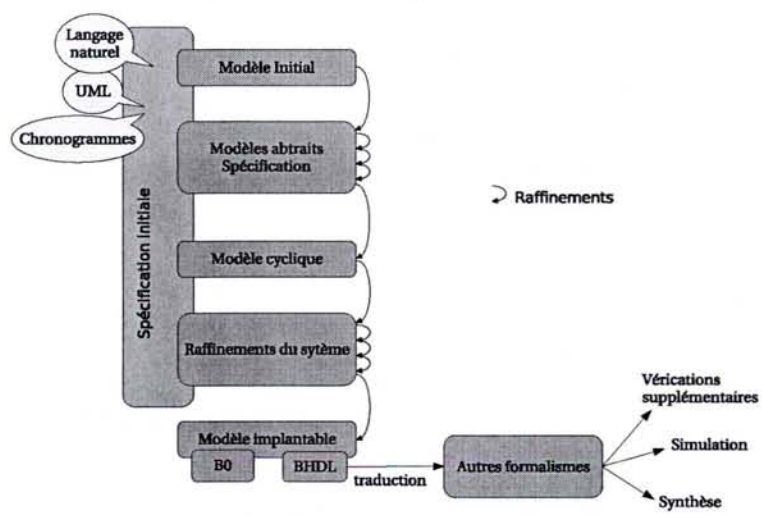
En pratique chaque élément de la spécification doit être introduit au niveau de modélisation le plus abstrait possible permettant d'exprimer les propriétés souhaitées par la spécification. Plus les propriétés sont introduites à un niveau abstrait et plus les preuves de ces propriétés seront simples (voire triviales) à réaliser. Le processus de raffinement assure que ces propriétés sont conservées dans les modèles plus concrets. De plus, les modèles abstraits comprenant moins de détails sont plus simples à comprendre et à appréhender par des interlocuteurs. En particulier les modèles abstraits permettent de prouver les protocoles utilisés sans être submergé par les détails d'une implantation par circuit électronique.

Lorsque l'aspect cyclique du comportement du système est nécessaire pour poursuivre le processus de raffinement (le plus tard possible), il peut être modélisé de façon abstraite en synchronisant les composants du système. Un modèle B est par nature un modèle asynchrone, il faut donc ajouter au modèle une synchronisation pour modéliser l'enchaînement des cycles d'horloge, la concurrence et les communications entre les composants.

Au niveau auquel l'aspect cyclique est modélisé, le système est déjà modélisé de façon relativement concrète. Les raffinements suivants consistent essentiellement au raffinement des données pour obtenir un modèle n'utilisant que des types de données implantables (variables booléennes, vecteurs de bits ...). Pour être implantable, un modèle doit respecter certaines contraintes : n'utiliser que des types de données implantables, l'état de chaque composant doit être modélisé de façon indépendante (chaque composant ne modifier que son propre état), identifier les variables modélisant les entrées et les sorties de chaque composant du système.

Les chapitres suivants de ce document expliquent plus en détail les différentes phases de la méthodologie de développement de circuits électroniques en utilisant la méthode B.

FIG. 3.4 – Processus général de développement d'un système électronique



Une fois un modèle implantable obtenu, le modèle B peut-être écrit en conformité avec le langage BHDL. Si le modèle B du circuit est constitué de plusieurs événements, le modèle BHDL s'obtient en regroupant les événements en un seul, dans lequel les gardes des événements deviennent des conditions de substitutions conditionnelles, et l'ordonnancement séquentiel qui peut exister entre différents événements se raffine simplement en utilisant la substitution séquentielle.

Un modèle BHDL est doté de la sémantique classique d'un modèle B mais nous avons également défini

une sémantique formelle spécifique à la modélisation de circuits, définie dans la suite de ce document. Ce modèle BHDL peut ensuite être traduit vers des langages de description de circuit (KeesDA a développé des traducteurs vers VHDL et SystemC) pour permettre la simulation et la synthèse, ou d'autres formalismes proposant un meilleur support pour certaines formes de vérifications que la méthode B, comme l'utilisation du model-checking pour la vérification de propriétés temporelles au niveau cycle d'horloge.

3.5 Exemple de développement de circuit en B

Nous montrons dans cette section comment nous pouvons développer un circuit électronique en utilisant la méthode B. Pour cela, nous poursuivons notre exemple trivial du compteur. Rappelons qu'outre le signal d'horloge et le signal de réinitialisation asynchrone (qui ne sont tous deux pas modélisés explicitement), il possède une entrée pour remettre le compteur à zéro et une sortie qui est une alarme, activée lorsque le compteur atteint sa position maximale.

3.5.1 Calcul de la valeur de l'alarme

Dans ce raffinement du modèle présenté dans la section 3.2.1, nous faisons sortir le calcul de la valeur de la variable alm de l'événement "circ". Pour cela nous raffinons l'ordonnancement. La variable d'ordonnancement $PCIRC$ est remplacée par les deux variables $PCIRC1$ et $PCIRC2$. Comme la modification de la valeur de l'alarme ne s'effectue plus au même endroit dans le modèle, nous devons retirer la variable alm pour la remplacer par une nouvelle, nommée $alm2$. Nous devons tout de même prouver que la variable alm conserve, quoi qu'il arrive, la valeur telle qu'elle était calculée dans le modèle précédent. On notera également qu'il nous faut prouver que le nouvel ordonnancement doit préserver l'ancien. Ceci est fait grâce aux invariants donnés ci-dessous.

VARIABLES
/* Variables préexistantes */
$rst, compt, PENV,$

/* Nouvelles variables */
$alm2,$
$PCIRC1,$
$PCIRC2$

INVARIANT
/* Typage */
$alm2 \in BOOL \wedge$
$PCIRC1 \subseteq JET0 \wedge$
$PCIRC2 \subseteq JET0 \wedge$

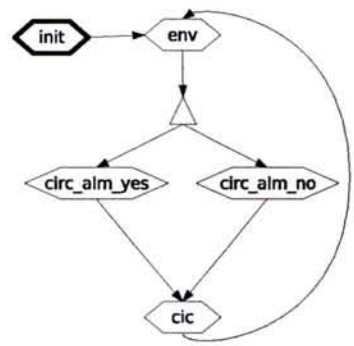
/* Propriétés*/
$PENV \cup PCIRC1 \cup PCIRC2 = \{jeton\} \wedge$
$(PCIRC1 = \varnothing \vee PCIRC2 = \varnothing) \wedge$
$(PCIRC2 = \{jeton\} \Rightarrow alm2 = bool(compt = MAX)) \wedge$

/* Invariant de collage */
$(PENV = \{jeton\} \Rightarrow alm = alm2) \wedge$
$PCIRC1 \cup PCIRC2 = PCIRC$

L'initialisation a peu de changement, seule la variable alm est remplacée par la variable $alm2$. L'environnement est également légèrement modifié pour prendre en compte le raffinement de l'ordonnancement. Nous ajoutons deux nouveaux événements ayant la charge de calculer la valeur de $alm2$. Ces deux événements sont complémentaires, un seul d'entre eux se déclenchera. Le premier, "circ_alm_yes", se déclenche si le compteur a atteint sa valeur maximale et le deuxième, " circ_alm_no", si ce n'est pas le

cas. La figure 3.5 donne le graphe d'événements de ce raffinement. Le triangle représente un choix entre plusieurs transitions.

FIG. 3.5 – Graphe d'événements du premier raffinement



INITIALISATION
$rst :\in BOOL \|$
$alm2 :\in BOOL \|$
$compt := 0 \|$
$PENV := \{jeton\} \|$
$PCIRC1 := \varnothing \|$
$PCIRC2 := \varnothing$

env =
WHEN
$PENV = \{jeton\}$
THEN
$rst :\in BOOL \|$
$PENV := \varnothing \|$
$PCIRC1 := \{jeton\}$
END

circ_alm_yes =
WHEN
$PCIRC1 = \{jeton\} \wedge compt = MAX$
THEN
$alm2 := TRUE \|$
$PCIRC1 := \varnothing \|$
$PCIRC2 := \{jeton\}$
END

circ_alm_no =
WHEN
$PCIRC1 = \{jeton\} \wedge compt \neq MAX$
THEN
$alm2 := FALSE \|$
$PCIRC1 := \varnothing \|$
$PCIRC2 := \{jeton\}$
END

Pour prouver que les deux nouveaux événements "circ_alm_yes" et "circ_alm_no"ne prennent pas le contrôle indéfiniment, on peut donner le variant ci-dessous. En effet ce variant décroît nécessairement puisque qu'il vaut 1 avant l'application d'un des deux événements et vaut 0 après.

VARIANT
$card(PCIRC1)$

Dans l'événement "circ", la variable alm disparaît, mais l'invariant de collage nous assure que la valeur de alm est bien conforme au modèle précédent.

circ =
ANY c WHERE
$PCIRC2 = \{jeton\} \wedge$
$c \in \mathbb{N} \wedge$
$(rst = TRUE \Rightarrow c = 0) \wedge$
$(compt = MAX \wedge rst = FALSE \Rightarrow c = MAX) \wedge$
$(compt \neq MAX \wedge rst = FALSE \Rightarrow c = compt + 1)$
THEN
$compt := c \|$
$PCIRC2 := \varnothing \|$
$PENV := \{jeton\}$
END

3.5.2 Première implantation

Dans ce raffinement nous éclatons l'événement "circ" en trois événements pour réaliser le calcul du compteur. Ce raffinement nous amène à une première implantation, qui après regroupement des événements nous donne le modèle donné sur la figure 3.7 (en prenant 7 comme valeur maximale pour le compteur). Ici encore l'ordonnancement est raffiné. Le graphe d'événement est donné sur la figure 3.6.

VARIABLES
$rst, PENV, alm2, PCIRC1, compt3, PCIRC3, PCIRC4$
INVARIANT
$compt3 \in \mathbb{N} \wedge$
$PCIRC3 \subseteq JET0 \wedge$
$PCIRC4 \subseteq JET0 \wedge$

$(PCIRC3 = \varnothing \vee PCIRC4 = \varnothing) \wedge$

$PCIRC3 \cup PCIRC4 = PCIRC2 \wedge$

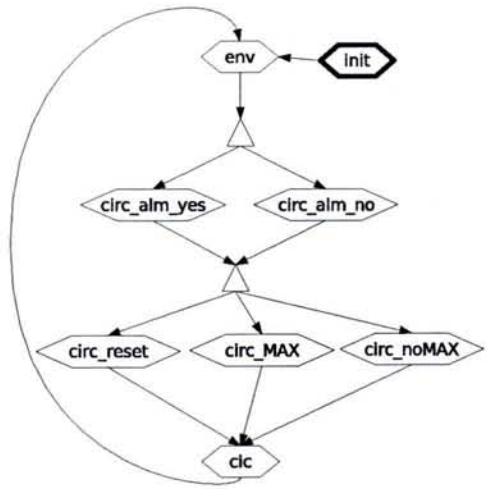
$(PCIRC4 = \{jeton\} \wedge rst = TRUE => compt3 = 0) \wedge$
$(PCIRC4 = \{jeton\} \wedge compt = MAX \wedge rst = FALSE \Rightarrow compt3 = compt) \wedge$
$(PCIRC4 = \{jeton\} \wedge compt \neq MAX \wedge rst = FALSE \Rightarrow compt3 = compt + 1) \wedge$

$(PENV = \{jeton\} \Rightarrow compt3 = compt) \wedge$
$(PCIRC1 = \{jeton\} \Rightarrow compt3 = compt) \wedge$
$(PCIRC3 = \{jeton\} \Rightarrow compt3 = compt)$
VARIANT
$card(PCIRC3)$

INITIALISATION
$rst :\in BOOL \|$
$alm2 :\in BOOL \|$
$compt3 := 0 \|$
$PENV := \{jeton\} \|$
$PCIRC1 := \varnothing \|$
$PCIRC3 := \varnothing \|$
$PCIRC4 := \varnothing$

env =
WHEN
$PENV = \{jeton\}$
THEN
$rst :\in BOOL \|$
$PENV := \varnothing \|$
$PCIRC1 := \{jeton\}$
END

FIG. 3.6 – Graphe d'événements du premier niveau d'implantation



circ_alm_yes =
WHEN
$PCIRC1 = \{jeton\} \wedge compt3 = MAX$
THEN
$alm2 := TRUE \|$
$PCIRC1 := \varnothing \|$
$PCIRC3 := \{jeton\}$
END

circ_alm_no =
WHEN
$PCIRC1 = \{jeton\} \wedge compt3 \neq MAX$
THEN
$alm2 := FALSE \|$
$PCIRC1 := \varnothing \|$
$PCIRC3 := \{jeton\}$
END

Les trois événements "circ_reset", "circ_MAX" et "circ_noMAX" sont ajoutés pour modéliser le calcul du compteur. Chacun de ces événements correspond à une situation différente. Le premier quand le signal rst est activé, le deuxième lorsque le compteur a atteint sa valeur maximale et le troisième lorsque ce n'est pas le cas.

circ_reset =
WHEN
$PCIRC3 = \{jeton\} \wedge$
$rst = TRUE$
THEN
$compt3 := 0 \|$
$PCIRC3 := \varnothing \|$
$PCIRC4 := \{jeton\}$
END

circ_MAX =
WHEN
$PCIRC3 = \{jeton\} \wedge$
$compt3 = MAX \wedge rst = FALSE$
THEN
$PCIRC3 := \varnothing \|$
$PCIRC4 := \{jeton\}$
END

circ_noMAX =
WHEN
$PCIRC3 = \{jeton\} \wedge$
$compt3 \neq MAX \wedge rst = FALSE$
THEN
$compt3 := compt3 + 1 \|$
$PCIRC3 := \varnothing \|$
$PCIRC4 := \{jeton\}$
END

L'événement "circ" est maintenant vide puisque tous les calculs sont effectués par les précédents événements.

```
circ =
  WHEN
    PCIRC4 = {jeton} ∧
  THEN
    PCIRC4 := ∅ ||
    PENV := {jeton}
  END
```

Nous donnons sur la figure 3.7 le modèle BHDL correspondant à ce niveau d'implantation (en prenant 7 comme valeur maximale pour le compteur). Les clauses INPUTS et OUTPUTS sont propres à BHDL et ne sont pas des clauses valides en B. La machine "vue" (clause SEES) $BHDL$ définit le type $UINT3$ comme l'ensemble des nombres représentables sur 3 bits, c'est à dire l'ensemble 0..7.

FIG. 3.7 – Modèle BHDL d'un compteur 3 bits

```
MACHINE
  counter
SEES
  BHDL
INPUTS
  rst
OUTPUTS
  alm2
VARIABLES
  compt3
INVARIANT
  rst ∈ BOOL ∧ alm2 ∈ BOOL ∧ compt3 ∈ UINT3
INITIALISATION
  compt3 := 0 || rst :∈ BOOL || alm2 :∈ BOOL
OPERATIONS
  IF compt3 = MAX THEN
    alm2 := TRUE
  ELSE
    alm2 := FALSE
  END
  ;
  IF rst = true THEN
    compt3 := 0
  ELSE
    IF compt3 ≠ MAX THEN
      compt3 := compt3 + 1
    END
  END
END
```

Pour obtenir ce modèle, nous avons appliqué deux règles de regroupement d'événements. La première règle stipule que lorsque deux événements sont complémentaires, alors on peut les regrouper en utilisant un IF.

```
WHEN P ∧ Q THEN S END
WHEN P ∧ ¬ Q THEN T END
------------------------------
WHEN P THEN
   IF Q THEN S ELSE T END
END
```

Dans notre modèle, on peut d'abord regrouper circ_MAX et circ_noMAX en utilisant la règle avec $Q \equiv compt3 = MAX$. Nous obtenons un événement dont la garde est $PCIRC3 = \{jeton\} \wedge rst = FALSE$. Comme $rst = FALSE \equiv \neg(rst = TRUE)$ on peut encore regrouper ce nouvel événement avec circ_reset pour finalement obtenir l'événement *circ'* ci-dessous. En utilisant la même règle, on peut regrouper circ_alm_yes et circ_alm_no pour obtenir l'événement circ_alm ci-dessous.

circ' =
WHEN
$PCIRC3 = \{jeton\}$
THEN
IF $rst = true$ THEN
$compt3 := 0$
ELSE
IF $compt3 \neq MAX$ THEN
$compt3 := compt3 + 1$
END
END ||
$PCIRC3 := \varnothing$ ||
$PCIRC4 := \{jeton\}$
END

circ_alm =
WHEN
$PCIRC1 = \{jeton\}$
THEN
IF $compt3 = MAX$ THEN
$alm2 := TRUE$
ELSE
$alm2 := FALSE$
END ||
$PCIRC1 := \varnothing$ ||
$PCIRC3 := \{jeton\}$
END

La deuxième règle de regroupement consiste à implanter l'ordonnancement. . Dans la règle ci-dessous, "P et Q ne contiennent que de l'ordonnancement" signifie que les gardes des deux événements ne contiennent que des variables d'ordonnancement : si le modèle est bien formé (c'est-à-dire que les gardes des événements représentent une bonne partition de l'espace des états), la règle de regroupement précédente permet d'obtenir de tels événements ; toutes les variables (hormis les variables d'ordonnancement) qui étaient présentes dans les gardes se retrouvent dans les conditions des IF.

A = WHEN P THEN S END
B = WHEN Q THEN T END
P et Q ne contiennent que de l'ordonnancement
L'ordonnancement est "A puis B"

WHEN P THEN S ; T END

Plus concrètement, sur notre exemple cette règle peut se réécrire sous la forme ci-dessous. Cette règle supprime la variable d'ordonnancement $PCIRC_{i+1}$ du modèle en l'implantant par une composition séquentielle.

A = WHEN $PCIRC_i = \{jeton\}$ THEN $S \| PCIRC_i = \varnothing \| PCIRC_j := \{jeton\}$ END
B = WHEN $PCIRC_j = \{jeton\}$ THEN $T \| PCIRC_j = \varnothing \| PCIRC_k := \{jeton\}$ END
$S' \equiv skip$ si A fait partie de l'environnement, $S' \equiv S$ sinon
$T' \equiv skip$ si B fait partie de l'environnement, $T' \equiv T$ sinon
---
C = WHEN $PCIRC_i = \{jeton\}$ THEN $S'; T' \| PCIRC_i = \varnothing \| PCIRC_k := \{jeton\}$ END

Pour savoir dans quel ordre regrouper les événements on regarde la clause INITIALISATION. Celle-ci positionne la variable $PENV$ à $\{jeton\}$, on commence donc par regrouper l'événement qui a $PENV = \{jeton\}$ comme garde (c'est-à-dire l'événement env) avec son successeur. On trouve son successeur en regardant la substitution de l'événement *env* : elle positionne la variable $PCIRC1$ à $\{jeton\}$, le successeur de env est donc l'événement qui a $PCIRC1 = \{jeton\}$ comme garde ; c'est-à-dire l'événement circ_alm. On notera que comme l'événement env ne fait pas partie du circuit mais de l'environnement, la substitution $rst :\in BOOL$ qu'il contient est remplacée par *skip* (la substitution qui n'a pas d'effet), puisque l'événement env ne doit pas avoir d'effet sur le circuit. On regroupe donc les événements env et circ_alm (suppression de la variable $PCIRC1$) pour obtenir un événement circ_alm'. Ensuite on regroupe circ_alm' avec circ' (suppression de $PCIRC3$) pour obtenir un événement circ". Puis circ" est regroupé avec circ (suppression de $PCIRC4$) pour obtenir un événement circ3. Notons que ce regroupement est spécial puisque le résultat est de la forme circ^3 = WHEN $PENV = \{jeton\}$ THEN $S \| PENV := \varnothing \| PENV :=$

$\{jeton\}$, ce qui n'est pas bien formé puisqu'on ne peut modifier la variable $PENV$ dans deux substitutions parallèles. Ceci indique la fin du regroupement, l'événement bien formé que l'on obtient finalement est BEGIN S END. Le modèle final obtenu dans notre exemple est montré sur la figure 3.7.

3.5.3 Vers une autre implantation

Le modèle précédent est une version implantable utilisant l'addition sur des entiers pour incrémenter le compteur. Dans ce raffinement nous remplaçons le compteur modélisé par un entier par un compteur modélisé par un banc de registres de valeurs booléennes. Ce banc de registres contient un seul jeton (valeur booléenne 'TRUE') qui passe d'un registre à un autre à chaque cycle, modélisant ainsi l'évolution du compteur. La valeur entière du compteur correspond à l'indice du registre contenant le jeton. Ce raffinement supprime donc la variable *compt*3 du modèle et la remplace par la variable *tabcount* qui est un tableau de valeurs booléennes. L'invariant de collage donne la relation entre *tabcount* et *compt*3 ($dom(tabcount \rhd \{TRUE\}) = \{compt3\}$), il assure en même temps qu'une et une seule des cases du tableau *tabcount* contient un jeton à un moment donné (car $dom(tabcount \rhd \{TRUE\})$ est un singleton).

VARIABLES
$rst, PENV, alm2, PCIRC1, PCIRC3, PCIRC4,$
$tabcount$
INVARIANT
$tabcount : 0..MAX \rightarrow BOOL \wedge$
$dom(tabcount \rhd \{TRUE\}) = \{compt3\}$

L'initialisation positionne le jeton dans la première case du tableau *tabcount*, c'est-à-dire qu'on positionne la valeur du tableau à l'indice 0 à la valeur $TRUE$ et toutes les autres à $FALSE$.

INITIALISATION
$rst :\in BOOL \|$
$alm2 :\in BOOL \|$
$PENV := \{jeton\} \|$
$PCIRC1 := \varnothing \|$
$PCIRC3 := \varnothing \|$
$PCIRC4 := \varnothing \|$
$tabcount := \lambda k.(k = 0 | TRUE) \cup \lambda k.(k : 1..MAX | FALSE)$

Les événements modélisant le circuit sont raffinés de façon à prendre en compte le remplacement de la variable *compt*3 par la variable *tabcount*. Les modifications se trouvent dans les gardes des événements "circ_alm_yes", "circ_alm_no" et "circ_MAX", ainsi que dans les substitutions des événements "circ_reset" et "circ_noMAX". On remarquera en particulier l'expression utilisée pour l'affectation de *tabcount* dans l'événement "circ_noMAX" : il s'agit d'un décalage de registre ; l'opération arithmétique a été remplacée par une simple opération de décalage de registre.

circ_alm_yes =
WHEN
$PCIRC1 = \{jeton\} \wedge$
$tabcount(MAX) = TRUE$
THEN
$alm2 := TRUE \|$
$PCIRC1 := \varnothing \|$
$PCIRC3 := \{jeton\}$
END

circ_alm_no =
WHEN
$PCIRC1 = \{jeton\} \wedge$
$tabcount(MAX) = FALSE$
THEN
$alm2 := FALSE \|$
$PCIRC1 := \varnothing \|$
$PCIRC3 := \{jeton\}$
END

circ_reset =
WHEN
$PCIRC3 = \{jeton\} \wedge$
$rst = TRUE$
THEN
$tabcount := \lambda k.(k = 0|TRUE) \cup \lambda k.(k : 1..MAX|FALSE) \|$
$PCIRC3 := \varnothing \|$
$PCIRC4 := \{jeton\}$
END

circ_MAX =
WHEN
$PCIRC3 = \{jeton\} \wedge$
$tabcount(MAX) = TRUE \ \wedge rst = FALSE$
THEN
$PCIRC3 := \varnothing \|$
$PCIRC4 := \{jeton\}$
END

circ_noMAX =
WHEN
$PCIRC3 = \{jeton\} \wedge$
$tabcount(MAX) = FALSE \ \wedge rst = FALSE$
THEN
$\lambda k.(k = 0|FALSE) \cup \lambda k.(k : 1..MAX|tabcount(k-1)) \|$
$PCIRC3 := \varnothing \|$
$PCIRC4 := \{jeton\}$
END

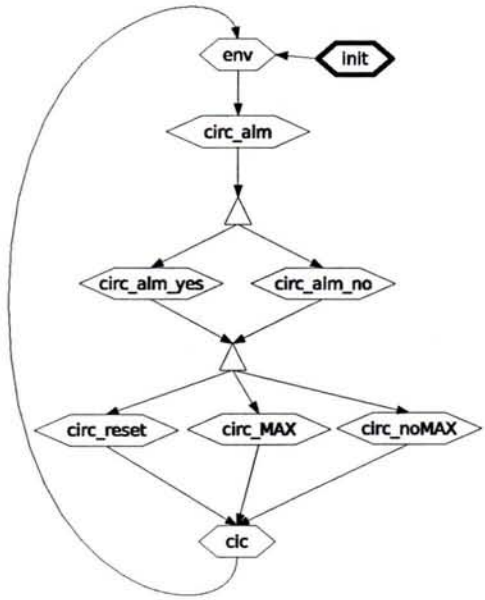
3.5.4 Nouvelle implantation

On pourrait implanter directement le modèle obtenu à la section précédente, mais en observant les événements "circ_alm_yes"et "circ_alm_no", on se rend compte que la valeur de la variable d'alarme alm est exactement la même que la valeur de $tabcount(MAX)$. On fait un nouveau raffinement pour exprimer cela. Nous introduisons le nouvel événement "circ_alm" qui a en fait pour but de remplacer "circ_alm_yes"et "circ_alm_no" qui se retrouvent vides. L'introduction de ce nouvel événement nécessite de raffiner l'ordonnancement, cela se fait en remplaçant la variable $PCIRC1$ par les variables $PCIRC5$ et $PCIRC6$. Le graphe d'événements du nouveau niveau d'implantation est donné sur la figure 3.8.

VARIABLES
$rst, PENV, PCIRC3, PCIRC4,$
$alm3,$
$PCIRC5,$
$PCIRC6$
INVARIANT
$alm3 : BOOL \wedge$
$PCIRC5 \subseteq JET0 \wedge$
$PCIRC6 \subseteq JET0 \wedge$

$(PCIRC5 = \varnothing \vee PCIRC6 = \varnothing) \wedge$
$PCIRC5 \cup PCIRC6 = PCIRC1 \wedge$
$(PCIRC3 = \{jeton\} \Rightarrow alm2 = alm3) \wedge$
$(PCIRC6 = \{jeton\} \Rightarrow alm3 = tabcount(MAX))$
VARIANT
$card(PCIRC5)$

FIG. 3.8 – Graphe d'événements du deuxième niveau d'implantation



INITIALISATION
$rst :\in BOOL \|$
$alm3 :\in BOOL \|$
$PENV := \{jeton\} \|$
$PCIRC3 := \varnothing \|$
$PCIRC4 := \varnothing \|$
$PCIRC5 := \varnothing \|$
$PCIRC6 := \varnothing \|$
$tabcount := \lambda k.(k = 0 | TRUE) \cup \lambda k.(k : 1..MAX | FALSE)$

env =
WHEN
$PENV = \{jeton\}$
THEN
$rst :\in BOOL \|$
$PENV := \varnothing \|$
$PCIRC5 := \{jeton\}$
END

circ_alm =
WHEN
$PCIRC5 = \{jeton\}$
THEN
$alm3 := tabcount(MAX) \|$
$PCIRC5 := \varnothing \|$
$PCIRC6 := jeton$
END

circ_alm_yes =
WHEN
$PCIRC6 = \{jeton\} \wedge$
$tabcount(MAX) = TRUE$
THEN
$PCIRC6 := \varnothing \|$
$PCIRC3 := \{jeton\}$
END

circ_alm_no =
WHEN
$PCIRC6 = \{jeton\} \wedge$
$tabcount(MAX) = FALSE$
THEN
$PCIRC6 := \varnothing \|$
$PCIRC3 := \{jeton\}$
END

circ_reset =
WHEN
$PCIRC3 = \{jeton\} \wedge$
$rst = TRUE$
THEN
$tabcount := \lambda k.(k = 0|TRUE) \cup \lambda k.(k : 1..MAX|FALSE) \|$
$PCIRC3 := \varnothing \|$
$PCIRC4 := \{jeton\}$
END

circ_MAX =
WHEN
$PCIRC3 = \{jeton\} \wedge$
$tabcount(MAX) = TRUE \wedge rst = FALSE$
THEN
$PCIRC3 := \varnothing \|$
$PCIRC4 := \{jeton\}$
END

circ_noMAX =
WHEN
$PCIRC3 = \{jeton\} \wedge$
$tabcount(MAX) = FALSE \wedge rst = FALSE$
THEN
$\lambda k.(k = 0|FALSE) \cup \lambda k.(k : 1..MAX|tabcount(k-1)) \|$
$PCIRC3 := \varnothing \|$
$PCIRC4 := \{jeton\}$
END

circ =
WHEN
$PCIRC4 = \{jeton\} \wedge$
THEN
$PCIRC4 := \varnothing \|$
$PENV := \{jeton\}$
END

Le modèle obtenu ici en B correspond à l'exemple du modèle VHDL donné sur la figure 1.2 page 14. Le modèle BHDL correspondant est donné sur la figure 3.9.

3.5.5 Conclusion sur l'exemple du compteur

Nous avons détaillé ci-dessus l'exemple d'un développement d'un compteur. Cet exemple est trivial c'est pourquoi le développement a presque commencé directement au niveau de l'implantation, d'où la nécessité d'utiliser des variables d'ordonnancement. On a pu voir qu'à chaque fois qu'on introduit un nouvel événement il est nécessaire de raffiner l'ordonnancement, bien que cela se fasse de manière

FIG. 3.9 – Modèle BHDL d'un compteur

MACHINE
  counter
SEES
  $BHDL, PARAM$
INPUTS
  rst
OUTPUTS
  $alm3$
VARIABLES
  $tabcount$
INVARIANT
  $rst \in BOOL \wedge alm3 \in BOOL \wedge tabcount \in TYPETABCOUNT$
INITIALISATION
  $rst :\in BOOL \|$
  $alm3 :\in BOOL \|$
  $tabcount := \lambda k.(k = 0 | TRUE) \cup \lambda k.(k : 1..MAX | FALSE)$
OPERATIONS

  BEGIN
    $alm3 := tabcount(MAX)$
    ;
    IF $rst = TRUE$ THEN
      $tabcount := \lambda k.(k = 0 | TRUE) \cup \lambda k.(k : 1..MAX | FALSE)$
    ELSE
      IF $tabcount(MAX) = FALSE$ THEN
        $tabcount := \lambda k.(k = 0 | FALSE) \cup \lambda k.(k : 1..MAX | tabcount(k-1))$
      END
    END
  END
END

systématique, c'est une complication dont on préfère se priver. c'est pourquoi, d'une manière générale, les modèles sont développés le plus possible à des niveaux très abstraits, où les variables d'ordonnancement ne sont pas utiles, et descendre au niveau de l'implantation que lorsque cela s'avère nécessaire.

3.6 Conclusion

Ce chapitre a présenté la méthode B événementiel en présentant d'abord le langage puis les obligations de preuve associées à un modèle pour garantir sa correction, ainsi que les obligations de preuve liées à un raffinement. Nous avons montré sur un exemple comment on peut l'utiliser pour modéliser un circuit électronique. Cela a été l'occasion de montrer qu'une fois une implantation atteinte on peut, si nécessaire, continuer à raffiner le modèle pour obtenir une autre implantation.

Deuxième partie

Modélisation de circuits par la méthode B

Chapitre 4

Principes de modélisation

L'objectif de ce chapitre est de présenter les principes généraux de modélisation dans le développement de circuits électroniques synchrones par la méthode B.

La plupart des langages de description de circuits (comme VHDL ou SystemC) utilise au niveau synthétisable RTL des modèles basés sur des affectations concurrentes de signaux. Une telle représentation d'un circuit est une vue structurelle du circuit : on énumère les composants et les fils composant le circuit et on décrit de quelle façon ils sont connectés. Ensuite une sémantique de simulation est donnée à une telle description et il est nécessaire d'avoir une idée de cette sémantique (généralement basée sur des δ-délais) pour comprendre l'évolution dans le temps du circuit.

Une modélisation en B événementiel possède une sémantique de l'évolution du système dans le temps très simple. Un ensemble d'événements est donné, chacun possédant une condition (sa garde) sous laquelle il peut se déclencher. La sémantique de l'évolution du système est la suivante : on part d'un état initial donné, sur l'ensemble des événements dont la garde est vraie, un seul se déclenche (le choix duquel est non déterministe) et cet événement se produit dans un délai nul. On arrive dans un nouvel état dans lequel on identifie les événements qui ont une garde vraie, un d'entre eux se déclenche, etc.

Un modèle B est une vue comportementale du système (à laquelle on peut ajouter des invariants qui sont des propriétés exprimant une vue statique, du système). Passer d'une modélisation d'un circuit dans un langage traditionnel comme VHDL à un langage comme B événementiel consiste donc à passer d'une vue statique à une vue dynamique du système. Le principal écueil à éviter est de chercher à reproduire dans un modèle B la sémantique de l'évolution d'un système décrit dans un langage traditionnel. Une telle approche mènerait inévitablement à des modèles très complexes totalement inutilisables. Notre approche consiste à montrer comment modéliser de façon simple un circuit électronique synchrone en B. Le chapitre 14 montre comment on peut retrouver des descriptions dans des langages traditionnels de description de circuit à partir de la description que nous proposons en B. Ce chapitre introduit les concepts de base de notre approche de la modélisation de circuits en B, les chapitres suivants entreront dans les détails.

Modéliser un système par la méthode B nécessite de décider quels sont les événements qui sont observés de l'extérieur. Le système peut être modélisé à différents niveaux d'abstraction. On peut modéliser une vue très abstraite en oubliant la nature physique du système. On peut aussi modéliser le système de façon très détaillée.

Notre démarche consiste à modéliser le système à un niveau d'abord très abstrait. A ce niveau le système n'est pas décomposé en sous-systèmes, les propriétés essentielles du système sont mises en valeur. Le modèle abstrait est ensuite raffiné dans un modèle plus concret dans lequel le système est décrit de manière plus détaillée, en faisant par exemple apparaître les différents modules fonctionnels et la façon dont ils interagissent. Le modèle est raffiné de plus en plus jusqu'à obtenir un modèle *implantable*. Nous considérerons qu'un modèle est à un niveau implantable lorsqu'il est au niveau RTL (Register Transfert Level, niveau transfert de registres), qui peut être pris en entrée par un outil de synthèse automatique produisant la description physique du circuit.

4.1 Hypothèses de modélisation

4.1.1 Propagation des signaux dans un délai nul

La principale hypothèse de modélisation que nous faisons est celle de la propagation dans un délai nul des signaux électriques. Ainsi, lorsque la valeur d'un fil du circuit est modifiée (le fil est une sortie d'un composant par exemple), la nouvelle valeur peut être vue immédiatement par tous les autres composants connectés à ce fil. C'est également l'hypothèse qui est faite dans les langages synchrones.

L'avantage de cette hypothèse est de permettre une modélisation des circuits électriques plus simple en B. Si nous voulions modéliser la propagation des signaux dans des délais non nuls nous pourrions modéliser le circuit en utilisant des δ-délai comme cela est fait en VHDL ou en SystemC. Le principe d'utilisation des δ-délai consiste à voir un ensemble d'affectations concurrentes de signaux comme un ensemble d'équations. La résolution de ce système se fait à la manière d'un point fixe : à chaque pas (appelé δ-délai) du calcul, la valeur de chaque signal est modifié en fonction de l'équation correspondante. Une fois que le calcul ne mène à aucune modification supplémentaire des variables, on considère que le système est stabilisé et que les signaux ont leurs valeurs finales. Par exemple, prenons le morceau de code suivant, écrit à la manière de VHDL. L'opérateur $\Leftarrow$ est une affectation de signal et l'opérateur ; est la mise en concurrence des deux affectations (il ne s'agit pas d'une composition séquentielle). Le code ci-dessous signifie que l'on a deux affectations concurrentes, une attribuant la valeur 2 au signal x et l'autre affectant au signal y la valeur du signal x.

$$x \Leftarrow 2; y \Leftarrow x$$

On fait l'hypothèse que le signal x a initialement la valeur 0 et y a la valeur 1. Dans une première passe, la valeur 2 est assignée x et la valeur de x, c'est-à-dire 0 est assignée à y. Après un premier δ-délai, le signal x a la valeur 2, et y la valeur 0. Une deuxième passe donne au signal x la valeur 2 et à y la valeur 2 (la valeur initiale de x avant la deuxième passe). Une troisième passe conduit aux mêmes valeurs pour les deux signaux. On dit que le système est stabilisé.

Modéliser un tel comportement en B nécessiterait de modéliser l'enchaînement des δ-délai jusqu'à la stabilisation et de prouver l'absence de cycle dans la relation entre les variables. L'hypothèse de délai de propagation nul, nous permet de modéliser l'évolution des signaux comme dans un langage de programmation classique : une fois la valeur d'une variable calculée on sait qu'elle ne changera pas même si une des variables à partir de laquelle sa nouvelle valeur a été calculée à été modifiée.

Cette hypothèse de délai de propagation nul amène cependant certains inconvénients : il n'est a priori pas possible de modéliser simplement des circuits tirant parti du délai de propagation non nul des signaux dans un circuit réel. Pour les circuits synchrones, qui sont ceux auxquels nous nous intéressons, les seuls éléments concernés par ce problème sont les registres, nous expliquons dans la suite comment nous pouvons tout de même modéliser les registres.

4.1.2 Synchronicité des composants

Le système modélisé peut contenir plusieurs composants électroniques, chacun ayant une entrée correspondant à l'horloge. Rien interdit en général d'attribuer à chaque composant une horloge différente, cela risque cependant de compliquer les communications entre les composants. En général, il existe dans le système une horloge de référence par rapport à laquelle tous les composants se synchronisent. Un composant peut par ailleurs avoir en interne une horloge *dérivée* de l'horloge de référence. Par exemple un composant dont le temps d'un cycle de référence ne suffirait pas à compléter ses calculs peut se donner une horloge interne deux fois moins rapide pour se donner plus de temps.

Dans l'approche décrite dans ce document, nous faisons l'hypothèse que tous les composants se réfèrent à une même horloge et sont donc toujours dans le même cycle à tout moment. Dans un système dont les composants communiquent sur des distances relativement longues, il faut faire l'hypothèse qu'il existe entre eux un moyen permettant de synchroniser leurs horloges (c'est souvent le cas dans les protocoles de communication).

4.1.3 Effet d'un événement en temps nul

Nous rappelons que la méthode B événementielle fait l'hypothèse que la substitution d'un événement s'applique dans un temps nul. C'est-à-dire qu'aucun autre événement ne peut se déclencher en même temps. Ceci implique que si un système est composé de plusieurs composants électroniques fonctionnant en concurrence et communiquant entre eux, il ne sera pas possible de modéliser un composant par un unique événement. Par exemple, un composant A peut nécessiter une valeur produite par un composant B pour produire une valeur utilisée par le composant A. Si le composant A était modélisé par un unique événement, l'événement du composant B ne pourra pas se produire avant la fin de l'événement du composant A. Pour pouvoir obtenir une modélisation avec un seul événement par composant il serait nécessaire de recourir à une approche avec δ-délai qui a été discutée plus haut.

Dans un modèle B, un composant sera donc constitué d'un ensemble d'événements. À la fin du développement, ces événements seront regroupés en un seul pour obtenir le code final qui sera traduit vers un langage classique de description de circuit.

4.2 Cycle et vue dynamique du système

Le comportement d'un système composé de circuits électroniques synchrones est cyclique. A chaque cycle, le comportement de chaque composant doit être modélisé (éventuellement par *skip* la substitution qui ne modifie par l'état, si le composant n'a rien à faire). Les cycles sont marqués par une entrée particulière du circuit, l'horloge, qui passe régulièrement de '0' à '1' (front montant) et de '1' à '0' (front descendant). Un cycle correspond à l'intervalle entre deux fronts montants du signal d'horloge. Dans certains cas, le circuit peut être sensible aux fronts descendants plutôt qu'aux fronts montants.

Dans le modèle B, nous avons choisi de ne pas modéliser explicitement le signal d'horloge, on modélise seulement le comportement cyclique du système. Ceci permet de s'abstraire de savoir si le circuit est sensible aux fronts montants ou aux fronts descendants. De plus la valeur du signal d'horloge n'est pas utilisée et on évite ainsi de polluer le modèle avec des signaux qui ne sont pas nécessaires à la modélisation des fonctionnalités du système.

Dans le cas simple de l'additionneur donné sur la figure 4.1 page 65 (on trouvera un exemple de développement détaillé d'un additionneur dans [26]), un cycle correspond à l'occurrence de l'événement modélisant l'environnement suivi de l'occurrence de l'événement modélisant l'additionneur. Ensuite un nouveau cycle recommence par l'occurrence de l'événement de l'environnement, etc. Ce comportement cyclique doit être modélisé explicitement dans le modèle B, sans quoi les événements pourraient se produire dans un tout autre ordre. Une façon simple de modéliser ce comportement est d'utiliser une variable s qui passe de '0' à '1' lorsque l'événement modélisant l'environnement se produit et de '1' à '0' lorsque l'événement modélisant l'additionneur se produit. Ceci modélise donc un ping-pong entre les deux événements. La variable s est initialisée à '0' pour que l'événement modélisant l'environnement se déclenche en premier. L'événement modélisant l'environnement a pour charge de fournir des entrées au circuit, c'est pourquoi il doit se déclencher en premier. La variable s ne fait pas partie de la description du circuit, elle est nécessaire à la modélisation en B mais n'apparaît pas dans le circuit physique, il n'y a donc pas de raison pour qu'elle soit d'un type implantable, nous verrons par la suite que nous synchroniserons les événements en utilisant un modèle de réseau de Petri.

Plus généralement, l'ensemble des événements modélise le comportement du circuit pendant un cycle. Cependant il doit prendre en compte tous les cas possibles pouvant se produire dans tout cycle. Cette contrainte est assurée par la preuve de non blocage du système de la méthode B événementiel. En effet, si un cas n'était pas pris en compte, c'est-à-dire qu'il n'existe aucun événement du circuit ne pouvant se déclencher dans ce cas précis, alors le système événementiel se retrouverait bloqué : ni la partie correspondant au circuit (par hypothèse), ni l'événement de l'environnement ne pourraient se produire puisque que l'environnement est en attente que le partie circuit soit terminée.

Dans le cas général, le système est composé de plusieurs composants communiquant entre eux, la modélisation du cycle est alors définie de façon plus générale, ceci est expliqué plus loin dans le chapitre 5.

4.3 Observation du circuit

Un circuit électronique est constitué de composants reliés entre eux par des fils électriques. Nous ne souhaitons pas descendre à un niveau aussi bas de description pour la modélisation de circuits en B. Il existe déjà des langages (comme VHDL, Verilog, SystemC) de description de circuit permettant de manipuler des nombres ou des vecteurs, l'outil de synthèse se chargeant de transformer automatiquement ces types de données en nappes de fils. Notre objectif est d'obtenir des modèles de systèmes qui puissent être traduits automatiquement vers des langages comme VHDL puis synthétisés.

Le niveau de détail que nous visons est d'observer les valeurs qui sont portées par les fils, représentées par des structures plus abstraites comme des entiers ou des vecteurs. C'est à dire que l'état du modèle B sera constitué de variables représentant les valeurs portées par les fils. Les événements exprimeront de quelle manière les valeurs des variables sont modifiées par les composants. Ainsi, les événements représenterons le "code" (au sens VHDL) des composants alors que les variables représenteront les fils du circuit.

Pour illustrer cela nous prenons l'exemple d'un additionneur. Il prend en entrée deux nappes de fils représentant les deux nombres à additionner et en sortie une nappe de fils représentant le résultat de l'addition et un fil correspondant à la retenue sortante de l'additionneur. Une représentation de ce circuit est donnée sur la figure 4.1. Dans le modèle B, les deux nappes de fils d'entrée sont modélisées par deux variables a et b, la nappe de fils de sortie est modélisée par une variable c et la retenue sortante par la variable r. Si on suppose qu'il s'agit par exemple d'un additionneur 4 bits, les nappes de fils sont composées de 4 fils, donc les variables a, b et c sont des entiers dans l'ensemble 0..15. Il n'est pas nécessaire de descendre à un niveau de modélisation plus concret où les nappes de fils sont représentées par des vecteurs de bits car les outils de synthèse sont capables de les générer automatiquement à partir de données de type "nombre entier". La fonctionnalité est représentée dans un événement par une substitution qui rend la variable c égale à la somme de a et b, et rend la variable r égale à la valeur de la retenue sortante. L'expression $(a+b)\ mod\ 16$ (qui équivaut à $a+b$ lorsqu'il n'y a pas dépassement et à $a+b-16$ en cas de dépassement) modélise le fait que si la somme n'est pas représentable sur 4 bits, le dernier bit est perdu et se retrouve dans la retenue sortante (le résultat est donc faux et cela est signalé par une retenue sortante qui vaut 1). La façon dont la fonctionnalité de l'additionneur est modélisée ici ne présume pas de la façon dont l'additionneur sera implanté en réalité, on s'attend uniquement à ce que le lien entre les entrées et les sorties soit le même que celui spécifié. Dans ce cas précis, on réutilisera probablement un additionneur faisant partie d'une bibliothèque standard. La garde de l'événement a pour fonction de spécifier à quel moment l'événement doit pouvoir se déclencher, ceci est fait en relation avec la modélisation de l'environnement du circuit (celui-ci doit avoir fourni les valeurs des variables a et b).

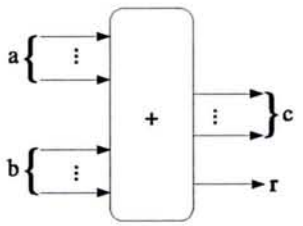
Remarquons que dans le modèle B, aucune différence n'est explicitement faite entre les entrées et les sorties d'un circuit. La différence principale réside dans leur utilisation dans la description des événements : une entrée ne doit pas être modifiée par le circuit. Cette contrainte est ajoutée ensuite comme une contrainte de bonne formation du modèle au niveau du langage BHDL, mais n'est pas présente dans un modèle B événementiel. C'est donc une contrainte de modélisation que le modélisateur doit s'imposer pour être certain d'obtenir finalement un circuit implantable. Une façon de l'imposer est de faire apparaître les variables représentant les entrées et les sorties des composants dès le premier modèle abstrait. Dans ce modèle, les variables d'entrée n'étant pas modifiées par le modèle du composant, aucun raffinement prouvé correct n'aura la possibilité de les modifier. Les seules parties du système ayant la possibilité de modifier les variables d'entrée sont les parties du système qui fournissent ces entrées, en l'occurrence l'environnement dans cet exemple.

Circuits synchrones non combinatoire

L'exemple de l'additionneur utilisé comme illustration sur la figure 4.1 est un cas de circuit particulier puisqu'il est purement combinatoire. Dans le circuit physique, les signaux électriques des entrées se propagent dans le circuit et des signaux sont récupérés en sortie. La relation entre les sorties et les entrées est toujours la même.

D'une manière plus générale, un circuit synchrone contient des éléments de mémorisation, appelés registres. A chaque coup d'horloge, un registre enregistre le signal qui lui est proposé en entrée et le

FIG. 4.1 – Exemple d'un additionneur



additionneur =
ANY carry WHERE
$s = 1 \wedge$
$a + b > 15 \Rightarrow carry = 1 \wedge$
$a + b \leq 15 \Rightarrow carry = 0 \wedge$
THEN
$c := (a + b)\ mod\ 16 \parallel$
$r := carry \parallel$
$s := 0$
END

environnement =
WHEN
$s = 0$
THEN
$a :\in 0..15 \parallel$
$b :\in 0..15 \parallel$
$s := 1$
END

Avec l'initialisation $a, b, c, s := 0, 0, 0, 0$, on peut écrire l'invariant suivant : $s = 0 \Rightarrow c = a + b\ mod\ 16$

délivre de manière constante pendant toute la durée du cycle qui commence. Au niveau physique un registre est un composant comme un autre et peut se modéliser sans problème dans des langages de description de circuits comme VHDL. L'implantation d'un registre est un court-circuit astucieux reliant la sortie du registre à sa propre entrée, en tirant profit du délai de propagation non nul des signaux électriques dans un circuit physique.

Un registre ne peut donc pas être modélisé de cette façon dans notre méthodologie de modélisation B car les modèles que nous écrivons font l'hypothèse d'une propagation dans un délai *nul* des signaux électriques. Comme nous l'avons déjà dit, une modélisation B consiste à choisir quels sont les éléments que nous décidons d'observer. Dans le cas d'un registre, nous décidons d'observer les valeurs d'entrée et de sortie du registre. La façon dont le registre est physiquement réalisé n'est pas modélisée en B. Par ailleurs, l'unité de temps que nous considérons est celui du cycle. De ce point de vue, le comportement d'un registre est de stocker une valeur d'un cycle à l'autre. Autrement dit, la valeur d'un registre reste inchangée pendant la durée du cycle, et sa valeur est celle calculée lors du cycle précédent (ou est égale à la valeur du registre du cycle précédent si aucune nouvelle valeur n'a été spécifiée). La solution adoptée pour modéliser un registre est d'utiliser une variable dont la valeur est la valeur stockée par le registre. Ainsi, tant que cette variable n'est pas modifiée à l'intérieur d'un cycle, elle représente la valeur du fil de sortie du registre. Si le modèle modifie la valeur de cette variable, alors sa nouvelle valeur est celle qu'aura le registre au prochain cycle ; la valeur stockée par le registre n'est alors plus accessible, à moins d'avoir utilisé une variable supplémentaire pour représenter explicitement la valeur du fil de sortie du registre. Après modification pendant le cycle de la valeur de la variable représentant le registre, celle-ci modélise le fil d'entrée du registre.

Ce choix de représenter un registre par une seule variable (une variable pouvant être considérée comme un élément de mémorisation d'un modèle) nous semble correspondre à la fonction de mémorisation du registre. Une alternative serait d'utiliser deux variables, une modélisant le fil de sortie et une autre le fil d'entrée. Ceci reviendrait à modéliser le registre de manière explicite et nécessiterait de modéliser le registre comme un composant du système en écrivant une substitution réalisant la mise à jour des valeurs des fils en début de cycle. Un système réaliste pouvant comporter un nombre important de registres, une telle représentation mènerait à une complication inutile des modèles (et des preuves).

4.4 Modélisation de l'environnement

Un modèle B est par principe un modèle fermé, c'est-à-dire qu'il n'y a pas de communication avec l'extérieur. Il est donc nécessaire de modéliser l'environnement comme une partie intégrante du système modélisé. Ceci permet en outre de rendre explicite les hypothèses qui sont faites sur l'environnement. L'environnement peut être modélisé comme un composant ordinaire ayant pour tâche de produire des entrées pour le système.

Dans la plupart des cas, les entrées du système ne peuvent pas être complètement déterminées (on

ignore qu'elles sont les entrées qui seront fournies par l'environnement), ou plus exactement on souhaite prouvé que le système à un comportement correct pour toute entrée appartenant à un certaine classe admissible. Pour modéliser cela on utilise une substitution non déterministe. Le plus simple est de définir un ensemble E des entrées possibles et d'écrire $in :\in E$ pour spécifier que l'entrée in appartient à l'ensemble E. On peut aussi utiliser la forme plus générale $in : (P)$ où P est un prédicat spécifiant les propriétés que in doit satisfaire. On a aussi la possibilité d'utiliser la forme ANY v WHERE Q THEN $in := v$ END d'un événement, où Q contient les propriétés que doit satisfaire v (et par conséquent in).

Dans l'exemple de l'additionneur, il nous faut dire que les entrées a et b sont des nombres entiers appartenant à l'intervalle 0..15, l'événement modélisant l'environnement est montré sur la figure 4.1.

La nécessité de modéliser l'environnement du système est un avantage apporté par la méthode B événementielle. Dans l'approche B "classique" un système est modélisé par un ensemble d'opérations appelées de l'extérieur. Pendant le processus de modélisation on peut ajouter des préconditions à chaque opération. Une précondition étant une hypothèse faite sur les variables de la machine appelée et sur les paramètres d'entrée de l'opération concernée ; elles ne peuvent pas prendre en compte l'état du module extérieur appelant. Ces préconditions disparaissent au niveau de l'implantation et on suppose que l'environnement est en accord avec ces hypothèses. On peut imaginer récupérer les hypothèses faites et les prendre comme assertions à vérifier sur le système lors de l'intégration du composant dans son environnement. Dans l'approche B événementielle, on ne dispose pas seulement de conditions à vérifier portant sur l'état du composant développé formellement, mais d'un modèle, très abstrait en général, de l'environnement. On dispose ainsi des contraintes à vérifier sur l'environnement lui-même s'il a été développé par ailleurs, ou de contraintes à ajouter à la spécification pour le développement de celui-ci. On peut donc identifier clairement les conditions nécessaires pour qu'un composant développé en B événementiel puisse s'intégrer sainement dans un contexte donné.

4.5 Le raffinement

Un point important de notre approche de la modélisation de circuits est celui du raffinement. Dans notre approche un circuit n'est pas modélisé directement au niveau implantable mais est d'abord décrit de manière abstraite. Les modèles abstraits correspondent à la spécification du système.

Les modèles abstraits ont l'avantage d'être plus concis que des modèles implantables car ils ne décrivent pas tous les détails d'une implantation. Ceci permet d'une part d'avoir des modèles sur lesquels il est plus simple d'exprimer des propriétés sur le système et plus facile de faire les preuves mais aussi des modèles qui peuvent servir de support de communication entre la personne en charge de la modélisation et le donneur d'ordre.

Les modèles abstraits permettent d'exprimer les différents éléments du système de manière directe (comme des protocoles, cf. chapitre 7), sans se soucier dans un premier temps de la façon dont ces éléments seront implantés.

Une première boucle d'interaction entre le donneur d'ordre et la personne écrivant les modèles abstraits est donc possible au niveau de la spécification. Ceci permet d'éviter d'avoir à attendre qu'une implantation soit réalisée pour se rendre compte d'une erreur d'interprétation de la spécification ou d'une modification de cette dernière par le donneur d'ordre..

Une fois obtenu un modèle abstrait du système satisfaisant le donneur d'ordre, le système peut être raffiné pas à pas, en introduisant les détails d'une implantation au fur et à mesure. Ceci permet de gérer la complexité du système (et des preuves associées) car il n'est pas nécessaire d'introduire tous les détails d'un seul coup.

Le raffinement se termine par un modèle dit implantable (cf. chapitre 5), c'est-à-dire un modèle qu'on soit capable de traduire vers des langages classiques de description de circuit, à partir desquels la synthèse vers un circuit électronique est possible.

Notre expérience dans la modélisation du contrôleur de bus du chapitre 7, nous a montré qu'il est possible de continuer à raffiner un modèle implantable pour obtenir un nouveau modèle implantable respectant certaines contraintes d'implantation (en l'occurrence le nombre de signaux d'entrée étaient trop élevés dans le premier modèle obtenu).

Les deux événements de l'exemple de l'additionneur de la figure 4.1 peuvent par exemple se raffiner dans les quatre événements suivants. Le principe de ce raffinement est de faire sortir le calcul de la retenue de l'événement *additionneur* en créant deux nouveaux événements *carry_0* et *carry_1*. Pour bien synchroniser tous les événements on raffine la variable s vers la variable t. L'invariant de collage est le suivant :

$$(t = 0 \Rightarrow s = 0 \ \wedge \ t \in \{1, 2\} \Rightarrow s = 1) \ \wedge$$
$$(t = 1 \Rightarrow ((a + b > 15 \Rightarrow carry = 1) \ \wedge \ (a + b \leq 15 \Rightarrow carry = 0)))$$

carry_0 =
WHEN
$t = 1 \wedge$
$a + b \leq 15$
THEN
$t := 2 \parallel$
$carry := 0$
END

carry_1 =
WHEN
$t = 1 \wedge$
$a + b > 15$
THEN
$t := 2 \parallel$
$carry := 1$
END

additionneur =
WHEN
$t = 2$
THEN
$c := (a + b)\ mod\ 16 \parallel$
$r := carry \parallel$
$t := 0$
END

environnement =
WHEN
$t = 0$
THEN
$a :\in 0..15 \parallel$
$b :\in 0..15 \parallel$
$t := 1$
END

4.6 Parallélisme

L'approche choisie du délai nul de propagation des signaux électriques dans le circuit, plutôt que l'approche traditionnelle par affectations concurrentes de signaux, pourrait nous empêcher de modéliser le parallélisme des circuits. En fait la méthode B propose deux moyens de modéliser ce parallélisme.

D'une part on peut utiliser l'opérateur $\parallel$ de composition parallèle du langage B. Contrairement à ce qui est dans le développement de logiciels en B, cet opérateur de composition n'a pas besoin de disparaître au niveau de l'implantation puisqu'il est parfaitement implantable en circuit électronique. Cet opérateur modélise directement une véritable composition en parallèle de deux blocs de codes et les règles de bonne formation d'un modèle B (interdiction de modifier une même variable dans les deux parties composées en parallèle) sont bien adaptées à l'utilisation de cet opérateur pour la modélisation de circuits. Cependant il ne peut être présent qu'à l'intérieur d'un événement pour composer deux substitutions ; il ne peut pas être utilisé pour mettre en parallèle deux composants (un composant étant représenté par un ensemble d'événements).

Pour modéliser la mise en parallèle de deux composants, on profite du non déterminisme dans le déclenchement des événements de la méthode B pour modéliser le parallélisme par une concurrence. Utiliser la concurrence pour modéliser le parallélisme est une approche traditionnelle dans la description de circuits, c'est le cas en VHDL et en SystemC. Si on souhaite mettre en parallèle deux composants, il suffit de ne pas les synchroniser entre eux, c'est-à-dire que leurs événements pourront s'intercaler les uns aux autres. Les preuves faites sur le modèle sont valables pour tout ordre de déclenchement des événements, ainsi on prouve non seulement la correction du parallélisme mais également que les deux composants peuvent effectivement être mise en concurrence et que le résultat qu'on obtient est toujours le même.

4.7 B : asynchrone par nature

Un modèle B est asynchrone par nature. C'est-à-dire que les événements peuvent a priori se déclencher dans n'importe quel ordre (en fonction de leurs gardes et un seul à la fois).

Il faut donc agrémenter le modèle de façon à synchroniser les composants lorsque cela est nécessaire. Par exemple, si un composant A utilise les sorties d'un composant B il faudra s'assurer que les événements du composant A se produisent après les événements du composant B.

De plus, dans la modélisation de systèmes synchrones, quand le système comporte plusieurs composants, il faut s'assurer que tous les composants se trouvent dans le même cycle d'horloge, il ne faut pas qu'un composant puisse avancer plus vite dans le temps que les autres.

Pour modéliser cela, nous mettons en place un ordonnancement dans le système. Nous proposons d'utiliser des modèles de réseaux de Petri pour ordonnancer les événements entre eux. Les variables participant à cet ordonnancement, et modélisant le réseau de Petri, ne font pas partie intégrante de la description du circuit. Cet ordonnancement exprime en fait une partie de l'architecture du système :

- lorsque deux composants sont en parallèle, l'ordonnancement laisse libre à la concurrence les événements des deux composants
- lorsque la sortie d'un composant est connectée à l'entrée d'un autre, l'ordonnancement exprime une séquentialité dans l'ordre de déclenchement des événements
- l'horloge synchronisant tous les composants est modélisée par un ordonnancement général assurant le caractère cyclique du modèle.

Ainsi, les variables nécessaires à la modélisation de l'ordonnancement ne sont pas traduites telles qu'elles en VHDL, mais on peut dire que l'ordonnancement est implanté par l'architecture du système. Cet ordonnancement explicite, modèle de l'architecture, n'est donc nécessaire qu'au niveau où on se rapproche de l'implantation. On peut a priori s'en passer dans les modèles plus abstraits du système où l'ordonnancement est beaucoup plus implicite et porte directement sur des variables (en général très abstraites) du modèle du système. Le processus de raffinement, outre le raffinement des fonctionnalités de chaque composant, aura donc également pour but de raffiner cet ordonnancement de façon à ce qu'il corresponde à l'architecture du système.

4.8 Spécification et modèles abstraits

Modéliser un système nécessite de décider quels sont les événements qui sont observés de l'extérieur. Le système peut être modélisé à différents niveaux d'abstraction. On peut modéliser une vue très abstraite en oubliant la nature physique du système. On peut aussi modéliser le système de façon très détaillée.

Un circuit électronique est constitué de composants connectés entre eux. Des courants électriques se propagent sur les fils et traversent les composants. Il semble évident qu'il ne sera pas nécessaire de descendre à un niveau d'abstraction aussi bas pour décrire le comportement d'un circuit.

Il existe déjà des langages de description de circuits (VHDL, SystemC ...) à partir desquels des outils permettent la *synthèse* de la description très bas niveau. Notre objectif est d'obtenir un modèle du système qui puisse être traduit automatiquement vers des langages comme VHDL puis synthétisé.

La modélisation abstraite de circuits ne diffère pas de la modélisation abstraite en B en général. Dans cette section, nous discutons de certains éléments qui ont pu nous paraître utiles ou spécifiques dans la modélisation abstraite de circuits électroniques.

4.8.1 Abstraction de la spécification

Les spécifications de circuits électroniques sont généralement données à un niveau très bas. Par exemple, dans le cas du standard SAE J1708 que nous avons étudié la description est faite au niveau du bit et même au niveau physique pour certains éléments.

Pour un constructeur de modèles B, ce niveau de description est extrêmement bas. Il constitue une cible pour le raffinement mais n'est pas un point de départ. Un premier travail, important, consiste à analyser la spécification pour l'abstraire.

Pour cela, on peut commencer par mettre au point un glossaire des termes utilisés par la spécification de façon à utiliser un vocabulaire précis. Cette étape permet également de distinguer quels sont les éléments constituant le système que l'on souhaite modéliser.

En général, le sens donné aux termes dans la spécification est donné en utilisant des notions de bas niveau. Une deuxième étape peut donc consister à identifier les termes auxquels on peut donner un sens abstrait. Ceci permet d'identifier dans quel ordre les différentes notions du système seront introduites dans la modélisation le long de la chaîne de raffinements. Petit à petit il faut être capable d'avoir une vision la plus abstraite possible du système. C'est pendant cette étape, revenant à donner un sens plus humain aux éléments du système, que peuvent apparaître les problèmes d'interprétation de la spécification.

Enfin, il faut dégager le plus de propriétés possibles du système. En effet, les spécifications de circuits sont rarement très verbeuses au sujet des propriétés que doit posséder le circuit. Il faut donc lire entre les lignes de la spécification pour en ressortir des propriétés qui constitueront le socle sur lequel le processus de modélisation se fera.

4.8.2 Modélisation de la spécification

Une modélisation en B ne commence généralement pas à un niveau d'implantation mais à un niveau de spécification. A ce niveau on modélise le système de façon abstraite de façon à le spécifier, on décrit les fonctionnalités et les propriétés importantes du système.

Après analyse, la spécification est modélisée par des modèles abstraits du système. En général on ne fait pas un seul modèle mais un premier modèle représentant la fonctionnalité principale du système puis des raffinements expliquant plus en détail le comportement du système.

A ce niveau on ne doit normalement pas se préoccuper de la façon dont les choses seront implantées mais plutôt de la façon dont les choses seront comprises le plus facilement pour avoir des aller-retour entre la personne chargée de la modélisation et le donneur d'ordre, ainsi que pour faciliter les preuves qui doivent être faites.

Les preuves principales se font au niveau des modèles abstraits. Les preuves faites aux niveaux plus concrets ont pour but de prouver le raffinement ; c'est-à-dire que la façon dont on implante les choses correspond bien à la spécification (aux modèles abstraits).

4.8.3 Environnement

Un modèle B événementiel est par définition un modèle fermé, il n'y a pas d'interaction avec des événements extérieurs au modèle. Il faut donc que l'environnement du système fasse partie du modèle. Dans la modélisation de circuits, l'environnement a pour charge de fournir les entrées du circuit.

Au niveau implantable il faut modéliser l'environnement explicitement de façon à ce que tous les composants soient modélisés séparément (l'environnement est alors considéré comme un composant du système. L'environnement possède alors ses propres événements (on utilise toujours un événement appelé *env* dans ce document pour modéliser l'environnement.

Au niveau abstrait cela peut compliquer le modèle de modéliser l'environnement ainsi, car cela nécessite en général d'ajouter un ordonnancement explicite pour s'assurer que l'événement de l'environnement se produit au bon moment. Une autre solution, qui s'avère plus efficace en général, pour acquérir de nouvelles entrées de l'environnement, est de tirer parti de la forme ANY que peut prendre un événement. Par exemple, si on souhaite utiliser une entrée i de type entier et que l'on doit l'additionner avec une variable interne x pour produire la nouvelle valeur d'une variable y, on peut écrire l'événement sous la forme suivante :

```
ANY i WHERE
  i ∈ ℕ
  ... /* reste de la garde */
THEN
  y := i + x
END
```

Il faut faire attention en utilisant cette technique si on doit utiliser l'entrée i plusieurs fois. En effet, si on écrit plusieurs événements sous cette forme, les valeurs choisies pour i seront a priori différentes alors qu'une valeur d'entrée est supposée constante pendant la durée d'un cycle. Pour éviter ce problème, on peut se rappeler la valeur de i en utilisant une variable. Pour que cela fonctionne correctement, il faut bien identifier l'événement qui est en charge de choisir la valeur de i et s'assurer que celui-ci se produit bien avant les autres événements qui veulent utiliser cette valeur. Ceci est une abstraction de la modélisation utilisant l'événement *env* permettant en général de se passer d'un ordonnancement explicite.

4.8.4 Événement nothing

Lors des études de cas il nous est apparu nécessaire de faire en sorte que le comportement de tous les composants du système soit modélisé dans tous les cas de figure, y compris lorsqu'un composant n'a pas de calcul à effectuer. Cela vient du fait que dans un circuit physique, les signaux électriques se propagent dans tous les composants du circuit, y compris dans ceux dont le résultat n'est pas utilisé (pour des entrées données).

Pour assurer cela, lorsque le premier modèle écrit ne prend pas en compte directement cette contrainte, nous ajoutons un événement, que nous appelons *nothing* dans ce document, qui modélise le comportement des composants dans les cas non modélisés. Cet événement possède la seule substitution *skip*, c'est à dire qu'il n'a aucune influence sur la fonctionnalité du système. Cela peut sembler inutile de prime abord car on pourrait penser n'introduire cet événement qu'au niveau de l'implantation. Nous avons cependant jugé utile de prendre en compte le fait que le comportement de chaque composant soit modélisé dans tous les cas dès les modèles abstraits car ceci contraint dès le départ le modèle du système de façon à ce que si on extrait un composant donné, on puisse montrer qu'il ne se bloque pas.

En effet, la bonne formation d'un modèle B événementiel impose que le système ne se bloque pas mais ne dit rien sur les composants du système (qui ne sont d'ailleurs pas identifiés comme tels par la méthode). Dans un modèle de circuit électronique, il nous semble logique, car un composant de circuit ne peut pas se bloquer, d'imposer en plus la contrainte que chaque composant (finalement considéré comme un sous-système) ne doit pas se bloquer. L'événement *nothing* permet d'assurer cela. Si on n'imposait pas cette contrainte, on pourrait se trouver dans la situation où le système global ne se bloque pas mais que certains composants se bloquent, ce qui ne nous semble pas acceptable pour un composant de circuit électronique.

Si le modèle du système prend déjà en compte cette contrainte, il est évidemment inutile d'ajouter un événement *nothing*. C'est le cas par exemple dans l'étude de cas sur la somme de convolution, cf. chapitre 6 ou dans le petit exemple du compteur (chapitre 3 section 3.5). Par contre nous utilisons l'événement *nothing* dans l'étude de cas SAE J1708 (chapitre 7).

4.9 Conclusion

Ce chapitre a introduit différents concepts nécessaires à la modélisation de circuits électroniques par la méthode B. Ces concepts sont développés dans les chapitres suivants. Nous avons vu en particulier que nous faisons des hypothèses sur le système modélisé (délais nuls et synchronicité des composants). Nous avons expliqué que l'approche B est une vue dynamique du système. Modéliser un système en B nécessite de décider quels sont les éléments que l'on observe, nous avons expliqué les choix que nous avons faits pour l'application de la méthode à la modélisation de circuits. En particulier nous avons expliqué qu'il est nécessaire de modéliser l'environnement du circuit.

La modélisation abstraite de circuit ne diffère pas de la modélisation abstraite en général en B. Nous avons cependant vu qu'il était en général nécessaire de faire une analyse de la spécification donnée de façon à l'abstraire car les spécifications de circuit sont en général données à un niveau très bas. Nous avons expliqué comment il est possible de modéliser l'environnement à des niveaux abstraits et les précautions qu'il est nécessaire de prendre. Nous avons expliqué l'intérêt et la nécessité d'avoir un événement au niveau abstrait pour modéliser le comportement des composants dans les cas où ils n'ont pas de calcul à effectuer.

Chapitre 5

Modèle implantable

L'objectif de ce chapitre est d'expliciter par des règles le type de modèle que l'on doit obtenir pour avoir un modèle implantable. Ces règles décrivent la cible du raffinement et pas le raffinement lui-même.

Nous avons déjà précédemment parlé de la nécessité de modéliser l'aspect cyclique du système. Cette nécessité n'intervient que lorsqu'on approche du niveau d'implantation ; les modèles abstraits peuvent oublier l'aspect cyclique pour simplifier les preuves. De la même façon, dans les niveaux abstraits les communications se font simplement car chaque composant a directement accès aux variables modélisant les autres composants. Aux niveaux d'implantation, ceci ne doit plus exister et les communications doivent se faire par des variables identifiées comme étant des entrées ou des sorties des composants. Nous décrivons dans ce chapitre les contraintes que doivent respecter les modèles implantables obtenus par raffinement. Nous expliquons comment modéliser les composants électroniques de manière indépendante les uns des autres, comment les faire communiquer entre eux et modéliser l'aspect cyclique du système en utilisant un ordonnancement dans le déclenchement des événements du modèle B. Nous verrons également comment modéliser des composants qui peuvent travailler en parallèle. Les règles que nous expliquons dans ce chapitre conduisent à un modèle dit implantable ; c'est-à-dire qu'il conduit à obtenir le code BHDL des composants en utilisant la technique de regroupement des événements que nous expliquons en fin de chapitre.

5.1 Séparation des composants

Le système modélisé est généralement composé de plusieurs composants. Dans les langages de description de circuits comme VHDL ou SystemC, chacun des composants est décrit de façon séparée et importé par la description du système global. Cette séparation des composants permet de les développer (et les vérifier) séparément puis de décrire l'architecture globale du système reliant les composants entre eux. Il faut alors vérifier que l'architecture en question, d'une part correspond à la fonctionnalité souhaitée, et d'autre part qu'elle utilise chaque composant dans les limites des hypothèses que celui-ci fait sur son environnement.

L'approche de modélisation formelle par la méthode B que nous avons choisie consiste à modéliser les systèmes dans leur ensemble. Les composants ne sont donc à priori pas décrits de manière indépendante les uns des autres. Aux niveaux d'abstraction élevés le système est doté d'un état global partagé par tous les composants, et un événement peut correspondre à la fonctionnalité de plusieurs composants. Par exemple la substitution $x := a + b - c$ correspond en fait à deux composants : un additionneur qui prend comme entrées a et b, et un soustracteur qui prend comme entrées la sortie de l'additionneur d'une part et c d'autre part. La sortie x correspond en fait à la sortie du soustracteur.

L'objectif est de décrire les composants du système de façon indépendante. Autrement dit, à chaque composant est attribué une collection de variables (son état) et d'événements. Une variable et un événement modélisant l'état et le comportement d'un seul des composants du système. Cette séparation a pour objectif d'identifier les différents composants du système. Ceci prend toute son importance lorsque, par exemple, les différents composants du système sont destinés à être physiquement éloignés les uns des

autres, et donc ne seront pas physiquement implantés sur la même puce.

Règle 1 (Séparation des composants) *Cette règle est à relativiser suivant la règle 2. Chaque variable et chaque événement du système B correspond à un et un seul composant du système. Chaque composant n'a le droit de modifier que ses propres variables. L'environnement du système est considéré comme un composant et doit donc suivre la même règle.*

Nous précisons que chaque composant n'a le droit de modifier que les variables appartenant à son état propre. Nous n'imposons pas ici de contraintes sur la lecture de variables qui ne font pas partie de son état. Dans ce cas il s'agit de la lecture d'une variable appartenant à un autre composant, on dit qu'il y a *communication* entre les composants. Ceci nécessite un certain *ordonnancement* entre les composants, de façon à ce qu'une variable ne soit lue que lorsque sa valeur a été mise à jour.

5.1.1 Cas de plusieurs composants identiques

Il arrive cependant qu'un système soit composé de plusieurs composants identiques. Dans ce cas, il est peu pratique de répéter plusieurs fois le même code. D'autant plus que la répétition complique le modèle et est toujours source d'erreurs. Dans ce cas, on se dote de variables et d'événements qui ne modélisent plus seulement un seul composant mais plusieurs.

Par exemple, prenons un ensemble de composants ayant deux variables *compt* (un entier) et *tic* (variable bivaluée). Chacun des composants pouvant incrémenter *compt* lorsque *tic* vaut 1. On commence par se donner l'ensemble abstrait *COMP* de ces composants (en utilisant la clause SETS en B). Les variables *compt* et *tic* sont définies par des fonctions totales.

cas un seul composant	cas plusieurs composants identiques
$compt \in \mathbb{N}$	$compt \in COMP \rightarrow \mathbb{N}$
$tic \in \{0,1\}$	$tic \in COMP \rightarrow \{0,1\}$

Une initialisation non déterministe se fait en utilisant $:\in$ et une initialisation déterministe avec une lambda expression. Nous donnons ci-dessous des exemples pour la variable *compt*.

cas un seul composant	cas plusieurs composants identiques
$compt :\in \mathbb{N}$	$compt :\in COMP \rightarrow \mathbb{N}$
$compt := 0$	$compt := \lambda c \cdot (c \in COMP \mid 0)$

Si besoin on peut également faire des choses plus compliquées, par exemple si l'on souhaite distinguer deux composants $c1$ et $c2$ que l'on souhaite initialiser différemment des autres. Il faut d'abord commencer par déclarer les deux composants comme des constantes, et pour se simplifier la vie on déclare aussi l'ensemble *AUTRES* qui est l'ensemble *COMP* sauf $c1$ et $c2$. Remarquez que *AUTRES* peut éventuellement être un ensemble vide et si on veut s'assurer que $c1$ et $c2$ désignent deux composants bien distincts il faut ajouter la propriété $c1 \neq c2$. On donne un exemple ci-dessous.

SETS
  COMP
VARIABLES
  x, y, z
CONSTANTS
  c1, c2, AUTRES
PROPERTIES
  $c1 \in COMP \wedge c2 \in COMP \wedge c1 \neq c2 \wedge AUTRES = COMP - \{c1\} - \{c2\}$
INVARIANT
  $compt \in COMP \rightarrow \mathbb{N}$
INITIALISATION
  $compt : (compt \in COMP \rightarrow \mathbb{N} \wedge compt(c1) = 0 \wedge comp(c2) = 1)$

La variable *compt* associe à chaque composant un entier et *tic* associe à chaque composant une valeur dans $\{0, 1\}$. Un composant c accède à la valeur de sa variable *compt* en lisant $compt(c)$. L'événement qui fait incrémenter *compt* pourra avoir la forme ci-dessous (on suppose qu'il y a d'autres événements qui peuvent faire évoluer la valeur de *tic*).

Incr =
ANY c **WHERE**
$c \in COMP \wedge tic(c)=1$
THEN
$compt(c) := compt(c)+1 \parallel tic(c) := 0$
END

L'utilisation de cette forme avec ANY permet de dire que c peut représenter n'importe quel composant de *COMP* dont la variable *tic* vaut 1. Dans le cas où il y en aurait plusieurs, le choix est non-déterministe. Ainsi, l'événement *Incr* représente le comportement *possible* d'un ensemble de composants (tous ceux dont la variable *tic* vaut 1).

Cette forme d'événement en utilisant ANY est équivalente à l'écriture d'autant d'événements identiques qu'il y a d'éléments dans *COMP*. Supposons par exemple que l'ensemble de composants *COMP* contient seulement deux éléments $c1$ et $c2$; alors l'événement *Incr* est équivalent à l'ensemble des deux événements *Incr_c1* et *Incr_c2* ci-dessous. Ainsi le non-déterminisme de ANY modélise le non déterminisme dans le choix des événements.

Incr_c1 =
WHEN
$tic\_c1=1$
THEN
$compt\_c1 := compt\_c1+1 \parallel tic\_c1 := 0$
END

Incr_c2 =
WHEN
$tic\_c2=1$
THEN
$compt\_c2 := compt\_c2+1 \parallel tic\_c2 := 0$
END

Dans cet exemple il y a une variable *tic* pour chaque composant (on peut la considérer comme une entrée de chaque composant), c'est pourquoi elle est définie comme une fonction de domaine *COMP*). Cela est différent d'un variable *tic* qui serait globale et qui serait 'lue' (comme entrée) par tous les composants. Dans ce cas les composants n'auraient pas le droit de modifier cette variable.

Règle 2 (Composants identiques) *Cette règle relativise la règle 1.*
Lorsque le système comporte plusieurs composants identiques, on évite de répéter le code plusieurs fois en factorisant les états des composants par l'utilisation de fonctions associant à chaque composant son état et en écrivant un événement avec ANY *à la place de plusieurs événements identiques.*

En plus d'éviter de répéter plusieurs fois le même code, cette méthode permet de modéliser des systèmes avec un nombre indéterminé de composants identiques. En effet, le nombre d'éléments de l'ensemble abstrait *COMP* n'est pas nécessairement spécifié (on sait juste que, par définition, cet ensemble n'est pas vide et contient un nombre fini d'éléments). Si le besoin s'en fait sentir, on peut utiliser la clause PROPERTIES de B pour donner des propriétés supplémentaires à cet ensemble (fixer le cardinal par exemple). Cela dit, moins de propriétés sont ajoutées et plus le modèle est générique, par exemple si on ne fixe pas le cardinal de l'ensemble alors le modèle est valide quel que soit le nombre de composants ; pratique pour prouver des protocoles sans fixer le nombre d'objets communicants par exemple.

5.2 Communication entre les composants

Le point précédent impose que les états des composants soient indépendants les uns des autres de façon structurelle. Cependant, les composants ont besoin de communiquer entre eux. Si le système était composé de deux blocs totalement indépendants, il n'y aurait pas d'intérêt à les regrouper à l'intérieur d'un même système.

La modélisation de la communication change d'un niveau d'abstraction à un autre. A un niveau abstrait, des composants peuvent s'échanger des données complexes directement alors qu'à des niveaux

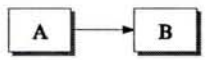
plus concret il sera peut être nécessaire d'implanter des protocoles pour que ces données puissent être échangées correctement. Par exemple au niveau abstrait deux composants peuvent être considérer comme communiquant directement ou même partageant un état commun alors que finalement ils communiqueront via un bus au niveau d'implantation. Au niveau d'implantation seules les communications correspondant à des fils reliant les composants entre eux sont modélisées. Pour modéliser des communications plus évoluées, il faudra introduire dans le modèle les composants implantant ces communications (contrôleurs de bus, routeurs, buffers, fifos, etc.).

5.2.1 Communication dans un seul sens

Au niveau de l'implantation, les seules communications concrètes existantes entre les composants passent par les fils qui relient les sorties aux entrées des composants. Comme nous l'avons vu dans le chapitre sur le principe de modélisation, les fils sont représentés par des variables. Au niveau d'implantation, les variables représentant les entrées et les sorties des composants doivent être explicitement modélisées.

Remarquons cependant que si nous introduisons d'une part des variables pour les sorties et d'autres part des variables pour les entrées, il faudrait alors ajouter des événements au modèle de sorte que les valeurs des variables de sortie soient systématiquement recopiées dans les variables d'entrée. Cette copie devant se faire, pour chacune des communications, après que la variable de sortie ait été mise à jour et avant que la variable d'entrée ne soit lue. Un tel mécanisme introduirait dans le modèle des événements complexes ne correspondant pas à proprement parler au modèle du système (on obtiendrait en fait un modèle de simulateur pour le système).

Une solution plus appropriée est d'utiliser des variables partagées entrées/sorties, c'est-à-dire des variables qui soient utilisées par les deux composants communicants, ainsi il n'est plus nécessaire de faire de la recopie de valeur. La variable porte la valeur qui est passée d'un composant A vers un composant B, elle joue donc à la fois le rôle de sortie pour le composant A et d'entrée pour le composant B.



Nous souhaitons cependant que les états des composants soient bien séparés. Il nous faut donc décider à quel composant cette variable partagée doit appartenir. Trois solutions peuvent apparaître :

1. A écrit directement sur l'entrée de B, l'entrée de B est partagée. Dans ce cas la variable partagée, appartenant au composant B est écrite par le composant A et lue par B.
2. B lit directement la sortie de A, la sortie de A est partagée. La variable partagée, appartenant au composant A est écrite par A et lue par B.
3. on utilise une variable supplémentaire v, A écrit sur v et B lit v.

La troisième solution revient à considérer le fil comme un composant intermédiaire C et la communication est alors un mixte entre les deux premières solutions : entre A et C la solution 1 est adoptée alors qu'entre C et B c'est la solution 2 qui est adoptée. Ce principe est un peu complexe et introduit les fils comme des composants, nous pensons donc qu'elle doit être mise de côté.

Par ailleurs, le choix entre les deux premières solutions revient à choisir si un composant doit (1) pouvoir écrire une variable qui ne lui appartient pas, ou (2) pouvoir lire une variable qui ne lui appartient pas. Notre principe de modélisation est que si une variable appartient à un composant alors ce sont les événements de ce composant qui doivent modéliser l'évolution de cette variable. Ainsi, il nous semble plus judicieux d'adopter la deuxième solution. En conséquence, une variable lue par un composant mais ne lui appartenant pas sera considérée comme une entrée du composant. Ce choix correspond au principe "une sortie est une variable écrite et une entrée est une variable lue" qui sera utilisé pour définir la sémantique du langage BHDL.

Ce choix est également compatible avec le principe de modélisation de l'environnement (cf. section 4.4) qui écrit les variables d'entrées du système, ces variables étant lues par le système. En posant que les variables correspondant aux entrées appartiennent à l'environnement, celui-ci peut être considéré comme un composant à part entière.

Règle 3 (Modélisation des communications) *Les communications entre les composants sont modélisées en utilisant des variables partagées. Une variable partagée est écrite par le composant initiateur de la communication (elle est une sortie pour ce composant) et lue par le composant destinataire (elle est une entrée pour ce composant)*

La règle 3 donne la façon de modéliser le fil de communication entre deux composants. Notre modèle événementiel du système étant dynamique, il reste encore à s'assurer qu'une variable partagée ne soit pas lue avant qu'elle n'ait été mise à jour. Ceci se fait en établissant un ordonnancement. Tout événement qui lit la variable ne doit pouvoir se produire qu'après les événements qui écrivent cette variable. Nous fixons pour le moment cette règle de la nécessité d'un ordonnancement, nous verrons plus loin (cf. section 5.5) un exemple de modélisation d'un tel ordonnancement.

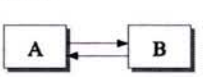
Règle 4 (Ordonnancement) *Lorsqu'il y a communication entre deux composants il doit y avoir un certain ordonnancement entre les événements de façon à ce que la valeur de la variable lue soit mise à jour avant qu'elle ne soit lue. Un ordonnancement bien formé ne doit pas provoquer le blocage infini d'un composant.*

A priori, l'ordonnancement en question n'est pas nécessairement total mais il doit assurer le déterminisme de la communication. S'il n'impose pas que la valeur de la variable lue soit mise à jour avant d'être utilisée on modélise un système non-déterministe qui peut aussi bien prendre la valeur non mise à jour ou la valeur mise à jour.

Il faut également remarquer que si on se trouve dans la situation où la valeur utilisée est toujours la valeur non mise à jour, cela signifie que la communication entre les deux composants n'a pas été bien pensée et doit être revue. Si la fonctionnalité souhaitée est celle-ci alors il faudrait sans doute penser à introduire un registre dans le composant auquel appartient la variable. En effet, si la traduction d'un composant vers un langage de description de circuit génère automatiquement les registres nécessaires, le système global doit être une architecture se contentant de relier les composants entre eux directement par des fils. Au besoin, si on ne souhaite pas introduire directement un registre dans le composant auquel appartient la variable, on peut aussi créer un composant supplémentaire ayant pour fonction de fournir l'ancienne valeur de la variable (ce composant représentant un simple registre).

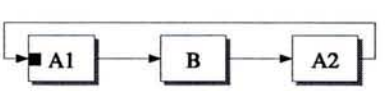
5.2.2 Communication dans les deux sens

Dans le cas où un composant A utilise les sorties d'un composant B, et que ce même composant B utilise les sorties du composant A, il est nécessaire de décomposer un des deux composants de façon à obtenir deux communications dans un seul sens.



Il faut d'abord remarquer que pour que cette communication à double sens soit bien formée, elle ne doit pas introduire de court-circuit. Un court-circuit est introduit lorsque pour calculer sa propre sortie, le composant A utilise directement (sans registre intermédiaire) la sortie de B, et que le composant B utilise la sortie de A pour calculer sa propre sortie. On peut donc faire l'hypothèse de bonne formation suivante : un des deux composants n'utilise pas directement la sortie de l'autre composant pour calculer sa propre sortie. Pour l'exemple, on supposera qu'il s'agit du composant A.

Le composant A peut alors se diviser en deux parties. D'une part la partie (notée A1) qui calcule la sortie (qui n'utilise donc pas la sortie de B), et le reste du composant, noté A2, (qui utilise la sortie de B). Dans le cas le plus général, A1 peut aussi utiliser la sortie du composant en passant pas l'intermédiaire d'un registre. C'est à dire qu'il n'utilise pas la sortie de B qui est calculée pendant le même cycle, mais celle calculée au cycle précédent. On peut donc utiliser l'ordonnancement représenté ci-dessous, le composant A1 étant chargé de donner une valeur initiale au registre qui prend la sortie du composant B.



A titre d'exemple des règles 3, 4 et 5, nous donnons ci-dessous un exemple de modèle. Il est constitué de deux composants et d'une communication dans les deux sens. D'une part il y a un additionneur 4-bits, et d'autre part un composant utilisant cet additionneur. Il y a communication dans les deux sens car l'utilisateur doit d'abord envoyer les données à additionner, puis il reçoit le résultat de l'additionneur. Tout se passe comme s'il y avait deux utilisateurs, un premier chargé d'envoyer les données et un deuxième chargé de récupérer de résultat. L'ordonnancement est modélisé par une variable *ord* qui est dans l'intervalle 0..2. Lorsqu'elle vaut 0 c'est la première partie de l'utilisateur qui est activée, lorsqu'elle vaut 1 c'est l'additionneur qui est activé, lorsqu'elle vaut 2 c'est la deuxième partie de l'utilisateur qui est activée.

On donne d'abord les deux événements modélisant l'additionneur 4-bits. L'additionneur teste s'il y a un dépassement de capacité (s'il y a une retenue sortante). Si c'est le cas il positionne la variable *erreur* à 1 et le résultat vaut $a + b \ mod \ 15$. Sinon le résultat vaut $a + b$.

adder_4bits_ok =
WHEN
$ord = 1 \wedge$
$a + b \leq 15$
THEN
$resultat := a + b \parallel$
$erreur := 0$
$ord := 2$
END

adder_4bits_ko =
WHEN
$ord = 1 \wedge$
$a + b > 15$
THEN
$resultat := a + b \ mod \ 15 \parallel$
$erreur := 1$
$ord := 2$
END

On donne ci-dessous les événements correspondant à l'utilisateur de l'additionneur. La fonctionnalité de cet utilisateur consiste à choisir une valeur de manière non déterministe pour a et de lui ajouter la valeur 1 (variable b). Si l'additionneur renvoie une erreur, c'est que la valeur de a était 15, sinon on sait que la valeur de a n'était pas 15.

user_1 =
WHEN
$ord = 0$
THEN
$a :\in 0..15 \parallel$
$b := 1 \parallel$
$ord := 1 \parallel$
$compteur := compteur + a\_est\_15$
END

user_2_ok =
WHEN
$ord = 2 \wedge$
$erreur = 0$
THEN
$a\_est\_15 := 0 \parallel$
$ord := 0$
END

user_2_ko =
WHEN
$ord = 2 \wedge$
$erreur = 1$
THEN
$a\_est\_15 := 1 \parallel$
$ord := 0$
END

La communication entre la première partie de l'utilisateur et l'additionneur se fait en utilisant les variables partagées a et b : elles ne sont écrites que par l'utilisateur et lues par l'additionneur. La communication entre l'additionneur et la deuxième partie de l'utilisateur se fait en utilisant les variables partagées *resultat* et *erreur*.

L'aspect cyclique est modélisé par le fait que la deuxième partie de l'utilisateur remet la variable *ord* à 0, ce qui provoque l'activation de la première partie de l'utilisateur. Nous avons une communication d'un cycle sur l'autre lorsque la première partie du compteur utilise la variable *a_est_15* qui n'est mise à jour que par la deuxième partie de l'utilisateur.

On donne ci-dessous l'initialisation des variables. La variable *ord* doit être initialisée à 0 pour s'assurer que c'est l'événement correspondant à la première partie de l'utilisateur qui se déclenche en premier. Les variables *resultat*, *erreur*, a et b n'ont pas besoin d'être initialisée de manière déterministe car elles sont toujours écrites avant d'être utilisées. La variable *a_est_15* doit être initialisée à 0 pour s'assurer que le compteur n'est pas incrémenté la première fois que l'événement user_1 se déclenche. Le compteur est lui-même initialisé à 0.

INITIALISATION
$a :\in 0..15 \| b :\in 0..15 \| resultat :\in 0..15 \| erreur :\in 0..1 \| ord := 0 \| a\_est\_1 := 0 \| compteur := 0$

5.2.3 Communications et cycle

Le principe de modélisation que nous avons choisi consiste à ce que l'ensemble des événements corresponde à la fonctionnalité du circuit pendant un cycle. Chaque événement correspond à un comportement possible (la garde de l'événement indique dans quels cas celui-ci s'applique) du circuit pendant un cycle. Les événements sont ordonnancés de façon à créer un comportement cyclique : une fois le cycle terminé (plus aucun événement ne peut se déclencher dans le cycle en cours), un nouveau cycle commence et les événements peuvent de nouveau (suivant leurs gardes) se déclencher. Leur nouveau déclenchement correspondant à l'évolution du système dans le nouveau cycle.

Les règles de communication définies plus haut imposent un ordonnancement supplémentaire (en plus de celui définissant le cycle) ayant pour fonction de fixer l'ordre dans lequel se font les lectures/écritures sur les variables partagées servant à la communication entre les composants. Nous avons donc deux ordonnancements nécessaires. Un premier pour assurer la succession correct des cycles, et un deuxième pour assurer la bonne formation des communications.

Il faut distinguer deux cas de communication en prenant en compte l'enchaînement des cycles. Soit la communication a lieu à l'intérieur d'un même cycle, soit la communication a lieu d'un cycle à l'autre. Le deuxième cas est celui dans lequel un composant utilise dans un cycle donné une valeur qui a été calculée (par un autre composant ou par lui-même) dans le cycle précédent.

Dans le cas d'une communication à l'intérieur d'un même cycle, l'ordonnancement nécessaire à cette communication se rajoute à l'ordonnancement nécessaire à la modélisation du cycle et n'interfère pas avec lui. Dans le cas d'une communication d'un cycle sur l'autre il faut faire attention à ce que les deux ordonnancements ne se bloquent pas l'un l'autre. D'une part, il faut noter que dans ce cas la communication se fait toujours dans le même sens : c'est l'événement du cycle suivant qui attend pour lire une variable partagée mise à jour par un événement du cycle courant (sinon cela reviendrait à lire une valeur dans le futur, on obtiendrait inévitablement un blocage du système). D'autre part, l'ordonnancement dû à la modélisation du cycle est lui même un ordonnancement correct pour la communication. En effet, lorsqu'un nouveau cycle commence, tous les événements du cycle précédent se sont produits (garanti par l'ordonnancement du cycle) et donc toutes les valeurs ont été mises à jour et peuvent être lues. Il n'est donc pas nécessaire d'ajouter un ordonnancement supplémentaire (à celui du cycle) comme dans le cas d'une communication dans un même cycle pour s'assurer que la lecture de la variable ait lieu après la mise à jour. Il faut cependant s'assurer que la valeur de la variable partagée soit utilisée avant qu'elle ne soit de nouveau mise à jour. Il doit donc exister un ordonnancement entre les événements qui utilisent la valeur (du cycle précédent) de la variable et ceux qui mettent à jour cette variable. Dans le cas d'une communication d'un cycle sur l'autre, l'ordonnancement doit donc être le contraire que dans le cas d'une communication dans un même cycle.

Règle 5 (Communication d'un cycle sur l'autre) *Lorsqu'une communication a lieu d'un cycle sur l'autre, il n'est pas nécessaire d'ajouter un ordonnancement supplémentaire pour assurer que la lecture s'effectue après la mise à jour. L'ordonnancement dû à la modélisation du cycle suffit pour cela. On ajoute cependant un ordonnancement pour assurer que la lecture s'effectue avant la nouvelle mise à jour.*

Le cas d'une communication sur plusieurs cycles (utilisation d'une valeur calculée plusieurs cycles avant), est une succession de communications d'un cycle sur l'autre. Il faut tout de même introduire de nouvelles variables pour stocker la valeur qui est transmise sur plusieurs cycles. Pour une communication sur n cycles, il faut introduire $n-1$ variables. Prenons l'exemple d'une communication sur deux cycles : une variable v est calculée au cycle n et on souhaite utiliser cette valeur au cycle $n+2$. Le cycle $n+1$ recalcule une nouvelle valeur pour v, l'ancienne valeur est perdue et ne peut donc pas être réutilisée au cycle $n+2$. On introduit une nouvelle variable, disons w, qui stocke l'ancienne valeur de v au cycle $n+1$. Ainsi, au cycle $n+2$, la valeur (avant nouvelle mise à jour), de w est la valeur de v calculée au cycle n.

Les variables rajoutées peuvent appartenir au composant fournissant la valeur, ou au composant utilisant la valeur, ou créer un composant dédié (buffer, mémoire ...) permettant de stocker des valeurs sur plusieurs cycles.

5.3 Non blocage des composants

Chaque composant est un circuit électronique qui fournit donc en permanence des sorties, dans le cadre du système global. Nous appliquons aux composants la même contrainte de modélisation que pour le système global : le modèle de chaque composant pris séparément ne doit pas pouvoir se bloquer.

Un modèle de composant qui se bloquerait serait un modèle dans lequel il serait possible qu'aucune des gardes des événements correspondant à ce composant ne puisse être évaluée à *vrai*. D'un point de vue de la modélisation en B événementiel cela signifierait que l'état du composant n'évoluerait plus, ce qui pourrait avoir un sens (à condition que le modèle du système global ne se bloque pas lui-même). Cependant, suivant le principe de modélisation des circuits électroniques que nous avons choisi, cela signifierait que l'évolution du composant ne serait pas spécifiée dans tous les cas. Or, un circuit électronique réagit nécessairement en fonction des signaux électriques qu'il reçoit en entrée et produit toujours des signaux de sorties. Ce comportement doit être modélisé explicitement (éventuellement en utilisant la substitution *skip* qui spécifie que l'état du système reste inchangé).

Cette contrainte de non blocage des composants doit être cependant relativisée en tenant compte de l'ordonnancement entre les composants dû aux communications. En effet, lorsqu'un composant utilise les sorties d'un autre composant, il est bloqué (par l'ordonnancement) en attendant que la sortie de l'autre composant soit mise à jour. Le fait que l'autre composant ne puisse pas se bloquer assure que le composant utilisant sa sortie pourra se débloquer. Il faut donc que l'ordonnancement dû aux communications assure le déblocage des composants en attente lorsque les sorties sont disponibles. La contrainte de non blocage des composants est donc divisée en deux parties : d'une part la bonne formation de l'ordonnancement, et d'autre part le non blocage du composant dans lequel les gardes dues à l'ordonnancement sont retirées.

Règle 6 (Non blocage des composants) *Chaque composant du système ne doit pas pouvoir se bloquer. On doit assurer que l'ensemble des événements correspondant à chaque composant constitue un système toujours en vie : la disjonction de l'ensemble des gardes d'un composant donné doit toujours être vraie).*

Les obligations de preuve provenant de l'application de la méthode B sur le système assurent que le système entier ne se bloque pas mais il faut assurer aussi que chacun des composants ne se bloque pas. Donc, en plus des contraintes liées à la méthode B, on se contraint en plus à ce que le comportement de tout composant soit défini pour tous les cas possibles. Cette contrainte supplémentaire peut être vue comme une contrainte de séparation de systèmes. Chaque composant peut être vu comme un système en lui-même (avec le reste du système comme environnement), et ne doit donc pas pouvoir se bloquer. Les outils existants actuellement ne proposent pas le moyen de séparer les composants dans un système B. Il n'est donc pas possible pour le moment de générer automatiquement les obligations de preuve pour assurer cette propriété.

Cela dit, l'étape de *regroupement* (cf. section 5.6.2) nécessite que l'ensemble des états ait été correctement partitionné (particulièrement, tous les cas doivent avoir été pris en compte). Ainsi, si cette contrainte n'est pas respectée, cette étape nécessaire pour la traduction ne pourra pas être réalisée. Il n'est donc pas nécessaire de produire des obligations de preuve spécifiques au non blocage des composants, l'application du regroupement assure le non blocage des composants.

Dans l'exemple donné précédemment avec un additionneur et un utilisateur, cette règle n'est pas respectée. Si on fait par exemple la disjonction des gardes des événements de l'additionneur on trouve que la disjonction est équivalente à $ord = 1$. Pour respecter la règle 6 on peut ajouter un événement *adder_nothing* qui fait la substitution *skip* dans tous les cas non pris en compte. De la même façon il faut ajouter l'événement *user_nothing*. Comme ces événements ne modifient pas l'état du système, il n'ont pas d'influence sur le déroulement des autres événements.

adder_nothing=
WHEN
$ord \in \{0, 2\}$
THEN
skip
END

user_nothing =
WHEN
$ord = 1$
THEN
skip
END

On pourrait dire que ces événements permettent au système de ne jamais rien faire puisque cela

revient à avoir un événement $nothing = skip$ qui peut donc se produire en boucle indéfiniment. C'est *un* comportement possible, mais les preuves en B sont faites sur *tous* les comportements possibles, donc y compris ceux où ces événements ne se produisent pas en boucle infinie. Les preuves faites sur le système ont donc bien du sens.

5.4 Parallélisme, concurrence et non déterminisme

Dans un système électronique, les composants "fonctionnent" en même temps. En fait les signaux électriques parcourent l'ensemble du système électronique en permanence. On fait souvent référence au "parallélisme" des circuits électroniques : mettre deux composants en parallèle signifie qu'ils ne communiquent pas entre eux, chacun reçoit des signaux d'entrée (qui peuvent éventuellement être en partie communs) et produit des signaux de sortie. Les signaux d'entrée et de sortie du système résultant de la mise en parallèle de ces deux composants est la réunion des signaux d'entrée et de sortie des deux composants. Le temps de latence du système (temps nécessaire entre le changement des signaux d'entrée et la stabilisation des signaux de sortie) est égal au plus grand temps de latence des deux circuits, contrairement au temps de latence d'un système ou deux circuits sont mis en séquence qui est égal à la somme des deux temps de latence. Produire une architecture la plus parallèle possible permet donc de fournir un système avec le temps de latence le plus petit, et donc une fréquence d'horloge d'autant plus élevée.

L'opérateur $\|$ (composition parallèle de deux substitutions) en B correspond à cette notion de parallélisme : la sémantique stipule que dans une substitution $A\|B$, A et B utilisent les valeurs *avant* des variables, c'est-à-dire que si A modifie une variable x (pour lui donner une nouvelle valeur x') et que B utilise cette variable, la valeur utilisée par B est la valeur *avant* x_0 et non pas la valeur x' assignée par A. Cependant, dans notre méthodologie, les substitutions sont utilisées pour modéliser l'intérieur d'un composant, les composants étant modélisés par des ensembles d'événements. On utilisera donc l'opérateur $\|$ pour modéliser le parallélisme à l'intérieur d'un composant, mais pour modéliser le parallélisme entre les composants il nous faut le modéliser autrement. Deux composants évoluant en même temps (en parallèle) signifierait que deux événements peuvent se déclencher en même temps. Or, le modèle événementiel que nous utilisons est séquentiel, c'est-à-dire que les événements ne peuvent se déclencher que les uns après les autres, deux événements ne peuvent jamais se déclencher au même moment. Un modèle très répandu pour modéliser le parallélisme dans un système séquentiel est la notion de concurrence. Deux événements sont concurrents lorsque leurs gardes peuvent être toutes les deux évaluées à *vrai* à un même moment. Cela dit, le modèle séquentiel impose que seul l'un d'eux pourra se déclencher (éventuellement, l'autre se déclenchera par la suite).

Dans le cadre d'un langage computationnel, l'un des deux événements doit être choisi (selon des critères précis, ou choisi "au hasard") au dépend de l'autre événement. C'est, par exemple, le modèle choisi pour le simulateur SystemC fournit par l'OSCI. Le résultat du calcul est alors a priori dépendant des choix qui ont été faits entre les événements concurrents. Des choix différents peuvent éventuellement conduire à des résultats différents. Par exemple dans le cas de SystemC, il est très facile d'écrire des modèles dépendant des choix faits par le simulateur. Même si le standard impose qu'un simulateur donné donne toujours les mêmes résultats à chaque simulation, les résultats peuvent cependant différés d'un simulateur à l'autre. Pourtant, si cette concurrence est censée modéliser du parallélisme, le résultat devrait être le même, quels que soient les choix, puisque deux composants parallèles ne s'influencent pas. Dans ce cadre, la validation du modèle ne peut donc pas se suffire d'une exécution du modèle sur l'ensemble des combinaisons possibles des séquences d'entrée (à supposer que cela soit déjà possible en pratique), mais il faut également essayer toutes les combinaisons possibles de choix faits dans les choix des événements concurrents.

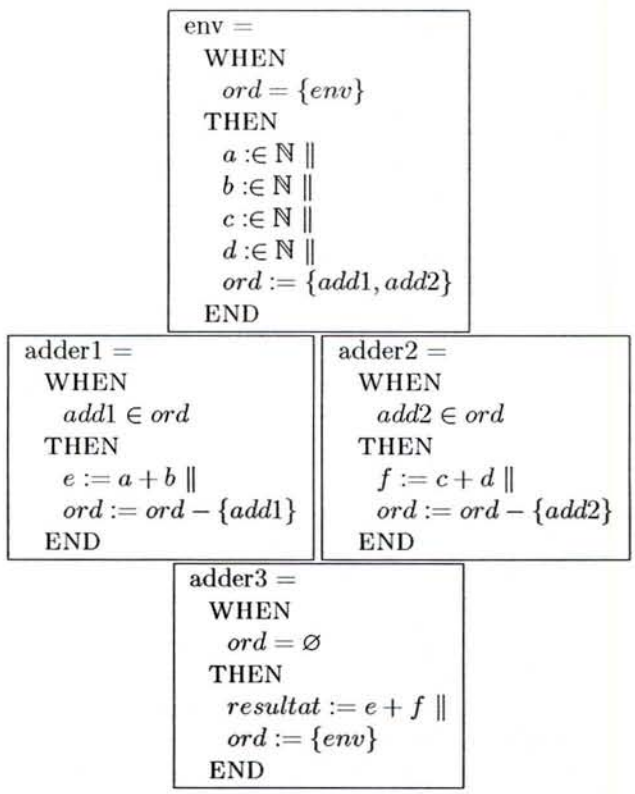
Dans le cadre de la méthode B, nous ne cherchons pas à obtenir des modèles qui soient directement exécutables. Au contraire, nous allons tirer parti de la possibilité que nous offre le langage d'utiliser le non déterminisme. Dans le cadre de la modélisation de la concurrence, nous tirons parti du non déterminisme qui existe dans le choix de l'événement qui peut se déclencher. Lorsque les gardes de deux événements peuvent s'évaluer à *vrai*, le choix duquel se déclenchera est non déterministe, c'est-à-dire que la méthode B ne fait aucune hypothèse sur le choix qui peut être fait. En conséquence de quoi, les propriétés qui sont prouvées sur le système sont valables quel que soit l'ordre de déclenchement des événements. Ainsi, si un modèle donné est prouvé correct par rapport à sa spécification, alors nous sommes

assurés que la spécification sera respectée par n'importe quel modèle de simulation du système. Grâce au non déterminisme de la méthode nous pouvons donc prouver le déterminisme (conduit systématiquement au même résultat) de la concurrence, qui est une condition nécessaire pour modéliser correctement le parallélisme.

Règle 7 (Modélisation de la concurrence) *Le parallélisme est modélisé par une concurrence entre les événements. La concurrence est modélisée en utilisant le non déterminisme dans le choix des événements. Deux composants parallèles sont modélisés par des événements dont les gardes peuvent être simultanément évaluées à vrai.*

Dans le cas du parallélisme, c'est-à-dire lorsque les composants ne communiquent pas, ce non déterminisme ne complique pas les preuves de correction puisque chaque composant est indépendant de l'autre, donc l'ordre dans lequel les événements sont choisis n'a aucune importance. Dans le cas où deux composants communiquent, ils ne sont plus parallèles (éventuellement une décomposition peut isoler des parties qui peuvent se paralléliser). Dans ce cas l'ordonnancement nécessaire à la modélisation de la communication (règle 4) impose un ordre entre les événements. Il n'y a donc plus de concurrence et la preuve n'est donc pas non plus compliquée dans ce cas.

A titre d'exemple nous donnons les événements pour un additionneur de 4 nombres entiers. Le calcul se fait en faisant en parallèle deux additions de deux nombres et en additionnant les résultats. Nous montrons ci-dessous une manière de synchroniser les événements effectuant les additions en parallèle.



Lorsque l'événement *env* se produit, à la fois *adder1* et *adder2* peuvent se déclencher. Par ailleurs le déclenchement de l'un n'empêche pas le déclenchement de l'autre par la suite. L'événement *adder3* attend que les deux événements *adder1* et *adder2* se produisent pour pouvoir se déclencher.

5.5 Ordonnancement

Nous avons expliqué précédemment que les communications entre les composants nécessitent de se préoccuper de l'ordre dans lequel les événements du modèle se déclenchent. Nous avons également vu qu'il

nous fallait modéliser l'aspect cyclique du système modélisé. Ces deux points sont modélisés en utilisant un ordonnancement entre les événements. Nous définissons un cycle comme une succession de *périodes*. L'aspect cyclique se retrouve dans le fait qu'à la dernière période d'un cycle succède la première période du cycle suivant.

5.5.1 Périodes

Dans le système modélisé, il existe une relation de dépendance entre les différents calculs. Par exemple si on souhaite additionner trois nombres, il faut d'abord additionner les deux premiers nombres puis additionner le résultat au troisième nombre. La deuxième addition est donc dépendante de la première. De façon plus générale, nous l'avons vu en ce qui concerne les communications entre les composants, il existe une relation de dépendance entre les différentes parties des différents composants du système.

Dans un circuit physique, cette dépendance se traduit par la propagation des signaux sur le *chemin des données*. Quand un composant A dépend d'un composant B, la sortie de B est connectée à l'entrée de A. Le signal est d'abord transformé par B avant d'arriver au composant A.

Le système est modélisé par un ensemble d'événements. Chacun des événements peut se déclencher à chaque fois que sa garde est évaluée à vrai. Il n'y a donc pas a priori de notion d'ordre entre les événements. On a vu que dans le cas d'un parallélisme (sans communication) entre les composants du circuit, on utilise le non-déterminisme dans le déclenchement des événements pour modéliser la concurrence. Dans le cas où il y a communication entre les composants, il doit exister un ordonnancement pour assurer que dans toute communication l'ordre des lectures et mises à jour des variables soit bien déterminé (par le développeur).

Nous proposons de modéliser cet ordonnancement en divisant le système en périodes successives. Le principe consiste à interdire les communications à l'intérieur d'une même période. Les communications utilisent des variables partagées, il s'agit donc de s'interdire de lire et modifier une même variable dans une même période.

Par ailleurs, on s'impose que le comportement de chaque composant soit modélisé dans chaque période, y compris si un composant n'a pas de calcul à faire pendant cette période. Ceci permet de maîtriser l'évolution du système en s'assurant que tous les composants se trouvent dans une même période à un instant donné. Cette contrainte facilite également la modélisation puisqu'il n'est ainsi pas nécessaire de distinguer les composants suivant qu'ils aient ou non des calculs à faire dans une période donnée.

Chaque période correspond à du parallélisme, modélisé comme cela a été vu précédemment. Il reste à modéliser le passage d'une période à l'autre.

5.5.2 Modélisation

Il y a de multiples manières de modéliser le cycle divisé en plusieurs périodes. La modélisation que nous avons choisie est systématique et une génération automatique peut être envisagée.

On se dote de l'ensemble abstrait des composants du système, disons ALL. On ajoute à l'état du système autant de variables supplémentaires qu'il y a de périodes, $E_1, ..., E_n$. Les nouvelles variables E_i sont des sous-ensembles de ALL. Un composant fait toujours partie d'un et un seul des E_i, et cela signifie que le composant est dans la période numérotée i. Bien évidemment, une bonne modélisation renommera chaque variable E_i par un nom caractérisant la période.

Chaque période est une collection d'événements. Aucun de ces événements ne doit pouvoir se déclencher avant que la période précédente ne soit complètement terminée. Pour réaliser cela, on impose une contrainte sur la forme des événements. Ci-dessous on montre la forme d'un événement pour la période numérotée i.

event_i =
ANY c **WHERE**
$c \in E_i \wedge$
$E_{i-1} = \varnothing \wedge$
...
THEN
... ||
$E_i := E_i - \{c\}$ ||
$E_{i+1} := E_{i+1} \cup \{c\}$
END

Les indices devront évidemment être adaptés dans les cas où i vaut 1 ou n. Cet événement ne peut se déclencher que lorsque la période précédente est terminée ($E_{i-1} = \varnothing$) et que le composant se trouve dans cette période ($c \in E_i$). On peut montrer que dans ce cas, tous les autres composants sont soit dans la même période soit prêt à effectuer la période suivante. On remarquera qu'un seul événement par composant peut se déclencher dans chaque période.

On s'imposera également de modéliser le comportement de chaque composant dans chaque période. C'est-à-dire que pour chaque période, les gardes des événements de cette période doivent couvrir tous les cas possibles. On pourrait assouplir cette contrainte puisque tous les composants n'ont pas quelque chose à faire dans chaque période. Mais cela rendrait la modélisation de l'ordonnancement dépendante du système et non systématique. Les composants qui n'ont rien à faire devront donc avoir tout de même un événement modélisant l'ordonnancement. De plus, cela simplifie considérablement l'étape de regroupement (section 5.6.2).

La modélisation de l'ordonnancement telle qu'elle est faite ici garantit que le comportement de chaque composant est modélisé de façon explicite dans chaque période. Si aucun événement ne pouvait se déclencher pour un événement dans une période i, la variable E_i ne pourrait jamais revenir à $\varnothing$ et on ne pourrait pas passer à la période suivante. Le système serait bloqué.

Les variables d'ordonnancement sont initialisées à l'ensemble vide, sauf la variable représentant la période par laquelle on souhaite commencer qui est initialisée à ALL.

5.5.3 Environnement

En général, on consacrera au moins une période particulière pour modéliser l'environnement.

Lorsqu'on utilise un circuit, les entrées ne doivent pas être modifiées au moment du coup d'horloge. Cela dit, ce qui se passe entre le coup d'horloge et le changement des entrées n'a pas besoin d'être modélisé. D'un point de vue modélisation, il faut donc que les entrées soient fixées au début du cycle du système, avant que les événements modélisant les composants se déclenchent.

Une bonne modélisation ne fera donc pas évoluer l'environnement et le système en même temps. Pendant la période consacrée à l'environnement, seul celui-ci évolue, aucun des autres composants ne modifie l'état du système. On s'autorisera donc une simplification dans la modélisation de l'ordonnancement, en prenant le modèle suivant pour l'environnement.

env =
WHEN
$E_{env} = ALL \wedge E_{env-1} = \varnothing$
THEN
... ||
$E_{env} := \varnothing$ ||
$E_{env+1} := ALL$
END

Cette simplification n'est pas gênante pour le regroupement puisqu'elle suit un schéma simple et que l'environnement est une partie du système qui n'est pas traduite.

En général, la période par laquelle le système commence à l'initialisation est la période de l'environnement.

5.5.4 Invariants de période

Chaque période représente une partie de l'évolution du système. Il est utile d'exprimer des propriétés sur chaque période. On souhaite spécifier des invariants de période, c'est à dire des propriétés conservées par tous les événements observables d'une période. Un système B ne permet d'exprimer des invariants que sur l'ensemble du système.

Pour pouvoir exprimer des invariants de période, il faut tenir compte explicitement de l'enchaînement des périodes dans l'énoncé de la propriété.

Par exemple, pour dire qu'une période E_i établit une propriété $P(c)$ sur l'état d'un composant c, on dira que le composant possède la propriété dans la période suivante : $c \in E_{i+1} \Rightarrow P(c)$.

Les choses se compliquent lorsque la propriété concerne plus d'un composant. En effet, les composants passent d'une période à la suivante un par un. Selon les cas, il pourra être nécessaire d'exprimer toutes les combinaisons.

Prenons un ensemble de composants possédant tous une variable compteur *count*. Le système possède une période *incr* qui incrémente le compteur de chaque composant. On veut exprimer que les variables *count* des différents composants ont toujours des valeurs identiques. Il est clair que la valeur de *count* pour un composant qui a passé la période *incr* est exactement supérieur d'une unité par rapport à la même variable d'un composant qui n'a pas encore passé cette période.

On écrira par exemple
$$\begin{cases} \forall(c,d).(c \in E_{incr} \wedge d \in E_{incr} \Rightarrow count(c) = count(d)) \wedge \\ \forall(c,d).(c \in E_{incr} \wedge d \in E_{incr+1} \Rightarrow count(d) = count(c) + 1) \wedge \\ \forall(c,d).(c \in E_{incr+1} \wedge d \in E_{incr+1} \Rightarrow count(c) = count(d)) \end{cases}$$

Lorsqu'on souhaite exprimer un invariant de période, il faut donc penser à prendre en compte de quelle façon la propriété évolue d'une période à l'autre et ne pas oublier que les composants passent d'un période à la suivante un par un. Pour prouver ces invariants, il peut être nécessaire de prouver des propriétés sur les autres périodes. En particulier, pour qu'une période respecte un invariant, il faudra entre autre prouver que l'invariant est vrai en début de période, c'est-à-dire que la période précédente établit l'invariant.

Il est difficile de penser précisément à tous les invariants qui seront nécessaires pour prouver la cohérence du système et la correction d'un raffinement. Le fait que toutes les obligations de preuve soient générées automatiquement par des outils permet de s'assurer qu'on a bien exprimé tous les invariants nécessaires. Lorsqu'un invariant manque, on s'en aperçoit assez vite lorsqu'on essaie de faire les preuves.

5.6 Regroupement, implantation de l'ordonnancement

Une fois le modèle implantable obtenu, il faut mettre le code lui-même en forme pour le traduire. Le traducteur ne travaille qu'à partir d'une forme dite *regroupée*.

5.6.1 Séparation des composants

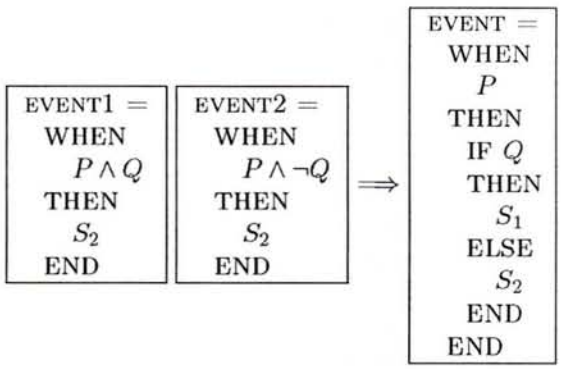
Le système modélise plusieurs composants. Chaque composant est modélisé par un ensemble de variables et une collection d'événements. On souhaite traduire les composants séparément. Les événements d'un composant sont traités de façon séparée des événements des autres composants.

Le seul cas problématique est le cas où le système comporte plusieurs composants identiques (section 5.1.1). L'état de ces composants est modélisé en utilisant des fonctions et les événements ne correspondent pas à un seul composant. Remarquons cependant que dans ce cas, on ne souhaite traduire le code qu'une seule fois. Pour cela, on transforme le code en transformant les variables $v \in COMP \rightarrow Type$ en variables simples $v \in Type$. Dans le code, on transforme les expressions $v(c)$ en v. Les événements de la forme **ANY** c **WHERE** ... **THEN** ... **END** sont transformés en **SELECT** ... **THEN** ... **END**.

5.6.2 Regroupement simple

Le regroupement du système consiste à prendre tous les événements et à les composer entre eux de façon à obtenir un code en un seul bloc. Ceci se fait itérativement en utilisant la règle de réécriture

ci-dessous [2].



On effectue ce regroupement itérativement pour chaque période et chaque composant. Notez que ce regroupement est faisable complètement pour chaque période et chaque composant puisque on s'était imposé au départ que pour chaque période, tous les cas sont explicitement modélisés (y compris quand un composant ne fait rien). Le cas échéant, il peut être nécessaire de raffiner les gardes des événements en gardes équivalentes mais syntaxiquement bien formées pour permettre l'application de la règle de réécriture du regroupement. La règle de [2] générant une substitution WHILE n'est pas utilisée ici car dans un circuit les boucles ne sont pas exprimées explicitement, elles sont le fait du comportement cyclique du circuit.

5.6.3 Implantation de l'ordonnancement

Après ce premier regroupement, on obtient un événement pour chaque période et chaque composant. C'est-à-dire que le comportement d'un composant c dans la période i est modélisé par un seul événement $event\_c_i$. Il faut maintenant composer ces événements entre eux en tenant compte de l'ordonnancement. L'ordonnancement utilisé ici exprime la séquentialité, il y a un ordre total entre les événements.

Le moyen le plus naturel pour composer les périodes est d'écrire le code ci-dessous, où $sl\_event\_c_i$ est la substitution définie par le regroupement des événements du composant c de la période i dans laquelle toutes les références aux variables d'ordonnancement ont été enlevées. Le point-virgule ; est une substitution séquentielle : appliquer la substitution $A; B$ signifie appliquer d'abord la substitution A puis la substitution B.

```
circuit_c =
        BEGIN
          sl_event_c1 ; ... ; sl_event_cn
        END
```

On peut regrouper l'ordonnancement d'autres façons, ce qui compte est de respecter l'aspect séquentiel de l'ordonnancement. Par exemple, pour un système en deux périodes (plus la période de l'environnement qui n'est pas traduite), on peut attribuer un cycle d'horloge à chaque période en ajoutant le moyen de basculer d'une période à l'autre à chaque cycle d'horloge. En utilisant une variable booléenne sc initialisée à *false*, on peut écrire le code ci-dessous.

```
circuit_c =
        BEGIN
          IF sc = FALSE THEN
            sl_event_c1 ||
            sc := TRUE
          ELSE
            sl_event_c2 ||
            sc := FALSE
          END
        END
```

Il peut être possible d'avoir un ordonnancement qui n'exprime pas un ordre total entre tous les événements. Ceci permet par exemple de modéliser une concurrence entre deux blocs de code d'un circuit. Si on peut montrer que l'ordre n'a pas d'importance, on peut envisager d'implanter l'ordonnancement par la substitution parallèle (à condition que les deux blocs de code utilisent des variables bien séparées). Nous n'abordons pas cet aspect ici car si on peut montrer que l'ordre n'a pas d'importance, on peut en imposer un afin d'obtenir un ordre total et retomber dans le cas décrit précédemment.

5.6.4 Génération de code

Le code obtenu après regroupement de l'ordonnancement est exprimé en BHDL, langage qui est syntaxiquement un sous-ensemble de B mais avec une sémantique de circuit. Le traducteur BHDL® [49] prend une description dans ce langage pour produire du code dans un langage de description de circuit comme VHDL ou SystemC. Le code produit est synthétisable [92].

On remarquera qu'il n'y a eu dans la modélisation aucune distinction entre les variables qui représentent des fils et celle qui représentent des registres. Cette distinction est effectuée automatiquement par le traducteur.

De la même façon, l'horloge n'a pas été modélisée explicitement dans le modèle et est ajoutée par le traducteur, ainsi que le signal *reset* qui réinitialise le composant.

5.7 Conclusion

Dans ce chapitre, nous avons vu comment modéliser les composants de façon indépendante, et en particulier comment faire pour modéliser, sans faire augmenter la taille du modèle B, le cas d'un système pouvant contenir plusieurs composants identiques. Nous avons également étudié la façon dont les communications peuvent se faire entre les différents composants. Nous avons montré comment modéliser le système de façon à permettre ces communications en ordonnançant les événements du modèle B. Cet ordonnancement permet également de modéliser l'aspect cyclique du système. Enfin nous avons vu comment regrouper les événements et implanter l'ordonnancement de façon à obtenir le code BHDL des composants du système. Les règles de regroupement ont été introduites par Abrial (cf. [2, 6]). A partir de ce code BHDL on peut effectuer des traductions automatiques vers d'autres formalismes. Les règles citées dans ce chapitre proviennent des constatations effectuées pendant la modélisation des études de cas, en particulier l'étude du protocole SAE (cf. chapitre 7).

Chapitre 6

Exemple de développement de circuit par la méthode B

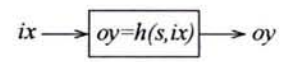
Les modèles présentés ici ont été développés avec Stefan Hallerstede et ont donné lieu à une publication [50]. Le but était de comparer les résultats obtenus par l'application de la méthode B avec ceux obtenus sur la même étude de cas en utilisant le logiciel Alpha [35]. La spécification initiale du circuit est initialement donnée par une équation mathématique. Cette spécification est formellement raffinée jusqu'à obtenir un modèle B synthétisable en circuit. Le circuit obtenu est donc correct par construction par rapport à la spécification initiale.

Nous présentons cet exemple ici pour illustrer le langage B dans un exemple non trivial et montrer comment il peut s'appliquer au développement d'un circuit électronique. Il permet également d'introduire le sous langage BHDL. La suite de ce document entre plus en détail sur le langage BHDL et sa traduction vers des formalismes de description de circuits.

6.1 Circuits synchrones en B événementiel

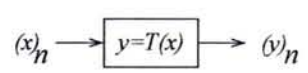
La cible que nous visons comme résultat du développement est un circuit synchrone. Un circuit synchrone évolue dans le temps par unités de cycles d'horloge. Durant chaque cycle, le circuit lit les valeurs ix présentées à ses entrées, produit les valeurs oy en sortie, et modifie son état s qui est invisible à son environnement (cf. figure 6.1).

FIG. 6.1 – Schéma d'un circuit synchrone



Une façon de spécifier un tel système est de donner une séquence de valeurs d'entrée, $(x)_n$, et une séquence des valeurs de sortie, $(y)_n$, et une relation $y = T(x)$ entre ces deux séquences (cf. figure 6.2) comme cela est usuel en traitement du signal [71]. L'entité $T(\cdot)$ est traditionnellement appelée une transformation.

FIG. 6.2 – Spécification d'un système discret



Il pourrait être contre-productif de penser à n comme un indice de temps bien que ce puisse être le cas dans un certains nombres de modèles. Dans le domaine du traitement du signal l'indice n est généralement désigné comme le temps, mais, pour ajouter à la confusion, ce temps peut tout aussi bien représenter

une position spatiale d'un pixel dans une image, et une définition de $y(n)$ en fonction de $x(n+k)$, avec $k \geq 0$, pourrait avoir un sens. Il n'y a, cependant, a priori aucune garantie qu'il existe une implantation physique d'un tel système. Pour éviter toute confusion dans la suite de cette section, nous désignerons n par le terme "position dans la séquence d'entrée" (resp. de sortie).

En partant de l'égalité $y = T(x)$, on peut construire un circuit synchrone, tel que cela est montré sur la figure 6.1, pour lequel ix et oy satisfont $y = T(x)$, en considérant que la séquence des valeurs de ix à chaque cycle correspond à la séquence x (resp. pour oy et y). L'utilisation de séquences pour spécifier des composants électroniques est un modèle standard dans le domaine du traitement discret du signal (cf. par exemple [44, 71]). Nous expliquons dans la suite comment ceci peut se faire par raffinement formel.

Comme dans [4] les restrictions sur l'utilisation des variables sont seulement appliquées au niveau de l'implantation mais pas nécessairement plus tôt dans le développement : la partie du système correspondant au circuit doit ne contenir que des variables dont les types sont implantables. Cependant, la spécification initiale modélise les effets du circuit sur son environnement. Dans ce premier modèle, il n'y a pas de variable qui corresponde au circuit lui-même. Le circuit est spécifié comme l'évolution de l'environnement modifiant une variable f, de telle façon que f soit un préfixe de la séquence y, la séquence f est la séquence y en construction. Le circuit au niveau d'implantation ne doit pas se référer aux variables n, f, x ou y car ils font partie de l'environnement. Pendant le raffinement, par contre, ces variables peuvent apparaître dans les événements qui modélisent le circuit. La distinction entre le circuit et l'environnement n'est pas faite initialement. Le long du processus de raffinement nous introduisons des jetons de contrôle, certains d'entre eux sont attachés au circuit et les autres à l'environnement[2]. En fin de raffinement, les événements correspondant au même circuit peuvent se combiner ensemble pour ne former qu'un seul événement par circuit, comme décrit dans [2, 70].

Une fois obtenu un événement modélisant un circuit, celui-ci peut se traduire dans un langage de description de circuit traditionnel [48]. L'approche décrite ici peut se voir comme une généralisation de [4] car les jetons de contrôle peuvent être raffinés et autorisent la modélisation de systèmes contenant plus d'un composant. Dans [4], le contrôle est passé entre l'environnement et le circuit en distinguant deux "modes" qui se succèdent alternativement. L'utilisation de jetons de contrôle permet également d'exprimer de façon simple l'ordonnancement entre plusieurs composants, comme la concurrence ou la mise en série par exemple.

6.2 Spécification du système linaire

Le domaine du traitement du signal discret concerne le traitement des signaux discrets qui sont observés à des instants discrets. Un signal est une quantité qui porte de l'information sur l'état et le comportement d'un système physique. Les signaux dans le domaine du traitement du signal discret sont généralement modélisés par des séquences de nombres [71]. Dans la suite nous ferons l'hypothèse que les nombres sont des entiers dans un intervalle $0 \,..\, MX$ où $MX > 0$ (intervalle qui peut tout aussi bien représenter des nombres non entiers mais avec un format de virgule fixe). Une séquence peut donc être représentée comme une fonction $x \in \mathbb{Z} \rightarrow 0 \,..\, MX$. Comme nous devons également modéliser l'initialisation du système, correspondant au temps 0, on supposera que le signal d'entrée x n'est pas disponible avant l'instant 1. Ainsi, la séquence d'entrée du modèle sera modélisée par la fonction $x \in \mathbb{N}_1 \rightarrow 0 \,..\, MX$. La séquence correspondant au signal de sortie est modélisée par la fonction $y \in \mathbb{N} \rightarrow 0 \,..\, MX$. L'indice 0 de la séquence de sortie y est utilisé pour "initialiser" l'élément de récurrence de l'équation du système, il ne s'agit pas d'une sortie du système. Le premier signal de sortie est donc disponible à l'instant 1.

Les systèmes de traitement du signal sont classifiés par rapport aux propriétés de la transformation $T(\,\cdot\,)$. Un système est dit *linéaire* si $T(s*x+t*y) = s*T(x)+t*T(y)$ pour tous scalaires s et t, et toutes séquences x et y. La séquence $s*x$ est la séquence où tous les éléments de la séquence x sont multipliés par le scalaire s et l'addition de deux séquences peut se définir récursivement comme la séquence commençant par l'addition des premiers éléments de chaque séquence suivi de l'addition du reste des séquences.

[2]Cette approche utilisant des jetons permet d'avoir plus d'un circuit dans le système.

Spécification abstraite du système

Le prédicat suivant spécifie la fonctionnalité attendue du système en reliant les séquences x et y. La première équation spécifie que les N premières sorties sur y doivent être 0. La deuxième équation est détaillée ci-dessous. Les éléments a et b sont des poids qui servent à pondérer les éléments de l'équation : a sert à pondérer l'influence de la sortie précédente dans la nouvelle sortie et b sert à pondérer les N dernières entrées. La constante N est déclarée comme étant un entier strictement supérieur à 1.

$$(0\,..\,N{-}1) \lhd y = (0\,..\,N{-}1) \times \{0\} \wedge$$
$$\forall n \cdot (n \geq N \Rightarrow y(n) = \sum\!\!\!\!\!\circ\, j \cdot (j \in 0\,..\,N{-}1 \mid x(n{-}j) \otimes b(j)) \ominus a \otimes y(n{-}1))$$

L'addition $\oplus$, la soustraction $\ominus$, et la multiplication $\otimes$ sont définies modulo $MX{+}1$ de façon à ce que les résultats soient toujours dans l'intervalle $0..MX$, par exemple $v \oplus w = v + w \bmod (MX{+}1)$; et $\sum\!\!\!\!\!\circ\, j \cdot (j \in \{m\} \mid e) = (e[j := m])$ et pour $n > m$: $\sum\!\!\!\!\!\circ\, j \cdot (j \in m..n \mid e) = \sum\!\!\!\!\!\circ\, j \cdot (j \in m..n{-}1 \mid e) \oplus (e[j := n])$ (cf. [26] pour une définition de l'induction basée sur la théorie des ensembles). Ces opérateurs n'existent pas en B mais peuvent être modélisés facilement par des fonctions constantes. Par souci de commodité, nous les notons $\oplus$, $\ominus$, $\otimes$ et $\sum\!\!\!\!\!\circ$ dans les modèles. Les premières sorties du système est une séquence de $N{-}1$ zéros, où $N \in \mathbb{N}$ est une constante telle que $N > 1$.

La fonctionnalité du système est spécifiée par une équation récurrente $y(n) = conv(n) - a \otimes y(n{-}1)$ qui est la différence entre la somme de convolution (de taille N)

$$conv(n) \mathrel{\widehat{=}} \sum\!\!\!\!\!\circ\, j \cdot (j \in 0\,..\,N{-}1 \mid x(n{-}j) \otimes b(j))$$

et le produit $a \otimes y(n{-}1)$ contenant un élément de récurrence. Les poids a et $b(j)$ sont spécifiés ainsi :

$$a \in 0\,..\,MX \wedge$$
$$b \in (0\,..\,N{-}1) \rightarrow (0\,..\,MX)\ .$$

Nous nous focalisons sur la manière dont une implantation par des composants matériels simples peut être atteinte en partant de l'équation récurrente ci-dessus. Nous ne nous préoccupons pas ici des problèmes de stabilité ou de calcul de points fixes [71]. Pour raffiner le système, seules les propriétés algébriques des nombres sont nécessaires. Nous visons une implantation du système qui lise les valeurs de la séquence d'entrée de façon séquentielle. Les sorties sont également écrites de manière séquentielle dans une séquence f telle que $f(i) = y(i)$ pour tout i tel que $0 \leq i \leq n$, où n est la position courante dans la séquence de sortie (rappelons que y est la séquence *entière* des sorties que nous devons calculer, f est la séquence incomplète qui est complétée à chaque pas du calcul).

Nous introduisons les deux variables f et n satisfaisant l'invariant :

$$f \in \mathbb{N} \nrightarrow 0\,..\,MX \wedge$$
$$n \in \mathbb{N} \wedge$$
$$\mathrm{dom}(f) = 0\,..\,n \wedge$$
$$f \subseteq y$$

Initialement le système n'a encore produit aucune sortie :

$$f, n := \{0 \mapsto 0\}, 0\ .$$

L'évolution du système se fait en avançant la position n dans la séquence d'entrée et en écrivant la valeur v dans la séquence de sortie f. La façon dont v est calculé satisfait l'invariant $f \subseteq y$, en fait à ce stade, le calcul de v est la réécriture de la définition de y par rapport à x.

$evo \mathrel{\widehat{=}}$ ANY p, v WHERE
$p = n{+}1 \wedge$
$(p < N \Rightarrow v = 0) \wedge$
$(p \geq N \Rightarrow v = \sum j \cdot (j \in 0 \,..\, N{-}1 \mid x(p{-}j) \otimes b(j) \ominus a \otimes f(p{-}1)))$
THEN
$f, n := f \Leftarrow \{p \mapsto v\}, p$
END

Cette spécification produit un ensemble d'obligations de preuve. Une concerne la propriété que le système ne se bloque pas, c'est-à-dire que l'événement *evo* peut toujours se produire. Les autres obligations de preuve concernent la préservation de l'invariant. Dans ce cas cela signifie que les valeurs $f(n)$ produites correspondent bien aux valeurs $y(n)$ comme cela est exprimé par l'invariant $f \subseteq y$. Les obligations de preuves sont générées automatiquement par l'outil AtelierB [28] (cf. section 6.9 pour les statistiques sur les preuves).

6.3 Introduction des calculs intermédiaires

Dans ce premier raffinement, on éclate l'événement *evo* en deux événements. Un nouvel événement *sum* est introduit, dans l'ordre de déclenchement des événements il prend place avant *evo* et est chargé de calculer la somme de convolution. Le résultat est mis dans une variable s. La position courante dans la séquence d'entrée est maintenant m, qui anticipe la prochaine position dans la séquence (qui est seulement connue quand l'événement *evo* se déclenche). La valeur initiale de s est assignée de façon non déterministe car cette valeur n'a pas d'importance puisque la variable s est toujours assignée correctement avant d'être utilisée.

$$m := 0 \parallel s :\in 0 \,..\, MX$$

En terme de circuit on peut interpréter s comme un signal qui passe une valeur de la première partie *sum* à la deuxième partie *evo* comme cela est exprimé par l'invariant de collage $m \notin \mathrm{dom}(f) \wedge m \geq N \Rightarrow s = conv(m)$:

$sum \mathrel{\widehat{=}}$ ANY v WHERE
$m \in \mathrm{dom}(f) \wedge$
$v \in 0 \,..\, MX \wedge$
$(m{+}1 \geq N \Rightarrow v = conv(m{+}1))$
THEN
$s, m := v, m{+}1$
END

$evo =$ ANY v WHERE
$m \notin \mathrm{dom}(f) \wedge$
$(m < N \Rightarrow v = 0) \wedge$
$(m \geq N \Rightarrow v = s \ominus a \otimes f(m{-}1))$
THEN
$f := f \Leftarrow \{m \mapsto v\}$
END

6.4 Ordonnancement par des jetons de contrôle

La structure du système devenant plus compliquée, nous utilisons des jetons de contrôle pour activer ou désactiver les événements. La variable *ct* est utilisée dans ce but. Ce pas de raffinement introduit les trois principaux composants du système linéaire, modélisés par les événements *sum*, *tim*, et *two*. L'événement *sum* est la spécification de la convolution que nous développerons séparément. L'événement *tim* spécifie un filtre qui produit la séquence initiale de zéros en sortie. Forcer le composant calculant la convolution à ne pas produire de "bruits" amènerait à le compliquer inutilement. La variable rz contient la

partie restante du calcul qui était localisé dans l'événement *evo* dans le raffinement précédent. La variable rz représente également la valeur de sortie du circuit que nous devons dériver. À partir de l'invariant de collage on peut voir le lien qui existe entre les valeurs sauvegardées dans rz à différents moments et les autres variables :

$$\begin{array}{l}(ct = EVO \wedge m < N \Rightarrow rz = 0) \wedge \\ (ct = EVO \wedge m \geq N \Rightarrow rz = f - a{*}f(m{-}1)) \wedge \\ (ct = TWO \wedge m > 1 \Rightarrow rz = f(m{-}1)) \wedge \\ (ct = SUM \wedge m > 0 \Rightarrow rz = f(m))\end{array}$$

La valeur initiale de rz est arbitraire puisque la première sortie est produite par l'événement *tim* (car $N > 1$).

$$ct := SUM \parallel rz :\in 0 \,..\, MX \parallel m := 0 \parallel s :\in 0 \,..\, MX \parallel f := \{0 \mapsto 0\}$$

Les deux nouveaux événements *tim* et *two* sont présentés ci-dessous. On peut aussi observer la manière dont l'événement *evo* est modifié pour prendre en compte le calcul de la variable rz, cette transformation est justifiée par l'invariant de collage donné ci-dessus.

$sum \mathrel{\widehat{=}}$ ANY v WHERE
  $ct = SUM \wedge$
  $v \in 0 \,..\, MX \wedge$
  $(m{+}1 \geq N \Rightarrow v = conv(m{+}1))$
THEN
  $s, m, ct := v, m{+}1, TWO$
END

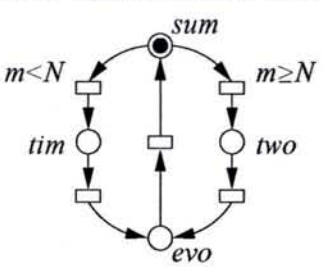
$tim \mathrel{\widehat{=}}$ WHEN
  $ct = TWO \wedge$
  $m < N$
THEN
  $rz, ct := 0, EVO$
END

$two \mathrel{\widehat{=}}$ WHEN
  $ct = TWO \wedge$
  $m \geq N$
THEN
  $rz, ct := s \ominus a{\otimes}rz, EVO$
END

$evo \mathrel{\widehat{=}}$ WHEN
  $ct = EVO$
THEN
  $f(m), ct := rz, SUM$
END

La figure 6.3 montre de quelle façon les événements sont ordonnancés par les jetons de contrôle. Cette représentation d'un système B événementiel est spécialement utile quand un modèle devient plus large. Nous n'exploitons pas cette représentation formellement, nous utilisons plutôt des réseaux de Petri informellement pour aider à la compréhension de systèmes complexes.

FIG. 6.3 – Flot de contrôle comme un réseau de Petri



6.5 Somme de convolution par pipeline

Dans ce raffinement, la somme de convolution est dérivée en faisant un calcul par pipeline. Nous raffinons également le filtre qui produit la séquence initiale de zéros en utilisant un compteur cn implantable.

La m^{ieme} étape du pipeline est représentée par la variable rs telle qu'elle est caractérisée par cet invariant :

$$
\begin{array}{l}
rs \in 0\,..\,N \rightarrow 0\,..\,MX \wedge \\
rs(0) = 0 \wedge \\
\forall r \cdot (r \in 1\,..\,N \wedge m \geq r \Rightarrow rs(r) = \sum j \cdot (j \in 1\,..\,r \mid x(m{-}r{+}j) \otimes b(N{-}j)))
\end{array}
$$

La somme de convolution est disponible dans $rs(N)$ avec un délai de N cycles. Notons que l'ordre dans lequel la somme est faite a été renversé, en se basant sur la propriété suivante :

$$
\begin{array}{l}
\forall m \cdot (m \in \mathbb{N} \wedge m \geq N \Rightarrow \\
\quad \sum j \cdot (j \in 1\,..\,N \mid x(m{-}N{+}j) \otimes b(N{-}j)) = \sum j \cdot (j \in 0\,..\,N{-}1 \mid x(m{-}j) \otimes b(j)))
\end{array}
$$

Le compteur cn commence à $N{-}1$ à l'initialisation et compte à rebours jusqu'à 0. À partir de là, la valeur $rs(N)$ (respectivement s) est utilisée comme entrée par le dernier composant du circuit. Les valeurs initialement présentent dans le pipeline sont arbitraires, excepté pour $rs(0)$ qui doit être constamment zéro.

$$rs : (rs \in 0\,..\,N \rightarrow 0\,..\,MX \wedge rs(0) = 0) \parallel cn := N{-}1$$

$tim \mathrel{\widehat{=}}$ WHEN
  $ct = TWO \wedge$
  $cn > 0$
THEN
  $rz, cn, ct := 0, cn - 1, EVO$
END

$two \mathrel{\widehat{=}}$ WHEN
  $ct = TWO \wedge$
  $cn = 0$
THEN
  $rz, ct := s \ominus a \otimes rz, EVO$
END

Dans l'événement *sum* toutes les valeurs du pipeline $rs(1)$, $rs(2)$, ..., $rs(N)$ sont modifiées en même temps. Pour illustrer la façon dont le calcul de rs se fait grâce à l'événement *sum*, on illustre les valeurs successives sur un pipeline de longueur 3 ($N = 3$). La ligne $m = i$ correspond à la $i^{ème}$ occurrence de l'événement *sum*. On peut constater que pour calculer la valeur de $rs(2)$ on utilise la valeur de $rs(1)$ de la ligne précédente ; de la même manière pour calculer $rs(3)$ on utilise la valeur de $rs(2)$ de la ligne précédente. Ainsi, à chaque étape, pour calculer $rs(i)$ on utilise la valeur de $rs(i{-}1)$ de l'étape précédente, de $x(m+1)$ et des valeurs $b(0)$, $b(1)$ et $b(2)$.

	$rs(1)$	$rs(2)$	$rs(3)$
$m = 0$ $(cn = 2)$	$x(1) * b(2)$		
$m = 1$ $(cn = 1)$	$x(2) * b(2)$	$x(1) * b(2)$ $+x(2) * b(1)$	
$m = 2$ $(cn = 0)$	$x(3) * b(2)$	$x(2) * b(2)$ $+x(3) * b(1)$	$x(1) * b(2)$ $+x(2) * b(1)$ $+x(3) * b(0)$ $= conv(3)$
$m = 3$ $(cn = 0)$	$x(4) * b(2)$	$x(3) * b(2)$ $+x(4) * b(1)$	$x(2) * b(2)$ $+x(3) * b(1)$ $+x(4) * b(0)$ $= conv(4)$

$sum \mathrel{\widehat{=}}$ ANY qs WHERE
$ct = SUM$
$qs \in 0..N \rightarrow \mathbb{Z}$
$qs = \{0 \mapsto 0\} \cup \lambda r \cdot (r \in 1..N \mid rs(r-1) \oplus x(m+1) \otimes b(N-r))$
THEN
$rs, s, m, ct := qs, qs(N), m+1, TWO$
END

Notons que dans le calcul de la valeur qs dans l'événement sum le terme $x(m+1)$ est indépendant de l'indice r. Ainsi, $x(m+1)$ est une constante dans cette expression et peut être remplacée par une variable d'état qui sauvegarde la valeur d'entrée $x(m+1)$. Ceci est fait au pas de raffinement suivant.

6.6 Implantation du système linéaire

Nous raffinons les événements sum et evo de telle sorte que le circuit résultant prend en entrée une valeur ix à chaque cycle et présente en sortie une valeur oy. Deux nouveaux événements one et sig sont introduits pour produire respectivement les valeurs ix et oy. L'ensemble des jetons de contrôle est étendu par les jetons ONE et SIG. Les variables ix et oy sont initialisées de manière non déterministe car ces valeurs n'ont pas d'importance.

$$ix :\in \mathbb{Z} \parallel oy :\in \mathbb{Z}$$

Aucun des événements modélisant le circuit (sum, tim, two, et sig) ne contient de référence aux entités qui appartiennent à l'environnement du circuit, c'est-à-dire les séquences f, x et y, ou la position k (raffinement de m) dans les séquences. Par conséquent, et puisque toutes les variables utilisées ont des types implantables, nous obtenons un modèle implantable du circuit.

L'événement one appartient à l'environnement. A ce stade, l'environnement est constitué de deux événements : un producteur one et un consommateur evo. L'événement sig appartient au circuit. A la fin d'un processus de raffinement, les entrées et les sorties doivent être modélisées explicitement par des variables dédiées, c'est le cas ici par les variables ix et oy.

$one \mathrel{\widehat{=}}$ WHEN
$cu = ONE$
THEN
$k, ix, cu := k+1, x(k+1), SUM$
END

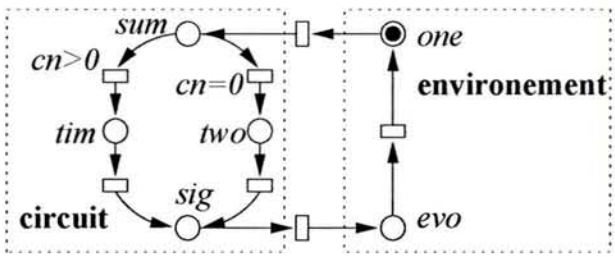
$sig \mathrel{\widehat{=}}$ WHEN
$cu = SIG$
THEN
$oy, cu := rz, EVO$
END

L'événement sum a été raffiné de telle sorte qu'il ne contient plus d'expressions imbriquées. La même chose a été faite pour l'événement two que nous ne réécrivons pas ici (l'expression $s \ominus a \otimes rz$ est séparée en deux : d'abord une multiplication et ensuite une soustraction). Ceci est nécessaire pour bien séparer les composants de base utilisés (additionneurs, multiplicateurs ...).

$sum \mathrel{\widehat{=}}$ WHEN
$cu = SUM$
THEN
$ts := \lambda r \cdot (r \in 1..N \mid ix \otimes b(N-r))$;
$rs := \lambda r \cdot (r = 0 \mid 0) \cup \lambda r \cdot (r \in 1..N \mid rs(r-1) \oplus ts(r))$;
$s, cu := rs(N), TWO$
END

Nous avons utilisé ci-dessus la composition séquentielle (;) de substitutions. Dans le B événementiel tel qu'il est définit aujourd'hui ceci n'est plus possible, cela l'était lorsque nous avons commencé nos travaux et les outils que nous utilisons supportent cette composition séquentielle.

FIG. 6.4 – Ordonnancement des événements



La figure 6.4 montre une représentation par réseau de Petri de l'ordonnancement des événements et la partition entre circuit en environnement. Les jetons de contrôle représentés dans le dernier modèle B par la variable *cu* et précédemment par la variable *ct* sont reliés suivant cet invariant de collage :

$$(cu = ONE \Rightarrow ct = SUM) \wedge (cu = SIG \Rightarrow ct = EVO) \wedge (cu \notin \{ONE, SIG\} \Rightarrow ct = cu)$$

6.7 Regroupement

Nous avons maintenant atteint une implantation du système linéaire en B événementiel. Ce modèle ne peut pas être traduit de manière simple vers un langage cible permettant la synthèse du circuit. Un raffinement spécial est requis pour regrouper en un seul événement les quatre événements correspondant au circuit. Cet unique événement contiendra une substitution représentant le circuit. Les événements restants sont ceux correspondant à l'environnement du circuit.

Il faut également supprimer du modèle les variables correspondant à l'ordonnancement des événements. En fait le regroupement des événements est une façon d'implanter l'ordonnancement de façon synthétisable. Si le modèle B obtenu est implantable, le non déterminisme dans la fonctionnalité doit avoir disparu. S'il existe encore du non déterminisme dans l'ordonnancement des événements, par exemple si deux événements peuvent se produire à un même moment, cela peut signifier que les deux parties du circuit correspondantes peuvent se faire en parallèle (à condition qu'il n'y ait pas de communication entre les deux parties du circuit). Par contre lorsque l'ordonnancement impose un ordre, cela peut s'implanter en mettant les deux parties en séquence (en utilisant la substitution de composition compositionnelle " ;"). Lorsque deux événements décrivent le comportement du circuit dans des situations différentes, ils peuvent être composés en utilisant une substitution conditionnelle "IF". Des règles de composition des événements peuvent être appliquées. Les deux règles utilisées dans cet exemple (cf. [2]) sont décrites ci-dessous (cf. section 3.5.2 page 52).

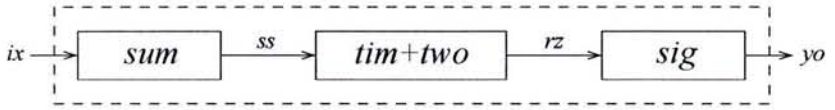
WHEN P ∧ Q THEN S END	A = WHEN P THEN S END
WHEN P ∧ ¬ Q THEN T END	B = WHEN Q THEN T END
WHEN P THEN	P et Q ne contiennent que de l'ordonnancement
IF Q THEN S ELSE T END	L'ordonnancement est A puis B
END	WHEN P THEN S ; T END

Les événements *tim* et *two* peuvent être combinés en un seul événement contenant une substitution conditionnelle. L'événement résultant de cette composition peut lui-même être composé séquentiellement avec les événements *sum* et *sig*. Ainsi, le circuit complet suit le schéma montré sur la figure 6.5.

6.8 Modèle BHDL

Le développement est générique au sens où la valeur maximale MX des nombres utilisés n'est pas déterminée. C'est également le cas pour la taille N de la convolution. Avant de traduire le modèle B

FIG. 6.5 – Circuit Complet pour la fonction système spécifiée



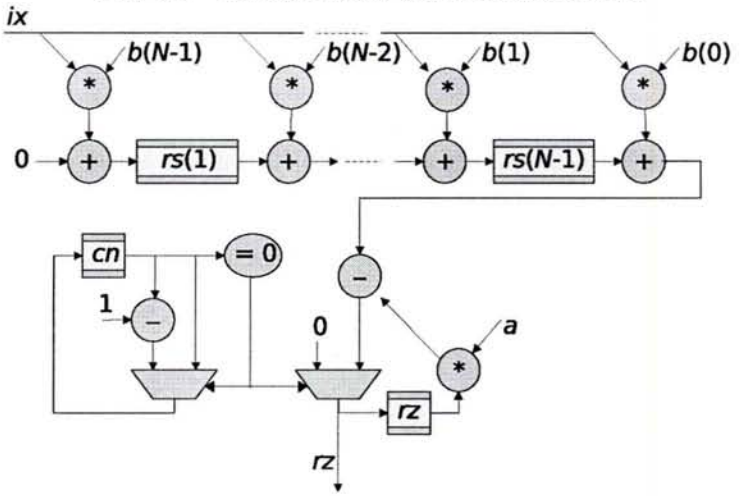
vers une description dans un langage permettant la synthèse il faut fixer ces valeurs. Ceci est fait dans une machine BHDL à part (ici la machine s'appelle PARAM). Nous avons choisi $N = 8$ et $MX = 255$ (nombres sur 8 bits). Une machine spéciale appelée BHDL contient la définition d'un certain nombre de types implantables, comme le type $UINT8$ correspondant aux nombres sur 8 bits et $UINT3$ pour les nombre sur 3 bits. Les deux machines sont vues (via la clause SEES) par la machine $LINEAR$ décrite ci-dessous modélisant le système linaire. La clause OPERATIONS est le résultat du regroupement des événements du modèle B précédent.

MACHINE
  $LINEAR$
SEES
  $BHDL, PARAM$
INPUTS
  a, b, ix
OUTPUTS
  oy
VARIABLES
  rs, ts, s, cn, rz, tz
INVARIANT
  $rs \in UINT8 \wedge ts \in UINT8 \wedge \ldots \wedge tz \in UINT8 \wedge cn \in UINT3$
INITIALISATION
  $rs := \lambda r \cdot (r \in 0 \,..\, N \mid 0) \parallel cn := N{-}1 \parallel s :\in UINT8 \parallel \ldots$
OPERATIONS
  BEGIN
    $ts := \lambda r \cdot (r \in 1 \,..\, N | ix \otimes b(N{-}r))$;
    $rs := \lambda r \cdot (r = 0 \mid 0) \cup \lambda r \cdot (r \in 1 \,..\, N \mid rs(r{-}1) \oplus ts(r))$;
    $s := rs(N)$;
    IF $cn > 0$ THEN
      $rz, cn := 0, cn{-}1$
    ELSE
      $tz := a \otimes rz$;
      $rz := s \ominus tz$
    END ;
    $oy := rz$
  END
END

Les variables sont partitionnées en 3 classes : les entrées, les sorties et les variables ordinaires. Ceci permet de fixer l'interface du circuit avec son environnement. A partir des variables ordinaires, on peut

inférer automatiquement les registres et les fils du circuit physique. La figure 6.6 donne une représentation graphique du circuit obtenu.

FIG. 6.6 – Circuit obtenu en fin de raffinement



A partir de ce modèle BHDL, on peut obtenir par traduction automatique la description du circuit dans un langage de description de circuit comme VHDL ou SystemC au niveau transfert de registre (cf. chapitre 14).

6.9 Quelques statistiques sur les preuves

La figure 6.7 montre pour chaque machine B utilisée dans ce développement le nombre d'obligations de preuve générées par les outils. Les modèles LTI à LTI5 désignent la séquence des raffinements. Toutes les preuves ont été faites en utilisant l'interface Click'n'Prove [7].

La machine $MODULO$ contient la propriété $\forall(n,p,m)\cdot(n \in \mathbb{N} \wedge p \in \mathbb{N} \wedge m \in \mathbb{N}_1 \Rightarrow (n+p)\ mod\ m = (n\ mod\ m + p\ mod\ m)\ mod\ m)$ difficile à prouver par manque de règles sur l'opérateur modulo. La machine $TYPES$ contient des propriétés sur le type $0..MX$ et définie l'addition, la somme, le produit et la soustraction modulo $MX+1$. Cette machine utilise une machine INDUCTION qui contient un théorème permettant d'effectuer des preuves par induction en B. La table liste plus de raffinements que ce qui a été présenté car certains d'entre eux sont utilisés pour la décomposition de la preuve. Par exemple, les événements *one* et *sig* ont en fait été introduits en deux pas de raffinement. Nous n'avons présenté que les étapes importantes apparues lors du processus de raffinement.

6.10 Conclusion

Ce chapitre a montré comment on pouvait modéliser en B un système dont la spécification initiale est donnée par une équation récurrente. Nous avons d'abord fait une spécification en B de ce système puis nous l'avons raffiné pas à pas jusqu'à obtenir un niveau implantable à partir duquel nous avons pu traduire le système en VHDL, le simuler et le synthétiser. Cela a également été l'occasion de montrer comment le système pouvait se résumer sur un réseau de Petri, donnant ainsi une vision globale de l'enchaînement des événements.

FIG. 6.7 – (1) Nombre de commandes du prouveur interactif utilisées pour faire l'ensemble preuves qui n'ont pas été déchargées automatiquement. (2) Nombre d'appels au prouveur automatique dans les preuves interactives. (3) nombre moyen de commandes (y compris appels au prouveur automatique) utilisées pour les preuves.

Modèle	nombre d'obligations de preuve	nombre de preuves interactives	nombre de preuves automatiques	taille script de preuve (1)	appels au prouveur automatique (2)	taille moyenne (3)
LTI	7	3	4	83	16	27,67
LTI1	12	1	11	31	12	31
LTI2	16	1	15	8	3	8
LTI3	18	5	13	462	151	92,4
LTI4	19	0	19	-	-	-
LTI5	2	0	2	-	-	-
INDUCTION	1	1	0	34	11	34
TYPES	13	8	5	750	230	93,75
MODULO	3	3	0	80	28	26,7
Total	91	24,1%	78,9%			65,8

Chapitre 7

Étude de cas : contrôleur d'accès à un bus série

Ce chapitre présente l'étude de cas du standard SAE J1708 décrit dans le document [83]. Cette étude de cas a été réalisée dans le cadre du projet européen PUSSEE [80] et a été choisie par Volvo. Elle a également donné lieu à une publication [93]. Le standard SAE J1708 définit un lien de communication en série pour des engins de chantier. Plusieurs composants sont connectés à un bus, ils peuvent envoyer des messages sur le bus et recevoir des messages. Les composants ne peuvent pas communiquer d'une autre façon en eux. Il peut y avoir des conflits d'accès au bus, i.e. que plusieurs composants souhaitent utiliser le bus au même moment. Il n'y a pas d'arbitre pour gérer l'accès au bus. Les conflits sont résolus par un protocole de communication distribué sur tous les composants. Tous les contrôleurs d'accès au bus sont supposés identiques. La spécification originale décrit un protocole probabiliste : les conflits entre les composants peuvent entraîner un blocage du système. Nous montrons par cette étude de cas qu'il est possible de résoudre les conflits de manière déterministe et en évitant le blocage du système.

Nous avons modélisé ce système en utilisant la méthode B. Le système est d'abord décrit de façon abstraite. Puis, étape par étape, le système est décrit de manière de plus en plus détaillée en utilisant le raffinement proposé par la méthode B. Le processus de raffinement permet de développer le système pas à pas. Chaque pas introduit un aspect du système. Cela permet de mieux gérer la complexité d'un modèle (à comparer avec un modèle écrit d'un seul coup avec tous les détails). Le raffinement ajoute des détails au modèle, distribue la complexité du système et de sa preuve de correction. Il rend également plus simple l'explication, la communication pour arriver à une validation étape par étape d'un modèle.

Le dernier raffinement doit être proche d'une implantation. En particulier, notre but est d'obtenir un modèle du système dans lequel les descriptions des contrôleurs ne se réfèrent qu'à leurs propres états locaux. Nous disons que les composants sont *indépendants*. Dans le langage B, le système est décrit par une collection d'événements. Les événements permettent de modéliser la concurrence. Lorsque plus d'un événement peut se déclencher à un moment donné, le choix duquel se déclenche effectivement est non déterministe. L'indépendance des composants assure la concurrence entre les contrôleurs car elle n'impose pas d'ordre dans l'occurrence des événements.

7.1 Présentation de l'étude de cas

Il peut exister de multiples manières de représenter un système de manière abstraite. La représentation abstraite du système est orientée par la vision globale que l'on a du système.

Dans cette étude de cas, la spécification du système est décrite de manière très concrète au niveau de l'implantation. C'est le cas de la plupart des spécifications actuelles. Les spécifications expriment le plus souvent la manière d'implanter un concept plutôt que la description du concept lui-même. Dans le document [83], il est indiqué de quelle façon le protocole peut être implanté mais le document ne donne pas les propriétés elles-mêmes du protocole.

Une première étape a donc consisté à interpréter la spécification donnée [83] de façon à lui donner un sens indépendant de la façon de l'implanter. Notre façon d'interpréter le standard a été la suivante :

- Le système est composé de plusieurs nœuds communiquant en envoyant des messages sur un bus.
- À un instant donné, un seul composant au plus est en train d'envoyer un message.
- Lorsqu'un message est en cours d'envoi, il n'est ni interrompu ni modifié.
- L'envoi d'un message est précédé par une phase de compétition permettant d'attribuer le bus à un des composants en demande.
- Cette compétition termine toujours par l'attribution du bus à l'un des compétiteurs

7.1.1 Présentation du système - Glossaire

Cette section présente le système que nous avons modélisé. Nous le présentons sous la forme d'un glossaire des termes utilisés pour désigner les éléments du système, aussi bien les composants que les éléments du protocole. Ce glossaire a été fait à partir du document du standard SAE J1708 [83], sa réalisation est une partie importante du travail à faire en amont de la modélisation. C'est pendant cette étude que des discussions peuvent apparaître sur des problèmes d'interprétation du standard.

Le système

Le système consiste en un réseau de composants, qui communiquent via un bus série. Tous les composants sont connectés à l'unique bus par un contrôleur. Chacun des composants peut envoyer des messages sur le bus, les messages sont reçus par tous les autres composants. Un protocole permet de gérer les conflits d'accès au bus.

Dans le glossaire ci-dessous, lorsque nous nous référons à une définition donnée dans le standard, nous l'indiquons en mettant entre parenthèses le numéro de la définition correspondante dans le standard.

Réseau (5.1) Le réseau connecte tous les composants du système via un bus global.

Nœud (3.10) Les composants sont connectés au bus par un *nœud*. Le nœud est la partie contrôleur du composant. Un nœud peut être un récepteur ou un *transcepteur*. Un transcepteur peut être un transmetteur ou un récepteur.

État du bus (4.2) Le bus porte une seule valeur binaire à la fois. Le bus peut être dans deux états : un état logique haut '1' (qui sera représenté par l'entier 1 dans les modèles) ou un état logique bas '0' (qui sera représenté par l'entier 0 dans les modèles). La valeur du bus dépend de tous les nœuds. Lorsque certains nœuds envoient '0' sur le bus et que d'autres envoient '1', le bus est dans l'état '0'. On dit que l'état '0' domine le bus, et que l'état '1' est récessif. C'est-à-dire que si un composant envoie '0' sur le bus alors le bus est nécessairement '0', quelles que soient les valeurs envoyées par les autre nœuds. Lorsque tous les composants envoient '1' sur le bus alors l'état du bus est '1'.

Transmetteur inactif Un transmetteur est *inactif* quand il n'est pas en train d'envoyer un message sur le bus. Quand un transmetteur est inactif, il envoie la valeur '1' en permanence sur le bus.

Bus en conflit (5.2.2) Si deux transmetteurs envoient en même temps des messages (différents) sur le bus, on dit qu'il y a un conflit.

Domination de '0' (4.2.1, 4.2.2) Le bus est dans l'état '0' quand un ou plusieurs nœuds envoient '0' sur le bus. Sinon, le bus est dans l'état '1', i.e. quand tous les transmetteurs envoient '1' sur le bus. L'état '0' est dominant lorsque le bus est en conflit.

Message (6.3.5) Les composants utilisent le bus pour envoyer et recevoir des messages. Un *message* est composé d'un nombre fini de caractères.

Caractère (6.2) Un *caractère* est une liste de 10 bits qui sont envoyés un par un sur le bus. Le premier bit d'un caractère, i.e. le premier bit envoyé, est toujours '0', il est appelé le *bit de départ*. Le dernier bit d'un caractère est toujours '1', il est appelé le *bit d'arrêt*. Les 8 bits intérieurs restants sont les données qui sont transmises. Les données sont envoyées le bit de poids faible d'abord.

MID (6.3.1) Le premier caractère d'un message est le *MID* (Message IDentification character). Le MID identifie le nœud (de manière unique) qui envoie le message.

Les 8 bits de données du MID forment un nombre (entre 0 et 255) qui identifie le transmetteur d'un message. Par conséquent, le réseau est supposé ne pas avoir plus de 256 transmetteurs.

Notre modèle est moins contraignant en ce qui concerne la taille d'un caractère. Nous définissons une constante (*MAX_BIT_CHAR*) pour modéliser la taille d'un caractère mais nous ne lui fixons pas sa valeur. Ainsi, cette constante peut être considérée comme un paramètre du modèle. Le modèle est développé et prouvé en utilisant cette constante non déterminée. Le modèle peut donc être utilisé pour modéliser un réseau avec plus de transmetteur que par le standard (en augmentant la taille des caractères).

De la même façon, dans le standard [83], il est indiqué qu'un message ne doit pas excéder 21 caractères. Cette limitation n'apparaît pas dans notre modèle car il n'a aucune influence sur le protocole.

Priorité (5.2.1.2) Une *priorité* est assignée à chaque message. Il s'agit d'un nombre entre 1 et 8. Cette priorité est utilisée dans le protocole pour définir la longueur d'une phase d'attente avant d'accéder au bus en écriture. Plus la longueur de la phase d'attente est faible, plus le transmetteur a un avantage sur les autres.

Cependant cette priorité n'est pas absolue. Par exemple, si un nœud nd_1 a une priorité de 1 et un nœud nd_2 une priorité de 8, nd_1 est considéré avoir une plus forte priorité que nd_2. Pourtant, si nd_2 commence sa phase d'attente suffisamment tôt avant nd_1, alors nd_2 pourra accéder au bus avant nd_1. Bien que $nd1$ ait une plus forte priorité que $nd2$, $nd2$ peut transmettre son message avant $nd1$.

De plus, il existe une autre, implicite, priorité. Elle est dû au fait que 0 domine le bus. Nous allons voir que la façon dont les conflits d'accès au bus sont réglés favorise les nœuds qui envoient des '0' les plus tôt (pendant la transmission de leur MID).

Résolution des conflits

Notre objectif est de modéliser la résolution des conflits entre les transmetteurs. La réception des messages n'est pas explicitement modélisée.

Les données sont envoyées sur le bus bit par bit. Chaque bit a une durée, la même durée pour tous les bits. Il n'est pas nécessaire de modéliser cette durée explicitement dans notre modèle. Nous utilisons un modèle cyclique ; le bus reçoit les données en début de cycle, ensuite le bus est stable jusqu'au cycle suivant.

La durée d'un caractère est la durée nécessaire pour envoyer tous les bits, un par un, d'un caractère.

Durée d'un bit (3.3) La *durée d'un bit* est la durée d'une unité (bit) d'information. Cela correspond à la période du cycle.

Durée d'un caractère (3.4) La *durée d'un caractère* est la durée nécessaire pour envoyer un caractère en entier.

Le premier bit envoyé d'un caractère est toujours '0', et '0' domine le bus. Ainsi, si un caractère est en train d'être envoyé, le bus se trouve nécessairement dans l'état '0' au moins dans une période inférieure ou égale à la durée d'un caractère. Ainsi, si on observe que le bus reste dans l'état '1' pour au moins la durée d'un caractère, on peut conclure qu'aucun caractère n'est en train d'être envoyé sur le bus. Nous appelons cette condition la *période de latence*. Telle que nous la décrivons ici, il s'agit d'une modification de la définition de [83], voir la remarque dans la section 7.1.2.

Période de latence La *période de latence* est la condition qui existe lorsque le bus reste continuellement dans l'état '1' pendant au moins la durée d'un caractère.

Durée d'accès au bus (5.2.1.1) La *durée d'accès au bus* est un temps égal au minimum de la longueur d'une période latence auquel on ajoute deux durées d'un bit multipliées par la priorité du message : $T_a = T_i + [2 * T_b] * P$.

Avant d'accéder au bus, un nœud doit attendre pour une période décrite par la *durée d'accès au bus*. Dans notre modèle, cette durée est divisée en deux parties. D'une part la période de latence (T_i), et d'autre part le terme de priorité ($[2 * T_b] * P$). La première partie est prise en considération pour déclarer que le bus est libre.

Bus libre Le bus est considéré comme libre une fois qu'une période de latence a été observée. Lorsqu'un nœud termine d'envoyer son message, le bus n'est pas considéré libre immédiatement, mais après une période de latence

La deuxième partie, concernant le terme de priorité, n'est pas modélisée aussi explicitement dans notre modèle, nous considérerons que le temps d'attente est donné directement en entrée, ce qui évite d'inclure un calcul inutile à notre modèle. On considère que c'est le composant souhaitant envoyer un message qui fait ce calcul et pas le contrôleur lui-même, nous ne modélisons que le contrôleur, le reste du composant est considéré comme faisant partie de l'environnement du contrôleur.

Phase d'attente La phase d'attente correspond à la première phase du protocole d'accès en écriture au bus. Chaque nœud (après qu'une période de latence ait été observée) doit attendre un certain temps déterminé par sa priorité ($[2 * T_b] * P$).

Durant, cette phase, aucun nœud n'accède au bus. Ainsi le bus reste continuellement dans l'état '1'. Lorsqu'un composant termine sa phase d'attente, il écrit '0' sur le bus. Ce bit '0' correspond au bit de départ de son MID. Notons que plusieurs nœuds peuvent terminer leurs phases d'attente au même moment. Les nœuds qui sont toujours en train d'attendre observent le bit '0' sur le bus et doivent stopper leur attente et devenir des récepteurs. Cette étape est appelée *phase d'élimination.*

Phase d'élimination Chaque nœud qui est encore en train d'attendre, lorsqu'au moins un autre nœud a terminé d'attendre, doit stopper son attente. La compétition entre les nœuds ayant fini leurs attentes commence.

Tous les nœuds qui ont complété leurs phases d'attente au même moment tentent d'envoyer leur MID sur le bus en même temps. A chaque cycle, tous les compétiteurs envoient un bit de leurs MIDs. Supposons que le bus porte la valeur '0'. Parce que '0' domine le bus, les nœuds qui ont envoyé '1' sur le bus n'observent pas ce qu'ils y ont envoyé, ils détectent une *collision.* Les nœuds qui ont envoyé '1' sur le bus doivent quitter la compétition, ils deviennent des récepteurs. Si le bus porte la valeur '1', cela signifie que tous les nœuds ont envoyé le bit '1' sur le bus, il n'y a donc aucune collision.

Dans notre modèle, cette vérification de la valeur portée par le bus est faite à chaque cycle. Ainsi, une collision est détectée immédiatement et le nœud quitte immédiatement la compétition. Si le bus porte la valeur '0', il existe au moins un nœud qui a envoyé le bit '0' sur le bus. Si le bus porte la valeur '1', tous les nœuds ont envoyés le bit '1'. Ainsi, à chaque cycle, il y a au moins un nœud qui ne détecte pas de collision. Il y a toujours un nœud qui reste en compétition pour obtenir l'accès au bus. Comme tous les nœuds ont des MIDs différents, il ne peut rester qu'un seul nœud à la fin de la compétition, avant que le bit d'arrêt du MID soit envoyé sur le bus. Par conséquent, lorsqu'un nœud a réussi à envoyer l'ensemble de son MID sur le bus, il est le seul à l'avoir fait et il a gagné la compétition.

Phase de compétition A chaque cycle, tous les nœuds qui sont en train d'envoyer leurs MIDs sur le bus vérifient que la valeur qu'ils ont envoyée est la même que celle portée par le bus. Si ce n'est pas le cas, les nœuds qui détectent une collision doivent s'arrêter immédiatement d'envoyer leurs MIDs.

Détection d'une collision (5.2.2) "*Si un transmetteur détecte une collision, le transmetteur doit rendre le contrôle du bus après avoir complété la transmission du caractère courant, ou plus tôt si possible*". Dans notre modèle, le transmetteur *rend le contrôle du bus* immédiatement (il n'envoie pas le bit suivant). Donc, comme nous l'avons expliqué précédemment, avant la fin de l'envoi d'un MID complet, il reste toujours exactement un seul nœud en compétition qui gagne l'accès au bus pour envoyer son message.

On donne ci-dessous un petit exemple de compétition entre deux nœuds :

```
( ---- = phase d'attente )

c1      ------ 0 1 0 q1 q2 ...     --> le noeud c1 peut continuer à envoyer q1 q2 ...
c2        ---- 0 1 1              --> le noeud c2 s'arrête

bus   *latence* 0 1 0 q1 q2 ...
```

Dans la phase de compétition, un nœud est sélectionné pour obtenir l'accès au bus. Ainsi, ce nœud continue en envoyant le reste de son message. Nous appelons cela la *phase de message*. Durant cette phase, il n'y a plus de conflit sur le bus puisque tous les autres nœuds savent que le bus n'est pas libre et qu'il n'y a plus de nœud en compétition. Ainsi, le message peut être envoyé sans risque de la moindre occurrence d'un conflit. Notons que ceci est un modèle idéal, le standard ne précisant aucune procédure à suivre pour le traitement de défaillances du système.

Phase de message Après la compétition, le nœud gagnant envoie son message sur le bus.

Dans le standard [83] il est indiqué qu'il peut y avoir un délai entre l'envoi de deux caractères consécutifs. Ce délai ne doit pas excéder la durée de deux bits. Notons qu'en plus il n'est pas précisé dans le standard ci ce délai est fixé ou variable. Dans cette étude, nous nous sommes principalement focalisés sur la résolution des conflits d'accès au bus. En conséquence, nous avons fait le choix le plus simple : il n'y a aucun délai entre deux caractères consécutifs.

Espacement des caractères (6.3.1) "*La durée entre deux caractères dans un message ne doit pas excéder deux durées de bit*". Dans notre modèle , il n'y a aucun délai entre deux caractères d'un message.

Comme nous l'avons expliqué précédemment, le bus n'est considéré libre qu'après qu'une période de latence soit observée. Ainsi, la phase de message est suivie pas une période d'attente jusqu'à ce qu'une période de latence soit observée. Ceci est appelé la *phase de latence.*

Phase de latence Après la transmission d'un message, tous les nœuds attendent jusqu'à observer une période de latence du bus.

Pour résumer, le protocole est organisé en cinq phases. Il commence par une phase d'attente. Cette phase ne peut commencer qu'après qu'une période de latence ait été observée sur le bus. Tous les nœuds qui souhaitent envoyer un message sur le bus entrent dans cette phase d'attente. La phase d'attente se termine dès qu'un des nœuds au moins a complété son temps d'attente. Pour commencer l'accès au bus, les nœuds envoient le bit '0' sur le bus. Au même moment, les nœuds qui étaient encore en phase d'attente observent que le bus porte la valeur '0' et deviennent des récepteurs (phase d'élimination, qui ne dure qu'un seul cycle). Ensuite, une phase de compétition entre les nœuds restants sélectionne l'un d'entre eux. Le gagnant obtient l'accès au bus et envoie son message (phase de message). S'ensuit une phase de latence jusqu'à ce qu'une période de latence du bus soit observée et qu'ainsi des nœuds puissent commencer de nouveau une phase d'attente.

7.1.2 Remarques

Il peut être difficile d'interpréter une description de bas niveau. Lorsqu'on ne dispose pas de l'interprétation qu'avaient en tête les auteurs de la spécification, on peut interpréter certaines parties de façons différentes par rapports aux auteurs initiaux. Dans le cas de standard SAE J1708 que nous présentons ici, certaines parties de la spécification pouvaient mener à différentes interprétations. Certaines parties restaient floues et nécessitaient de faire des hypothèses sur la raison de ce flou.

Deux phrases en particulier nous ont posé question. Une première concerne la phase de compétition précédent l'envoi d'un message et la deuxième un temps d'attente nécessaire après l'envoi d'un message pour que les nœuds détectent que le bus est libre.

La première phrase dit : "*Si un nœud transmetteur détecte une collision, il doit abandonner la transmission après l'envoi du caractère courant ou plus tôt si possible*". Un caractère étant une série de 10 bits et une collision étant le fait que le transmetteur lise le bit '0' sur le bus alors qu'il envoie le bit '1'. Dans notre façon d'aborder ce système, il nous a paru possible (et c'est ce qui a été finalement fait) d'implanter le transmetteur de façon à ce qu'il interrompe immédiatement son transfert. Une interprétation possible de l'expression "*après l'envoi du caractère courant ou plus tôt si possible*" serait qu'un transmetteur pourrait être implanté de telle sorte qu'il ne puisse envoyer que des caractères en entier et ainsi de ne pouvoir s'interrompre en milieu de caractère. Cependant, rien dans la spécification ne précise une telle chose. Par ailleurs, cette phrase peut être interprétée de deux façons : soit elle laisse le choix entre les deux possibilités, soit il est nécessaire de prendre en compte les deux cas dans une implantation. Dans

ce cas précis, la façon d'interpréter cette phrase est importante puisque que si un transmetteur n'est pas capable de stopper une transmission en milieu de caractère alors la phase de compétition précédent l'envoi d'un message ne peut pas être résolue de manière systématique et peut même conduire à un blocage du système. Le choix que nous avons fait est d'interrompre une transmission immédiatement dès qu'une collision est détectée, ce qui nous permet de faire l'interprétation abstraite suivante du système : la phase de compétition mène systématiquement au choix d'un des compétiteurs. Un choix différent aurait conduit à un système qu'il aurait fallu étudier de manière probabiliste pour connaître les chances de blocage du système.

La deuxième phrase dit : *Le bus doit rester dans l'état logique haut ('1') pendant au moins dix durées de bit après la fin du dernier bit d'arrêt d'un message.* Ceci est la définition d'une période de latence suivant l'envoi d'un message. Nous ne définissons pas cette période de latence exactement de la même façon. Notre définition est moins contraignante (cf. la définition plus haut).

Premièrement, la définition du standard peut créer une inégalité entre les nœuds puisque dans la section 5.2.1.1 de [83], il est sous-entendu qu'il pourrait exister des transmetteurs qui pourraient "*ne pas pouvoir distinguer un bit d'arrêt d'un autre bit logique haut*". Ces nœuds auraient alors à attendre plus longtemps que les autres (il est dit "*19 bits logiques hauts*" dans le standard). Ainsi certains nœuds auraient des avantages sur les autres. Cependant, en principe, il est possible pour tout nœud de distinguer le dernier bit d'arrêt. En effet, tous les nœuds peuvent distinguer le premier bit de départ d'un message (un état '0' après une période de latence) ; et ils peuvent donc compter les bits à partir du premier bit de départ : chaque bit après dix cycles est un bit d'arrêt.

Deuxièmement, il n'est pas nécessaire d'attendre dix durées de bit *après le dernier bit d'arrêt.* On peut assurer qu'aucun nœud n'est en train d'utiliser le bus lorsque celui-ci est resté continuellement dans l'état '1' pendant la durée d'un caractère. Ainsi, la condition *Le bus doit rester dans l'état logique haut ('1') pendant au moins dix durées de bit ~~après la fin du dernier bit d'arrêt~~* est suffisante.

Par rapport à ces observations, nous avons modifié la définition de la *période de latence.* On peut dire que [83] est un raffinement de notre modèle : si nécessaire, on peut rallonger les périodes d'attente sans invalider le modèle. Ce serait comme si une période d'attente inutile était ajoutée au protocole.

7.2 Premier modèle abstrait

Cette machine modélise le bus tel qu'il serait vu par un observateur qui serait seulement capable de voir quel nœud est en train d'utiliser le bus. Le système est constitué d'un bus et de plusieurs nœuds représentés par leurs contrôleurs de bus. La machine abstraite modélise la plus importante propriété du système. Nous considérons uniquement le fait que le bus puisse être libre ou utilisé. Nous prouvons sur ce modèle abstrait que le bus ne peut être utilisé que par au plus un nœud à la fois.

7.2.1 L'état du système

L'ensemble des nœuds du système est modélisé par un ensemble abstrait *ND* (défini dans la clause SETS)

SETS
ND

L'état du bus est modélisé par une variable *wr* qui est un sous-ensemble des nœuds. La variable *wr* représente l'ensemble des nœuds qui sont en train d'utiliser le bus à un instant donné. Si l'ensemble *wr* est vide cela signifie que le bus est libre.

$$wr \subseteq ND$$

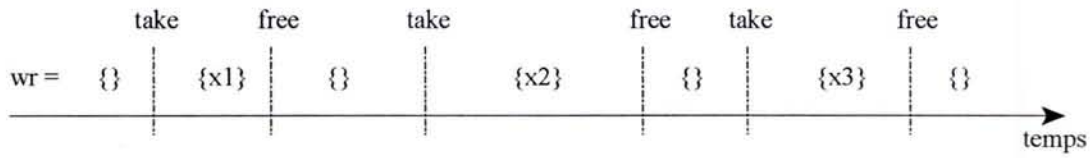
7.2.2 Évolution du système

Dans ce premier modèle abstrait du système, nous spécifions que le bus ne peut être utilisé que par un nœud au plus à la fois. Ceci peut s'exprimer par l'invariant suivant sur le système :

$$\forall\,(x,y).(x \in wr \wedge y \in wr \Rightarrow x=y)$$

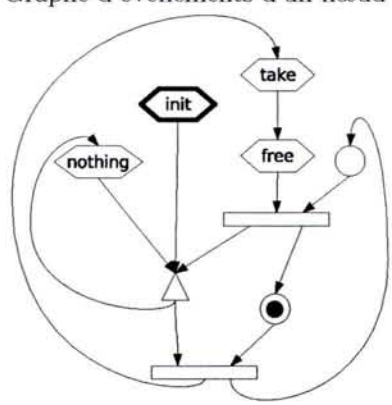
Cet invariant assure que l'ensemble wr est en permanence soit vide soit un singleton (s'il contient deux éléments, alors ces deux éléments ne sont pas distincts). On aurait tout aussi bien pu écrire $card(wr) \leq 1$ mais le prouveur de l'AtelierB gère relativement mal les expressions sur les cardinaux.

Nous pouvons modéliser cette évolution du système en utilisant deux événements. Un événement *take* qui permet à un nœud de prendre le contrôle du bus lorsque celui-ci est libre, et un événement *free* qui libère le bus. On peut représenter l'évolution dans le temps de la variable wr sur le graphique suivant. Sur ce graphique, les valeurs $x1$, $x2$ et $x3$ représentent des nœuds, non nécessairement distincts. Lorsque $wr = \{\}$ cela signifie que le bus est libre. Lorsque $wr = \{x1\}$, cela signifie que le nœud $x1$ a le contrôle du bus.



La figure 7.1 donne le graphe d'événement d'un nœud du système. Dans ce graphe nous utilisons un petit réseau de Petri de façon à imposer que le bus ne peut être utilisé que par au plus un composant. La transition vers l'événement *take* (c'est à dire que cet événement se déclenche) est conditionnée au fait que le jeton se trouve dans la place du bas. Lors de la transition, le jeton passe en même temps de la place du bas vers la place du haut. Il ne pourra repasser à la place du bas (et donc permettre à un noeud de prendre le bus) que lorsque le noeud qui a pris le bus déclenchera l'événement *free*. Cette représentation informelle est utilisée à titre d'illustration uniquement. Le jeton représente le bus : lorsqu'il est dans la place du bas, le bus est libre, lorsqu'il est dans la place du haut, le bus est utilisé.

FIG. 7.1 – Graphe d'événements d'un nœud du système



7.2.3 Événements

Pour modéliser ce comportement en B, nous avons besoin de spécifier l'initialisation du système qui initialise toutes les variables, en l'occurrence nous n'avons que la variable wr qui est initialisée avec la valeur $\varnothing$. C'est-à-dire que dans son état initial, le bus est libre.

INITIALISATION
$wr := \varnothing$

Nous avons déjà parlé des deux événements *take* et *free* qui modélisent respectivement la prise de contrôle du bus et la libération du bus. Nous donnons ci-dessous une façon décrire ces deux événements en B.

L'événement take ne peut se déclencher que lorsque le bus est libre, on ajoute donc la condition $wr = \varnothing$ dans sa garde. Par ailleurs, cet événement attribue le bus à un des nœuds. Une façon de modéliser cela est d'attribuer le bus à l'un des nœuds en écrivant la substitution $wr := \{nd\}$ où nd est un nœud choisi arbitrairement. Cela se fait en utilisant la forme ANY d'un événement. La substitution ANY nd WHERE $nd \in ND$ THEN ... END signifie que l'on choisit un nœud nd qui satisfait la propriété $nd \in ND$. C'est-à-dire que l'on choisit un nœud de manière non déterministe. Les raffinements suivants du système raffineront cette garde de façon à spécifier précisément quel nœud est choisi. À ce niveau d'abstraction il n'est pas encore nécessaire d'être plus précis.

take =
ANY nd WHERE
$wr = \varnothing \wedge$
$nd \in ND$
THEN
$wr := \{nd\}$
END

La forme ANY nd WHERE $nd \in ND$ THEN ... END de l'événement *take* peut aussi s'interpréter comme une forme compacte pour représenter un ensemble de n événements (où n serait la taille de l'ensemble abstrait ND, ensemble fini par définition en B) :

$$\begin{cases} take_{nd1} = \text{WHEN } wr = \varnothing \text{ THEN } wr := \{nd1\} \text{ END} \\ take_{nd2} = \text{WHEN } wr = \varnothing \text{ THEN } wr := \{nd2\} \text{ END} \\ \ldots \\ take_{nd_n} = \text{WHEN } wr = \varnothing \text{ THEN } wr := \{nd_n\} \text{ END} \end{cases}$$

Ces n événements sont en concurrences, et un seul d'entre eux pourrait effectivement se produire puisque lorsqu'un d'entre eux se produit, l'ensemble wr n'est plus vide donc la garde de tous les autres n'est plus vraie. L'utilisation de la forme ANY permet non seulement une écriture plus compacte mais également de ne pas préciser la taille de l'ensemble ND. Le modèle contenu est alors indépendant de la taille de l'ensemble ND et le système est prouvé quel que soit le nombre de nœuds dans le système.

L'événement *free* peut se produire lorsque le bus n'est pas libre, pour cela on ajoute la condition $wr \neq \varnothing$ dans sa garde. Son accès est de rendre le bus libre par la substitution $wr := \varnothing$.

free =
WHEN $wr \neq \varnothing$ THEN
$wr := \varnothing$
END

Les deux événements *take* et *free* pourraient suffire à modéliser le comportement du système. Mais nous nous imposons de modéliser explicitement le comportement de tous les nœuds dans toutes les circonstances. Si on observe les gardes des deux événements *take* et *free*, on constate que ces deux événements ne modélisent pas le comportement des nœuds n'ayant pas le contrôle du bus. On ajoute un événement supplémentaire, que nous nommons *nothing*, qui modélise le comportement de ces nœuds. La substitution *skip* est la substitution qui ne modifie pas l'état du système. Comme pour l'événement *take*, nous utilisons la forme ANY nd WHERE nous permettant de représenter en un seul événement le comportement de plusieurs nœuds. La garde $nd \notin wr$ assure que l'événement *nothing* ne peut pas se produire pour le nœud ayant le contrôle du bus. Ainsi, le seul événement pouvant se produire pour le nœud ayant le contrôle du bus est l'événement *free*, ce qui assure que le bus sera toujours libéré (sinon le nœud en question serait bloqué).

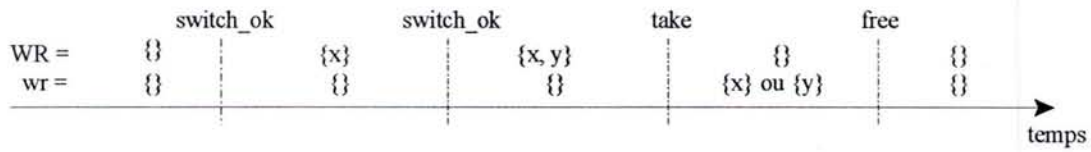
nothing =
ANYnd WHERE
$nd \in ND \wedge nd \notin wr$
THEN
skip
END

7.3 Phase de compétition

On prend maintenant en compte le fait que les nœuds passent par une phase de compétition pour obtenir le contrôle du bus. Nous ajoutons au modèle du système la variable WR qui est un ensemble contenant les nœuds souhaitant accéder au bus.

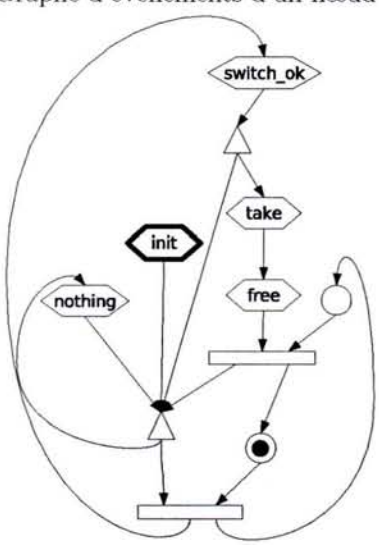
$$\boxed{WR \subseteq ND}$$

On ajoute également un événement au modèle, nommé *switch_ok* permettant à un nœud d'entrer en compétition, c'est-à-dire dans l'ensemble WR. Le nœud obtenant finalement le contrôle du bus (événement *take*) est un des nœuds de l'ensemble WR. On donne ci-dessous une illustration de l'évolution temporelle du système. Dans une première phase, certains nœuds entre en compétition (événement *switch_ok*). Dans l'illustration ci-dessous deux nœuds x et y entre en compétition. À un moment donné, l'un d'entre eux obtient le contrôle du bus et ensuite le libère.



La figure 7.2 donne le graphe d'événement d'un nœud du système.

FIG. 7.2 – Graphe d'événements d'un nœud du système



L'invariant stipule qu'aucun nœud ne peut être encore dans la phase de compétition lorsqu'un composant a pris le contrôle du bus : si l'ensemble des nœuds en compétition n'est pas vide alors le bus doit être libre.

$$\boxed{(WR \neq \varnothing \Rightarrow wr = \varnothing)}$$

La nouvelle variable WR est initialisée avec la valeur $\varnothing$, c'est-à-dire qu'aucun des nœuds n'est en compétition lorsque le système est initialisé.

$$\boxed{wr, WR := \varnothing\ ,\ \varnothing}$$

Nous raffinons l'événement *take* de façon à ce qu'il ne puisse se déclencher que lorsque que certains nœuds sont en demande pour le contrôle du bus (WR n'est pas vide). Nous sommes plus précis sur le choix du composant qui obtient le contrôle du bus en disant que ce nœud doit être un de ceux en compétition.

La garde $nd \in WR$ implique à la fois que l'ensemble des nœuds en compétition n'est pas vide et que le nœud choisi est l'un de ceux là. Remarquons que la partie $wr = \varnothing$ de la garde de l'événement *take* dans la machine abstraite précédente n'est plus présente dans ce raffinement. En effet, la condition $nd \in WR$ implique que WR n'est pas vide et donc, grâce à l'invariant $WR \neq \varnothing \Rightarrow wr = \varnothing$, on peut déduire que $wr = \varnothing$. La garde de l'événement raffiné est donc plus forte que la garde de l'événement abstrait, c'est-à-dire que la garde de l'événement abstrait est une conséquence logique de la garde de l'événement raffiné. Ceci est une obligation pour que le nouvel événement *take* soit un raffinement correct de l'événement abstrait. Dans cet événement on ajoute la substitution $WR := \varnothing$ qui remet à zéro l'ensemble des nœuds en compétition lorsqu'un nœud obtient le contrôle du bus. Ceci signifie que lorsqu'un nœud obtient le contrôle du bus, tous les autres composants doivent stopper la compétition.

take =
  ANY nd WHERE
    $nd \in WR$
  THEN
    $wr := \{nd\} \parallel$
    $WR := \varnothing$
  END

Le nouvel événement *switch_ok* modélise l'entrée d'un nœud dans la compétition pour obtenir le contrôle du bus. Cet événement peut se produire plusieurs fois avant que l'événement *take* ne se déclenche. Le fait qu'un seul nœud puisse être ajouté dans l'ensemble WR à chaque occurrence de l'événement *switch_ok* est un modèle de la concurrence entre les composants. Comme dans le cas de l'événement *take*, la forme ANY est une forme compacte de n événements, un par nœud, permettant à chacun d'entre eux d'entrer en compétition. Si on permettait à l'événement *switch_ok* de faire entrer plusieurs nœuds en compétition en même temps, cet événement ne pourrait plus être considéré comme représentant un ensemble de n événements, chacun étant rattaché à un nœud bien particulier.

switch_ok =
  ANY nd WHERE
    $wr = \varnothing \wedge$
    $nd \in ND \wedge nd \notin WR$
  THEN
    $WR := WR \cup \{nd\}$
  END

La forme ANY nd WHERE d'un événement exprime le fait que l'événement se déclenche pour le composant nd. Le choix du nœud nd est a priori non déterministe mais contraint par la garde. La garde peut se voir en deux parties : d'une part la condition sous laquelle l'événement peut se déclencher, et d'autre part la contrainte sur le choix du nœud pour lequel l'événement se déclenche.

Par exemple, l'événement *switch_ok* peut se déclencher lorsque le bus est libre ($wr = \varnothing$), c'est la condition sous laquelle l'événement peut se déclencher. La condition exprimant une contrainte sur le choix du composant est que le composant ne doit pas être déjà en compétition ($nd \in ND \wedge nd \notin WR$). Ainsi, cet événement peut se déclencher pour tout nœud respectant ces conditions, et non pas pour un unique nœud. En effet, après chaque occurrence de l'événement *switch_ok*, la garde reste satisfaisable et l'événement peut donc se produire de nouveau. Ce n'est par exemple pas le cas de l'événement *take* dont la garde devient fausse après la première occurrence.

7.4 Phase d'attente

Dans cette étape de raffinement, une phase d'attente est ajoutée au modèle. Chaque nœud doit attendre un certain temps (une fois le bus libre) avant d'entrer en compétition pour l'obtention du contrôle du bus. Cette phase d'attente modélise une forme de propriété spécifiée dans le standard [83, item 5.2.1]. Plus un nœud à un temps d'attente long moins sa priorité est importante car lorsque la

compétition commence, les nœuds n'ayant pas terminé leur phase d'attente ne doivent pas entrer dans la compétition et attendre que le bus redevienne libre.

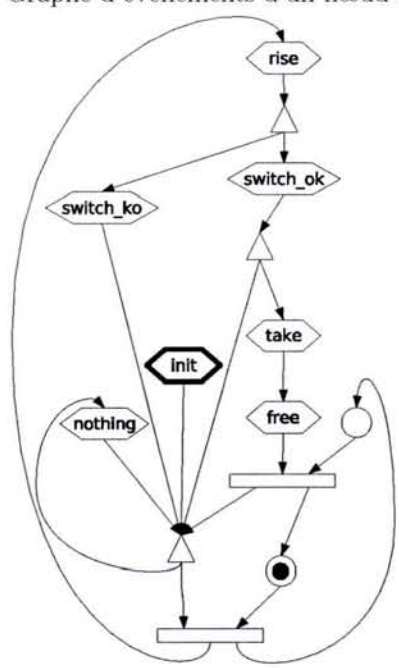
Pour modéliser cette phase d'attente, nous introduisons la nouvelle variable WA dans le modèle. Cette variable est un ensemble contenant l'ensemble des nœuds qui veulent avoir l'accès au bus et se trouvant en phase d'attente. Lorsqu'un nœud termine sa phase d'attente, il entre dans la phase de compétition en entrant dans l'ensemble WR introduit dans le raffinement précédent.

Une deuxième variable XR est ajoutée pour représenter l'ensemble des nœuds qui n'avaient pas terminé leur phase d'attente lorsque la phase de compétition a commencé. Les nœuds qui sont dans l'ensemble XR ont cessé d'attendre car au moins un autre composant a terminé sa phase d'attente avant eux, déclenchant ainsi la phase de compétition. La nécessité de cet ensemble XR nous est apparue pendant la preuve du raffinement : elle est nécessaire pour assurer que les nouveaux événements introduits dans ce raffinement ne puissent pas prendre le contrôle indéfiniment. Cette variable sert en fait à exprimer qu'un composant qui quitte la phase d'attente sans l'avoir terminée (la phase de compétition a donc commencé par ailleurs) ne peut pas entrer de nouveau dans la phase d'attente tant que la phase de compétition n'est pas terminée et le bus redevenu libre. Cette variable sera supprimée dans un futur raffinement car cette condition qu'elle permet d'exprimer pourra s'exprimer d'une autre manière lorsque nous aurons un modèle plus concret du système.

$$\boxed{\begin{array}{l} WA \subseteq ND \wedge \\ XR \subseteq ND \end{array}}$$

La figure 7.3 donne le graphe d'événement d'un nœud du système.

FIG. 7.3 – Graphe d'événements d'un nœud du système



7.4.1 Description Comportementale

A ce niveau de raffinement, l'évolution du système est exprimée en utilisant 5 événements. Les événements *switch_ok*, *take* and *free* étaient déjà présents dans le raffinement précédent. Nous ajoutons deux événements.

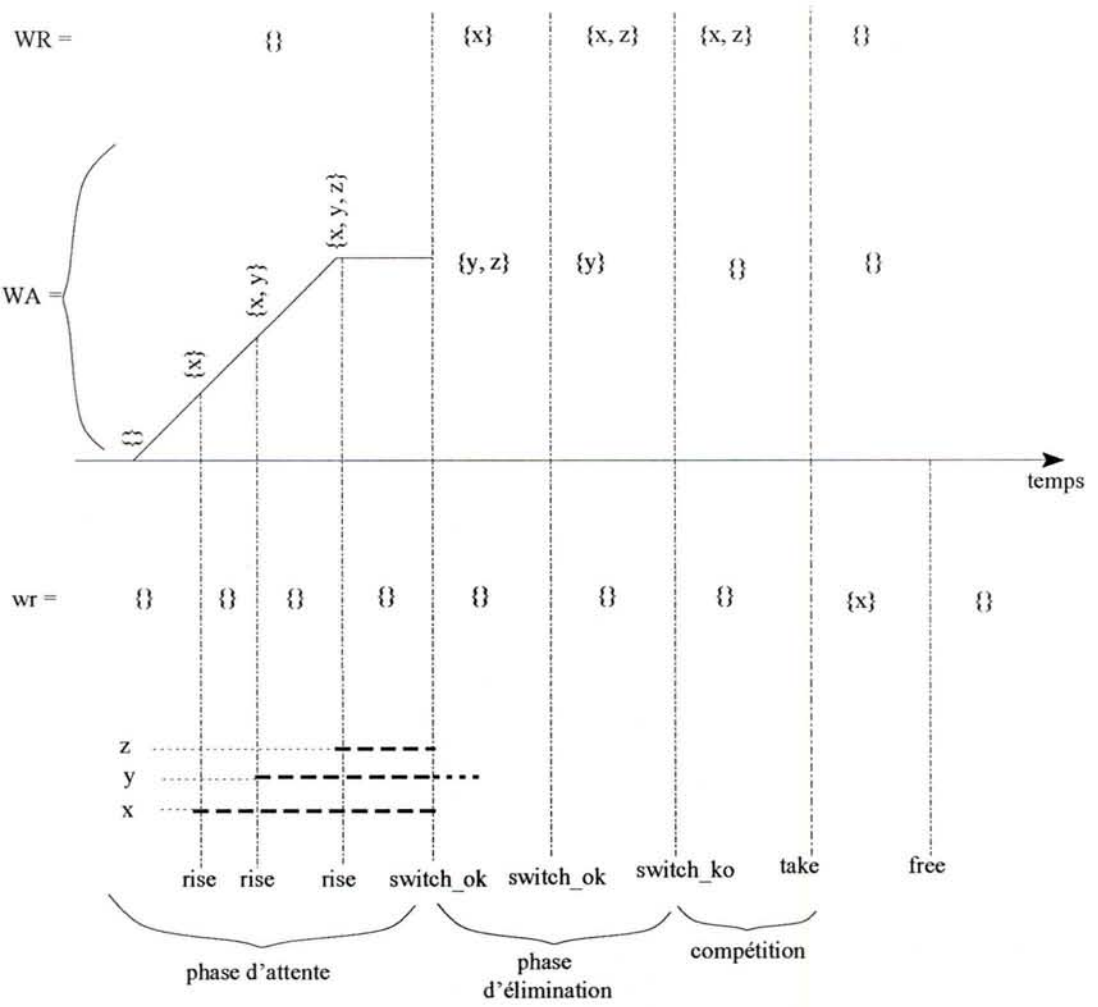
Le nouvel événement *rise* permet à un nœud d'entrer dans la phase d'attente (c'est-à-dire dans l'ensemble WA). Cet événement modélise l'augmentation du nombre de composants en attente. L'ensemble

WA peut grossir jusqu'à ce qu'au moins un nœud finisse sa phase d'attente. Il est possible que plusieurs nœuds finissent leur phase d'attente exactement au même moment, et entrent dans la phase de compétition, ce qui est modélisé par l'événement *switch_ok*.

Dès qu'un composant au moins finit sa phase d'attente, tous les composants qui étaient encore en train d'attendre sont éliminés. Le nouvel événement *switch_ko* modélise cette élimination. Ensuite, lorsqu'il n'y a plus de nœud en phase d'attente, l'événement *take* peut se déclencher pour choisir un nœud parmi ceux qui étaient entrés en compétition.

Notons que l'élimination des nœuds qui sont toujours en phase d'attente et la sélection de ceux qui ont terminé cette attente se déroule en même temps. C'est ce que nous avons appelé la phase d'élimination, qui succède à la phase d'attente et précède la phase de compétition.

Sur l'exemple ci-dessous, les trois nœuds (x, y et z) souhaitent obtenir le contrôle du bus pour pouvoir envoyer un message. Ils commencent tout d'abord par entrer en phase d'attente, ce qui est représenté par les trois occurrences de l'événement *rise*. Les nœuds x et z terminent leur phase d'attente au même moment, ils entrent alors en compétition (deux occurrences de l'événement *switch_ok*). Le nœud y est toujours en phase d'attente alors que la compétition commence, il doit donc s'arrêter (occurrence de l'événement *switch_ko*). Les nœuds x et z sont en compétition et un d'entre eux (par exemple x) finit par obtenir le contrôle du bus.



legend :

a ----- période d'attente de 5 unités

b ---------- période d'attente de 8 unités interrompues après 5 unités car le composant 'a' a fini son attente avant 'b'

L'invariant ci-dessous assure qu'il ne peut y avoir de nœud qui soit en phase d'attente lorsque le bus n'est pas libre. On a également la propriété qu'un nœud ne peut être à la fois en phase d'attente et en phase de compétition.

$$
\begin{array}{|l}
\hline
(\ WA \neq \varnothing \Rightarrow wr = \varnothing\ ) \wedge \\
(\ \forall\ nd.(nd \in ND \wedge nd \in WR \Rightarrow nd \notin WA)) \wedge \\
(XR \cap WA = \varnothing\ ) \wedge \\
(wr \cap XR = \varnothing\ ) \\
\hline
\end{array}
$$

7.4.2 Initialisation

Lorsque le système est initialisé, il n'y a encore aucun nœud en phase d'attente.

$$
\begin{array}{|l}
\hline
\text{INITIALISATION} \\
\quad wr, WR,\ XR,\ WA := \varnothing\ ,\ \varnothing\ ,\ \varnothing\ ,\ \varnothing \\
\hline
\end{array}
$$

7.4.3 Événements

Le nouvel événement *rise* modélise l'entrée d'un nœud en phase d'attente. Les nœuds qui souhaitent obtenir le contrôle du bus sont successivement ajoutés à l'ensemble WA des nœuds en attente. Ceci est un modèle abstrait des propriétés spécifiées dans [83]. Le temps d'attente n'est pas encore modélisé explicitement à ce niveau d'abstraction.

$$
\begin{array}{|l}
\hline
\textbf{rise} = \\
\quad \text{ANY } nd \text{ WHERE} \\
\qquad WR = \varnothing \wedge wr = \varnothing \wedge \\
\qquad nd \in ND \wedge \\
\qquad nd \notin WA \wedge nd \notin XR \\
\quad \text{THEN} \\
\qquad WA := WA \cup \{nd\} \\
\quad \text{END} \\
\hline
\end{array}
$$

L'événement *switch_ko* interrompt la phase d'attente de certains nœuds. Lorsque la phase d'attente est terminée (car certains nœuds ont terminés), l'événement *switch_ko* retire tous les nœuds qui sont encore dans l'ensemble WA et qui ne sont pas enlevés par *switch_ok*.

$$
\begin{array}{|l}
\hline
\textbf{switch\_ko} = \\
\quad \text{ANY} nd \text{ WHERE} \\
\qquad nd \in WA \\
\quad \text{THEN} \\
\qquad WA := WA \text{ - } \{nd\} \parallel \\
\qquad XR := XR \cup \{nd\} \\
\quad \text{END} \\
\hline
\end{array}
$$

Les événements *take* et *switch_ok* sont raffinés de façon à prendre compte la phase d'attente. L'événement *take* ne doit pouvoir se déclencher que lorsque la phase d'élimination est achevée ($WA = \varnothing$). L'événement *switch_ok* fait entrer des nœuds dans la phase de compétition, ces nœuds sont choisis uniquement dans l'ensemble WA, c'est-à-dire l'ensemble des nœuds en phase d'attente. A ce niveau d'abstraction, les temps d'attente ne sont pas explicites et les choix des nœuds qui doivent être éliminés ou pris dans la phase de compétition est non déterministe. Les deux événements *switch_ok* et *switch_ko* modélisent la phase d'élimination. Cette phase termine la phase d'attente et commence la phase de compétition.

La phase d'attente doit expirer au même moment pour tous les nœuds, elle expire dès lors qu'au moins un des nœuds termine sa période d'attente et tous les autres composants sont interrompus. La phase de compétition commence au même moment pour tous les nœuds qui ont complètement fini leurs phases d'attente (événement *switch_ok*). Tous les nœuds qui n'avaient pas complété leurs temps d'attente sont pris en charge par l'événement *switch_ko* au même moment.

On voit donc sur cet exemple que la concurrence entre les événements *switch_ko* et *switch_ok* est utilisée ici pour modéliser du parallélisme. Reprenons l'interprétation de la forme ANY d'un événement comme une forme contractée d'un ensemble de n événements en concurrence. Dans le cas de l'événement *take* nous avions déjà précisé qu'il s'agissait d'une véritable concurrence puisque dès qu'il s'est produit une fois, il ne peut plus se reproduire : l'événement *take* rend sa propre garde insatisfiable. Cela est différent pour les événements *switch_ko* et *switch_ok*. Ces deux événements peuvent se produire pour tous les nœuds de l'ensemble WA, en fait pour chaque composant, un seul des deux va se déclencher (le choix duquel est non déterministe à ce niveau d'abstraction). De plus, on remarquera la condition $WA = \varnothing$ dans la garde de l'événement *take*. Cette condition assure une certaine synchronisation entre les événements : il est impossible que *take* puisse se déclencher tant que WA n'est pas vide, c'est-à-dire tant que les événements *switch_ko* et *switch_ok* n'ont pas fini de faire le tri entre les nœuds qui doivent entrer en compétition et les nœuds qui doivent être interrompus.

take =
ANY nd WHERE
$nd \in WR \wedge WA = \varnothing$
THEN
$wr := \{nd\} \parallel$
$WR := \varnothing \parallel$
$XR := \varnothing$
END

switch_ok =
ANY nd WHERE
$nd \in WA$
THEN
$WR := WR \cup \{nd\} \parallel$
$WA := WA - \{nd\}$
END

La nouvelle variable XR est l'ensemble des nœuds qui ont été éliminés par l'événement *switch_ko*. La condition $nd \notin XR$ est ajoutée dans la garde de *rise* afin d'éviter qu'un composant qui vient d'interrompre sa phase d'attente ne rentre de nouveau dans WA. Ceci est nécessaire pour s'assurer qu'il n'y ait pas une boucle infinie *rise..switch_ko..rise* avec un composant qui ne ferait que rentrer et sortir de WA en permanence. Du point de vue de l'interprétation du système, cela signifie qu'un composant qui met fin à sa période d'attente doit soit entrer dans la compétition, soit attendre la prochaine phase d'attente après la libération du bus. La variable XR doit être réinitialisée avant la prochaine nouvelle phase d'attente. Ici, nous faisons cette réinitialisation à la fin de la phase de compétition, par l'événement *take*.

La variable XR sera retirée du modèle dans un futur raffinement car la condition $nd \notin XR$ pourra se déduire du contexte.

L'événement *nothing* est également raffiné. Comme nous l'avions déjà précisé, cet événement modélise le comportement des nœuds dont le comportement n'est pas modélisé par les autres événements. Dans ce raffinement nous avons ajouté des événements (*switch_ko* et *switch_ok*) modélisant le comportement des composants qui sont en phase d'attente (qui sont dans l'ensemble WA). Il faut donc retirer de la garde de l'événement *nothing* les nœuds qui sont dans l'ensemble WA. On notera qu'on ne retire pas les nœuds qui sont en compétition car un seul prendra le bus (événement *take*), et les autres ne feront rien (en tout cas à ce niveau d'abstraction).

nothing =
ANY nd WHERE
$nd \in ND \wedge nd \notin wr \wedge nd \notin WA$
THEN
skip
END

7.5 Choix du nœud vainqueur

Ce raffinement a pour but de spécifier de manière déterministe quel nœud, parmi ceux qui sont en compétition, obtiendra le bus. Nous n'entrons cependant pas encore dans les détails d'implantation du protocole tel qu'il est défini dans [83]. Nous définissons un ordre, nommé nb sur les nœuds ; cet ordre est défini par une fonction injective de l'ensemble des nœuds ND vers les nombres entiers. Nous spécifions que le choix du nœud qui obtient le bus est complètement déterminé par l'ordre nb.

L'ordre est simplement défini en associant un nombre entier à chaque nœud, c'est ce qui est fait par l'ordre nb, défini en B par une fonction constante. Deux nœuds distincts ont, par nb, deux nombres associés distincts : nb est une fonction injective. La notation $\rightarrowtail$ dénote une fonction injective en B.

CONSTANTS
nb
PROPERTIES
$nb \in ND \rightarrowtail \mathbb{N}$

Ce raffinement ne raffine que l'événement *take*. Jusqu'ici celui-ci choisissait de manière non déterministe le nœud auquel il donnait le contrôle du bus. Dans ce raffinement, l'événement *take* spécifie que le nœud qui obtient finalement le bus est celui, parmi les nœuds en compétition (c'est-à-dire les nœuds de l'ensemble WR) qui a le plus petit nombre associé par la fonction nb.

Ainsi, le nœud qui obtient le contrôle du bus est complètement déterminé par la donnée de l'ensemble WR des nœuds en compétition et la fonction constante nb.

Dans le code de l'événement *take* ci-dessous, l'expression $nb[WR]$ est l'image de l'ensemble WR par la fonction nb, c'est-à-dire l'ensemble des nombres associés aux nœuds qui sont dans l'ensemble WR. L'expression $\min(nb[WR])$ correspond donc au plus petit des nombres parmi ceux associés aux nœuds en compétition. Comme WR n'est pas vide (condition dans la garde de l'événement *take*) et que nb est une fonction totale, nous savons que $nb[WR]$ n'est pas vide et que par conséquent $\min(nb[WR])$ est bien défini. La fonction nb^{-1} est la relation inverse de nb : la fonction nb^{-1} associe à un nombre donné le nœud auquel il est rattaché. Pour que l'expression soit correcte, il faudra prouver que l'argument donné à nb^{-1} fasse bien parti de son domaine, c'est-à-dire du codomaine de nb. Ainsi, l'expression $nb^{-1}\ (\min(nb[WR]))$ désigne le nœud en compétition qui a le plus petit nombre associé par nb.

take =
WHEN
$WR \neq \varnothing \wedge WA = \varnothing$
THEN
$wr := \{nb^{-1}\ (\min(nb[WR]))\} \parallel$
$WR := \varnothing \parallel$
$XR := \varnothing$
END

Remarquons que dans le raffinement précédent, l'événement *take* utilisait la forme ANY de façon à pouvoir choisir un nœud de façon non déterministe. Ici nous n'avons plus besoin d'utiliser cette forme car nous calculons le nœud choisi de façon déterministe. Nous utilisons donc la forme WHEN de l'événement, et la garde $nd \in WR$ est remplacée par $WR \neq \varnothing$. Ce remplacement implique qu'une obligation de preuve sera générée pour assurer que le nœud calculé appartient bien à l'ensemble WR.

7.6 Résolution itérative de la compétition

Nous raffinons maintenant la phase de compétition. Dans le raffinement précédent, l'événement *take* calculait d'un seul coup le nœud qui obtient finalement le contrôle du bus. Nous montrons maintenant comment le choix du nœud vainqueur de la compétition peut se faire pas à pas, en éliminant des nœuds de la compétition au fur et à mesure. Cette élimination se fait itérativement jusqu'à ce qu'il n'en reste plus qu'un. Celui qui reste est celui qui était spécifié par l'événement *take* dans le raffinement précédent.

7.6.1 Nouvelle variable

Pour modéliser la modification du nombre de nœuds en compétition au fil du temps, nous utilisons une nouvelle variable CP qui représente l'ensemble des nœuds en compétition à un moment donné. Cet ensemble décroît au fur et à mesure que la compétition avance jusqu'à ce que CP ne contienne plus qu'un seul nœud. Il ne faut pas confondre ce nouvel ensemble CP avec l'ensemble WR qui représente l'ensemble

des nœuds en début de compétition. En début de compétition, l'ensemble CP est égal à WR et en fin de compétition, l'ensemble CP ne contient plus qu'un seul nœud, celui qui a gagné le contrôle du bus.

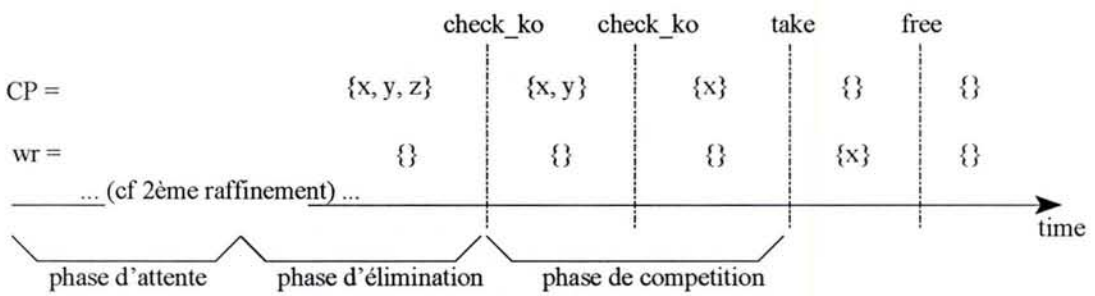
En fait, la variable CP est un raffinement de la variable WR. On peut dire aussi que WR est une abstraction de CP quand on ne considère que les états où il est soit vide soit plein (en début de compétition). Ce raffinement de donnée est exprimé par l'invariant de collage donné ci-dessous. Le but d'un invariant de collage est de permettre de retrouver la valeur d'une variable abstraite supprimée du modèle (ici WR) en fonction des variables présentes dans le nouveau modèle. L'invariant de collage donné ici peut s'interpréter de cette façon : l'ensemble CP est un sous-ensemble de WR, et si aucun nœud n'utilise le bus ($wr = \varnothing$) et qu'il n'y a pas de nœud en compétition ($CP = \varnothing$) alors la compétition n'a pas encore commencé ($WR = \varnothing$). C'est-à-dire que s'il y a des nœuds en compétition en début de compétition ($WR \neq \varnothing$), alors il y a toujours au moins un nœud en compétition, jusqu'à ce que l'un d'entre eux prenne le bus.

$$\boxed{\begin{array}{l} CP \subseteq WR \wedge \\ (wr = \varnothing \Rightarrow (CP = \varnothing \Rightarrow WR = \varnothing)) \end{array}}$$

7.6.2 Évolution du système

Un nouvel événement, *check_ko*, est introduit dans le modèle. Son rôle est de considérer deux événements et de comparer les nombres associés. Le composant qui a le nombre associé le plus grand est éliminé de la compétition. L'événement *check_ko* se produit jusqu'à ce qu'il ne reste plus qu'un seul nœud en compétition. Au bout du compte, puisqu'à chaque pas on a éliminé le nœud qui a le plus grand nombre associé, le nœud qui reste seul en compétition à la fin est celui qui a le plus petit nombre associé.

Un exemple d'évolution de ce système est représenté sur la figure ci-dessous. La figure 7.4 donne le graphe d'événement d'un nœud du système.



L'invariant nous assure que tous les nœuds qui étaient présents en début de compétition ($nd \in WR$) et qui ont été éliminés ($nd \notin CP$), sont associés à un nombre ($nb(nd)$) plus grand que le plus petit nombre associé à l'un des nœuds en compétition ($\min(nb[WR])$).

C'est-à-dire que le nœud qui est associé au plus petit nombre reste toujours en compétition. Ainsi, lorsqu'il ne reste plus qu'un nœud dans la compétition, c'est celui qui est associé au plus petit nombre. Cet invariant est primordial pour pouvoir prouver la correction du raffinement de l'événement *take*.

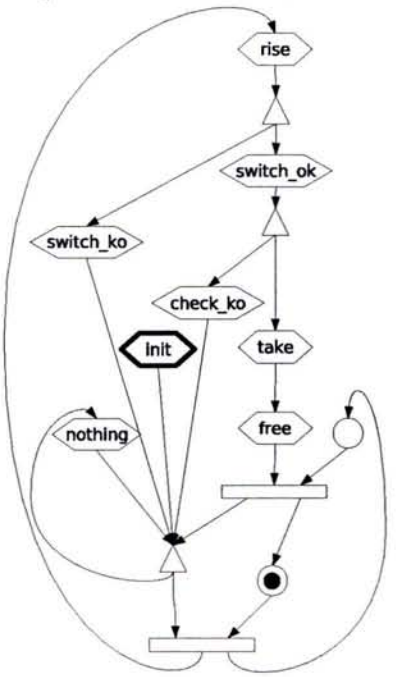
$$\boxed{\begin{array}{l} \forall\, nd.(nd \in WR \wedge nd \notin CP \Rightarrow nb(nd) > \min(nb[WR])) \quad \wedge \\ WA \cap CP = \varnothing \end{array}}$$

7.6.3 Initialisation

L'initialisation est modifiée pour tenir compte du remplacement de la variable WR par la variable CP.

$$\boxed{\begin{array}{l} \text{INITIALISATION} \\ \quad wr\,,\ WA,\ XR,\ CP := \varnothing\,,\ \varnothing\,,\ \varnothing\,,\ \varnothing \end{array}}$$

FIG. 7.4 – Graphe d'événements d'un nœud du système



7.6.4 Événements

Le nouvel événement *check_ko* peut se déclencher lorsqu'il reste au moins deux nœuds distincts en compétition. Le nœud auquel est associé (par la fonction *nb*) le plus grand nombre est enlevé de l'ensemble *CP*. Dans la garde de l'événement, la condition $nb(nd2)>nb(nd1)$ permet d'assurer que les deux nœuds $nd1$ et $nd2$ sont distincts et que c'est $nd2$ qui doit être éliminé de la compétition.

check_ko =
  ANY *nd1*, *nd2* WHERE
    $nd1 \in CP \wedge$
    $nd2 \in CP \wedge$
    $nb(nd2)>nb(nd1)$
  THEN
    $CP := CP\text{-}\{nd2\}$
  END

Puisque *nb* est injective, chaque nœud est associé à un nombre unique. Ainsi, tant qu'il reste au moins deux nœuds en compétition (dans l'ensemble *CP*), l'événement *check_ko* peut se déclencher.

L'événement *check_ko* est le seul événement qui puisse retirer des nœuds de l'ensemble *CP* pendant la compétition. À chaque occurrence, il élimine un nœud et laisse un autre nœud dans la compétition. L'ensemble *CP* des nœuds en compétition n'est donc jamais vidé par l'événement *check_ko*, il reste toujours au moins un composant dans *CP*, jusqu'à ce qu'un nœud prenne le contrôle du bus (événement *take*).

Lorsque l'événement *check_ko* ne peut plus se déclencher, cela signifie qu'il ne reste plus qu'un seul nœud dans l'ensemble *CP*. L'invariant donné plus haut assure que ce nœud restant seul en fin de compétition est celui qui est associé au plus petit nombre par la fonction *nb*. L'événement *take* peut alors se déclencher de façon à donner le contrôle du bus au nœud restant. L'événement *take* n'a plus besoin de refaire le calcul pour savoir à quel nœud donner le contrôle du bus, il suffit de choisir l'unique nœud restant dans l'ensemble *CP*. Le nœud restant est le même que celui qui était spécificité dans le précédent raffinement par l'expression $nb^{-1}(\min(nb[WR]))$.

take =
ANY nd WHERE
$nd \in CP \wedge WA = \varnothing \wedge$
$CP = \{nd\}$
THEN
$wr := \{nd\} \parallel$
$CP := \varnothing \parallel$
$XR := \varnothing$
END

Les événements *switch_ok* et *rise* doivent être raffinés de façon à prendre en compte le remplacement de la variable WR par la variable CP. Aucune autre modification n'est nécessaire à ce niveau de raffinement.

rise = ANY nd WHERE
$CP = \varnothing \wedge wr = \varnothing \wedge$
$nd \in ND \wedge$
$nd \notin WA \wedge nd \notin XR$
THEN
$WA := WA \cup \{nd\}$
END

switch_ok =
ANY nd WHERE
$nd \in WA$
THEN
$CP := CP \cup \{nd\} \parallel$
$WA := WA - \{nd\}$
END

L'événement *nothing* est raffiné en ajoutant la condition $nd \notin CP$ dans la garde, de telle sorte qu'il ne puisse se produire pour un nœud se trouvant dans la compétition, car seuls les événements *check_ko* (pour éliminer un nœud) et *take* (pour attribuer le contrôle du bus) doivent gérer les nœuds en compétition. Ceci permet d'assurer que la compétition se terminera toujours car seuls les événements *check_ko* et *take* peuvent se produire pour les nœuds en compétition, l'événement *check_ko* faisant décroître constamment le nombre de nœuds en compétition et *take* mettant fin à la compétition.

nothing =
ANY nd WHERE
$nd \in ND \wedge$
$nd \notin wr \wedge nd \notin WA \wedge nd \notin CP$
THEN
skip
END

7.7 Phase d'attente concrétisée

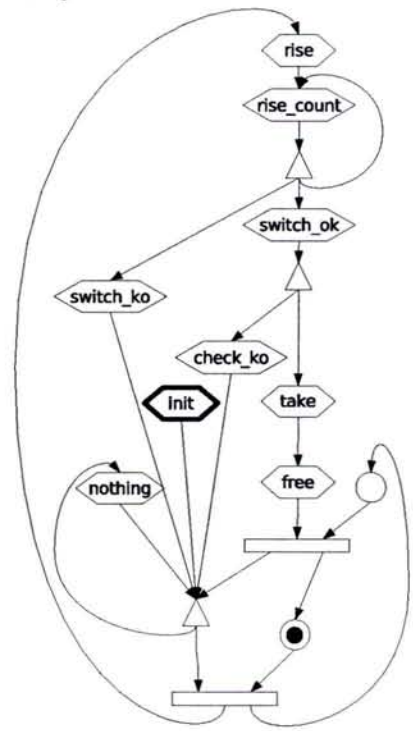
Après avoir décomposé la phase de compétition par des éliminations successives des composants, nous nous intéressons de nouveau à la phase d'attente en intégrant un modèle du temps d'attente des nœuds souhaitant obtenir le contrôle du bus. Les nœuds peuvent accéder à la phase de compétition lorsque leurs temps d'attente ont expiré. Si un nœud est toujours en train d'attendre alors qu'au moins un autre est arrivé à terme de son temps d'attente, la phase de compétition commence et le nœud encore en attente est éliminé. La figure 7.5 donne le graphe d'événement d'un nœud du système.

7.7.1 Nouvelle variable

Cette étape de raffinement consiste à raffiner la variable WA, représentant l'ensemble des nœuds en phase d'attente. Cette variable est raffinée par un tableau (une fonction en B) de compteurs AT, un pour chaque nœud en phase d'attente. Le lien entre la variable WA et la nouvelle variable AT est exprimé par l'invariant de collage ci-dessous. La valeur de l'ancienne variable WA peut donc se retrouver facilement en considérant le domaine de la fonction AT. Le signe $\nrightarrow$ dénote une fonction partielle en B. La fonction est définie pour les nœuds se trouvant en phase d'attente et pas pour les autres.

$AT \in ND \nrightarrow \mathbb{N} \wedge$
$dom(AT) = WA$

FIG. 7.5 – Graphe d'événements d'un nœud du système



7.7.2 Initialisation

Dans l'initialisation, on remplace simplement la variable WA par la variable AT. Le fait d'initialiser la fonction AT à l'ensemble vide signifie que son domaine est vide et que la fonction n'est définie sur aucun nœud à l'initialisation du système.

INITIALISATION
$wr,\ AT,\ XR,\ CP := \varnothing\ ,\ \varnothing\ ,\ \varnothing\ ,\ \varnothing$

7.7.3 Événements

L'événement *rise* est raffiné de telle sorte qu'il est chargé d'associer un compteur à chaque nœud entrant en phase d'attente. La façon exacte dont la valeur initiale (t) du compteur est choisie ne fait pas partie de cette étude de cas. Nous nous contentons d'assigner une valeur positive (choisie de façon non déterministe) au compteur. Nous verrons plus tard que ceci est une façon de spécifier t comme une entrée du circuit. Le calcul de la valeur exacte de t est donc laissé à l'environnement du circuit. La substitution $AT := AT \Leftarrow \{nd \mapsto t\}$ signifie qu'un compteur est créé pour le nœud nd et qu'il lui est affecté la valeur t, valeur choisie de façon non déterministe grâce à la forme ANY de l'événement.

rise =
ANY $nd,\ t$ WHERE
$CP = \varnothing \wedge wr = \varnothing \wedge$
$nd \in ND \wedge$
$nd \notin dom(AT) \wedge t \in \mathbb{N}_1 \wedge$
$nd \notin XR$
THEN
$AT := AT \Leftarrow \{nd \mapsto t\}$
END

Puisque nous avons introduit un compteur pour modéliser la phase d'attente précédent la compétition, nous avons besoin d'un événement qui soit chargé de décrémenter le compteur. C'est le rôle du nouvel événement *rise_count*. Cet événement modélise de façon très abstraite le passage du temps. La substitution $AT := AT \Leftarrow \{nd \mapsto AT(nd)\text{-}1\}$ signifie que la valeur du compteur du nœud nd est décrémentée d'une unité. On remarquera la condition $CP = \varnothing$ dans la garde de l'événement. Ceci permet d'interdire à cet événement de se déclencher lorsque la compétition a commencé. Dans ce cas, les seuls événements à pouvoir se déclencher sont *switch_ko* (le nœud est éliminé) et *switch_ok* (le nœud entre dans la compétition).

rise_count = ANY nd WHERE
$CP = \varnothing \wedge wr = \varnothing \wedge$
$nd \in dom(AT) \wedge$
$AT(nd) \neq 0$
THEN
$AT := AT \Leftarrow \{nd \mapsto AT(nd)\text{-}1\}$
END

L'événement *rise_count* est le seul nouvel événement dans ce raffinement. Les autres événements *take*, *switch_ko*, *switch_ok* et *nothing* doivent cependant être raffinés de façon à prendre en compte le remplacement de la variable *WA* par la variable *AT*. Le remplacement se fait de façon simple en remplaçant *WA* par *dom(AT)* dans toutes les expressions et la substitution $WA := WA - \{nd\}$ doit être remplacée par $AT := \{nd\} \lhd AT$. L'opérateur $\lhd$ est une soustraction sur le domaine d'une fonction.

De plus, nous pouvons maintenant être plus précis dans les gardes des événements *switch_ko* et *switch_ok*. Nous augmentons les gardes en disant que les événements qui sont éliminés sont ceux dont le temps d'attente restant n'est pas nul ($AT(nd) \neq 0$) et que ceux qui entrent en compétition sont ceux dont le temps d'attente a expiré ($AT(nd)$=0).

switch_ko =
ANY nd WHERE
$nd \in dom(AT) \wedge AT(nd) \neq 0$
THEN
$AT := \{nd\} \lhd AT \|$
$XR := XR \cup \{nd\}$
END

switch_ok =
ANY nd WHERE
$nd \in dom(AT) \wedge AT(nd)$=0
THEN
$CP := CP \cup \{nd\} \|$
$AT := \{nd\} \lhd AT$
END

take = ANY nd WHERE
$nd \in CP \wedge dom(AT) = \varnothing \wedge$
CP=$\{nd\}$
THEN
$wr := \{nd\} \|$
$CP := \varnothing \|$
$XR := \varnothing$
END

nothing =
ANY nd WHERE
$nd \in ND \wedge$
$nd \notin wr \wedge nd \notin dom(AT) \wedge nd \notin CP$
THEN
skip
END

Le modèle abstrait de la phase d'attente est maintenant complet. Un nœud $nd1$ souhaitant obtenir le contrôle du bus commence d'abord par entrer en phase d'attente. Ensuite, si un autre composant $nd2$ complète sa phase d'attente avant $nd1$, alors $nd1$ est éliminé. Si $nd1$ arrive à finir sa phase d'attente (c'est-à-dire $AT(nd1) = 0$), il entre dans la phase d'élimination (les autres composants n'ayant pas terminé leur phase d'attente sont éliminés), et ensuite dans la phase de compétition.

7.8 Vers une modélisation implantable

Les modèles précédents ont décrit de manière abstraite les différentes phases du protocole permettant à un nœud d'obtenir le contrôle du bus. Les modèles suivants montrent comment raffiner ce modèle abstrait

de façon à obtenir un modèle synthétisable du contrôleur de bus permettant à un nœud d'envoyer un message sur le bus.

Cette section présente la méthode utilisée pour modéliser l'aspect cyclique du circuit physique et comment transformer le modèle abstrait en modèle plus concret, où chaque nœud est modélisé de façon indépendante.

7.8.1 Indépendance des nœuds

Nous rappelons que le protocole est divisé en cinq phases :

- une phase d'attente, durant laquelle chaque nœud souhaitant obtenir le contrôle du bus doit attendre un certain temps,
- une phase d'élimination qui se déclenche lorsque l'un des nœuds termine sa phase d'attente et qui consiste à éliminer les nœuds qui n'ont pas complété leurs temps d'attente,
- une phase de compétition éliminant des nœuds les uns après les autres jusqu'à n'en obtenir plus qu'un qui obtiendra le contrôle du bus,
- une phase de message, pendant laquelle le nœud qui a obtenu le contrôle du bus peut envoyer son message,
- une phase de latence, suivant la phase de message, pendant laquelle les nœuds attendent de détecter que le bus soit libre.

Ces cinq phases ont été modélisées d'une façon très abstraite par les modèles précédents. Les phases du message et de latence n'ont pas été modélisées de façon explicite dans les modèles abstraits donnés précédemment. Elles s'enchaînent en commençant avec l'événement *take* qui commence l'envoi du message et se termine par l'événement *free* qui libère le bus. Les modèles concrets qui seront obtenus dans les raffinements suivants permettront d'exprimer ces phases de façon explicite.

Si on observe les événements des modèles abstraits donnés jusqu'ici, on peut voir que les états internes des différents nœuds ne sont pas séparés. Nous avons en fait utilisé jusque là les variables représentant l'état du système complet contenant tous les nœuds. Par exemple, la variable CP exprime l'ensemble des composants se trouvant en compétition, mais il n'y pas de variable d'état interne à chaque composant lui indiquant s'il se situe ou non dans la compétition. Pour savoir s'il est en compétition, un nœud nd doit faire le test $nd \in CP$ c'est-à-dire consulter l'état global du système. Un modèle sera considéré comme à un niveau implantable lorsqu'il ne pourra accéder qu'à ses propres variables d'état internes et ses entrées. C'est pourquoi notre but est d'obtenir un modèle dans lequel tous les événements suivront le schéma ci-dessous, où $GUARD(nd)$ et S (la substitution) utilisent uniquement des variables auxquelles le nœud nd a accès. Un travail similaire a été fait par Abrial, Cansell et Méry dans [8]

event_name =
  ANY nd WHERE
    $nd \in ND \wedge$
    $GUARD(nd)$
  THEN
    S
  END

Évidemment, le modèle du système doit tenir compte de l'évolution concurrente des nœuds. La concurrence est modélisée en utilisant la forme des événements ANY *nd* WHERE déjà donnée ci-dessus. On peut diviser la garde en deux parties. Une première partie décrit la condition sous laquelle l'événement peut se déclencher. La deuxième partie définit quels sont les nœuds qui sont concernés par cet événement. Prenons l'exemple de la garde de l'événement *rise_count*, introduit dans la section précédente. La garde de cet événement est la suivante :

$$\boxed{\begin{array}{l} CP = \varnothing \wedge wr = \varnothing \wedge \\ \text{- - - - - - - - - - - - -} \\ nd \in dom(AT) \wedge \\ AT(nd) \neq 0 \end{array}}$$

La première partie de la garde exprime que l'événement peut se déclencher lorsqu'il n'y a aucun nœud dans la phase de compétition ($CP = \varnothing$) et que le bus est libre ($wr = \varnothing$). La deuxième partie spécifie que seulement sont concernés les nœuds dont le temps d'attente n'a pas expiré ($nd \in dom(AT) \wedge AT(nd) \neq 0$). Si l'événement se déclenche, un de ces nœuds sera choisi de façon non déterministe.

7.8.2 Modèle du cycle

Le principe de modélisation que nous avons choisi est d'obtenir un modèle événementiel cyclique (cf. section 4.2). C'est-à-dire qu'à chaque cycle, le comportement de chaque composant du système doit être modélisé. Si on souhaite que le modèle soit déterministe (ce qui doit être le cas pour un modèle implantable), il ne doit pas y a voir de choix quant à l'événement qui doit se déclencher pour un composant donné. Autrement dit, à chaque instant, et pour un composant donné du système, un et un seul événement doit avoir sa garde vraie.

Par ailleurs, lors d'un cycle donné, le comportement d'un composant peut ne pas pouvoir être modélisé par un seul événement. En effet, il peut être nécessaire de recourir à plusieurs événements pour la modélisation du comportement d'un composant dans un même cycle, en particulier lorsqu'il y a des communications avec les autres composants du système (cf. section 5.2). Ces multiples événements modélisant le comportement d'un même composant lors d'un même cycle doivent être ordonnancés, d'une façon ou d'une autre, de manière déterministe, de façon à ne pas être en contradiction avec le fait qu'à chaque instant, et pour chaque composant, un seul événement doit avoir sa garde vraie.

Dans l'étude de cas présentée ici, chaque nœud doit faire deux choses à chaque cycle :

- Un nœud doit d'abord écrire une valeur sur le bus, il écrit la valeur '1' (valeur récessive) sur le bus lorsque le protocole ne prévoit pas d'autre valeur (un nœud envoie toujours une valeur sur le bus, même lorsqu'il n'est pas un transmetteur).
- Une fois le bus stabilisé, c'est-à-dire que tous les nœuds du système ont envoyé une valeur sur le bus et que la valeur du bus est physiquement stable, chaque nœud doit lire la valeur présente sur le bus.

Le modèle choisi pour modéliser ce comportement est de diviser le cycle en deux périodes. Une première période d'*écriture* dans laquelle tous les nœuds envoient une valeur sur le bus et une deuxième période de *lecture*, dans laquelle les nœuds lisent la valeur du bus, qui doit donc être stable. Dans ce modèle, la stabilité du bus pendant la deuxième période est assurée par le fait que les nœuds ne peuvent modifier la valeur portée par le bus seulement pendant la première période. De la même manière, la valeur du bus ne doit pas être utilisée pendant la première période car elle ne peut pas être considérée comme stable.

Pour modéliser ce cycle en deux périodes, nous utilisons deux variables supplémentaires (*WRI* et *VER*) pour ordonnancer les deux périodes, et un événement supplémentaire (*env*). Ces deux variables supplémentaires servent à ordonnancer les deux périodes du cycle, et l'événement *env* modélise le passage d'un cycle à l'autre, c'est-à-dire de la deuxième période d'un cycle à la première période du cycle suivant.

$$\boxed{\begin{array}{l} VER \subseteq ND \wedge \\ WRI \subseteq (ND\text{-}VER) \wedge \\ VER \subseteq (ND\text{-}WRI) \end{array}}$$

La variable *WRI* est un ensemble contenant tous les nœuds se trouvant dans le période d'écriture (première période), et la variable *VER* est un ensemble contenant les nœuds ayant envoyé une valeur sur le bus (qui ont donc fini la première période) et qui sont prêts à commencer la période d'écriture (deuxième période). Dans la première période, lorsqu'un nœud envoie une valeur sur le bus, il se retire de l'ensemble *WRI* et entre dans l'ensemble *VER*. Tant que l'ensemble *WRI* n'est pas vide, le système est encore dans la première période du cycle. Une fois que l'ensemble *WRI* est vide, la période de lecture commence.

L'événement *env* est ajouté au modèle pour modéliser l'environnement. Il est également chargé, en manipulant les variables WRI et VER, d'effectuer le passage d'un cycle au suivant. Ceci n'est pas illogique puisqu'en pratique, le cycle est gouverné par un signal d'horloge qui est une entrée du circuit physique. Nous donnons ci-dessous un modèle de l'événement *env* , la substitution S est la substitution modélisant l'environnement, chargée de modifier les variables représentant les entrées par exemple. On peut dire que l'événement *env* "initialise" le cycle.

env =
  WHEN
    $VER = \varnothing \wedge WRI = \varnothing$
  THEN
    $S \parallel$
    $WRI := ND$
  END

Les gardes et les substitutions des événements modélisant les nœuds doivent être augmentées de façon à prendre en compte l'ordonnancement en deux périodes.

En début de cycle, tous les nœuds sont dans la période d'écriture ($WRI := ND$ dans l'événement *env*). Le comportement de chaque composant doit être modélisé dans cette période. C'est-à-dire qu'un événement doit se déclencher pour chaque nœud. Lorsque cet événement se déclenche, outre la fonctionnalité du nœud modélisé, il retire le nœud de la période d'écriture ($WRI := WRI - \{nd\}$) et l'insère dans l'ensemble des nœuds prêts pour la période de lecture ($VER := VER \cup \{nd\}$). Les événements de la période d'écriture doivent suivre le schéma suivant, dans lequel la substitution S exprime la fonctionnalité du nœud concerné, et $GUARD(nd)$ le reste de la garde de l'événement.

Event_write =
  ANY *nd* WHERE
    $nd \in WRI$
    $GUARD(nd)$
  THEN
    $S \parallel$
    $VER := VER \cup \{nd\} \parallel$
    $WRI := WRI - \{nd\}$
  END

Lorsque tous les nœuds du système ont terminé la période d'écriture du cycle ($WRI = \varnothing$), la période de lecture du cycle commence. De la même façon que pour la période d'écriture , le comportement de chaque nœud doit être modélisé, c'est-à-dire qu'un événement doit se déclencher pour chacun des nœuds du système. Cet événement retire le nœud de l'ensemble des nœuds de la période de lecture ($VER := VER - \{nd\}$). Lorsque la période de lecture est terminée ($VER = \varnothing \wedge WRI = \varnothing$), le cycle complet est terminé et un nouveau cycle peut alors commencer (grâce à l'événement *env*). Les événements de la période de lecture doivent suivre le schéma ci-dessous.

Event_read =
  ANY *nd* WHERE
    $nd \in VER \wedge WRI = \varnothing \wedge$
    $GUARD(nd)$
  THEN
    $S \parallel$
    $VER := VER - \{nd\}$
  END

Les variables supplémentaires WRI et VER sont toutes les deux initialisées à l'ensemble vide. Ainsi, lorsque le système est initialisé, c'est l'événement *env* qui se déclenche nécessairement le premier. Ensuite la période de lecture du premier cycle commence.

7.8.3 Modèle du bus

L'envoi effectif d'une valeur sur le bus est une partie analogique du système [83, page 8]. Un composant analogique ne peut pas être représenté tel quel en B. On doit modéliser le bus en correspondance avec la façon dont nous avons modélisé l'aspect cyclique du système. Nous avions déjà pris en considération les problèmes de stabilité du bus en créant deux périodes dans le cycle. Une première période d'écriture durant laquelle le bus est instable car chaque nœud envoie une valeur sur le bus, et une période de lecture durant laquelle le bus est stable. Voyons maintenant comment modéliser en B l'envoi d'une valeur par un nœud sur le bus.

Par définition du protocole, la valeur '0' domine le bus, c'est-à-dire que si au moins un des nœuds envoie la valeur '0' alors la valeur du bus est nécessairement '0'. On peut modéliser cela arithmétiquement en disant que la valeur portée par le bus est le minimum des valeurs envoyées par les nœuds. L'intérêt de cette modélisation est qu'elle peut se faire itérativement, ce qui est indispensable puisque le modèle du cycle, en particulier la période d'écriture, est une concurrence entre les nœuds : dans le modèle, les nœuds envoient leurs valeurs les uns après les autres. La modélisation peut se faire de manière itérative de la façon suivante : lorsqu'un nœud envoie la valeur *output* sur le bus, la nouvelle valeur du bus peut se calculer en prenant le minimum entre *output* et la valeur courante du bus. Lorsque chaque nœud aura envoyé sa valeur, c'est-à-dire quand la période d'écriture sera terminée, la valeur du bus sera le minimum de toutes les valeurs envoyées, à condition que l'état du bus soit initialisé à la valeur '1' (valeur récessive) avant le commencement de la période d'écriture. Cette initialisation sera donc faite par l'événement *env*. Pour chaque nœud, l'envoi d'une valeur *output* sur le bus se fera donc par la substitution suivante :

$$bus := min(bus, output)$$

Nous modéliserons les deux valeurs *physiques* '0' et '1' respectivement par les entiers 0 et 1. La valeur portée par le bus est modélisée par une variable *bus* de type 0..1.

7.9 Modèle cyclique du système

Dans ce raffinement, nous modélisons l'aspect cyclique comme cela a été expliqué dans la section précédente. Nous raffinons également la relation d'ordre *nb* entre les nœuds du système. Chaque nœud est identifié par un caractère (un vecteur de bits), appelé *MID* (Message IDentification number). Le *MID* est différent pour chaque nœud et la relation d'ordre entre les nœuds qui était modélisée par *nb* se retrouve maintenant dans la relation d'ordre entre les *MIDs*. La relation d'ordre entre les *MID* est la relation d'ordre "inférieur à" sur les nombres entiers représentés par les vecteurs de bits que sont les *MIDs*. Les bits sont représentés par les deux nombres entiers 0 et 1.

Le bus et les messages qui sont envoyés dessus sont maintenant modélisés explicitement dans le modèle. Le bus possède l'une des deux valeurs 0 ou 1. Chaque nœud du système peut envoyer des valeurs sur le bus. Le bus est dit *dominé* par 0, ce qui signifie que si l'un des nœuds écrit 0 sur le bus, la valeur portée par le bus est nécessairement 0. Le bus porte la valeur 1 uniquement si tous les nœuds envoient 1 sur le bus. Lorsqu'un nœud envoie la valeur 1 sur le bus mais qu'il observe que la valeur portée par le bus est 0 (c'est-à-dire qu'un autre nœud a envoyé le bit 0 sur le bus), il détecte une *collision*. Cela ne doit normalement se produire que pendant la phase de compétition.

Introduction

Ce raffinement se positionne au niveau bit de la description du protocole. Nous modélisons en particulier la façon concrète dont la phase de compétition élimine les nœuds un par un jusqu'à ce qu'il n'en reste plus qu'un seul. Pour donner une première impression de ce fonctionnement, nous donnons un petit exemple.

Pendant la phase de compétition,, tous les nœuds envoient leurs *MIDs* bit par bit. Faisons l'hypothèse que seulement deux nœuds `nd1` et `nd2` finissent leurs phases d'attente au même moment, et commencent ainsi tous les deux à envoyer leurs *MIDs* sur le bus. Puisque les *MIDs* de deux nœuds sont nécessairement différents, une collision se produira inévitablement pendant que les bits des *MIDs* sont envoyés les uns

après les autres. Comme 0 domine le bus (cf. *domination de '0'* in section 7.1.1.0 page 100), le nœud qui envoie un 0 le premier pendant que l'autre envoie 1 gagne la compétition. Au même moment, l'autre nœud, qui envoie 1 sur le bus, observe 0 sur le bus alors qu'il s'attendait à observer 1 (sa valeur). Le second nœud détecte la collision et quitte la compétition, le premier nœud peut continuer à envoyer des valeurs sur le bus car il ne détecte aucune collision. Ceci est le modèle que nous faisons du protocole. Comme nous l'avons discuté en début de chapitre, une autre interprétation pourrait être faite de la description du protocole. Dans cette deuxième interprétation, le deuxième nœud ne détecte pas de collision immédiatement et continue à envoyer des bits sur le bus. Il peut alors arriver que tous les nœuds quittent la compétition et que finalement aucun ne gagne l'accès au bus. Ce cas est montré sur la deuxième illustration ci dessous.

Voici un exemple de compétition telle que nous le modélisons :

```
( ---- = phase d'attente )

nd1       ------ 0 1 0 q1 q2 ...     --> le noeud nd1 peut continuer q1 q2 ...
nd2         ---- 0 1 1               --> le noeud nd2 s'arrête

bus   *latence*  0 1 0 q1 q2 ...
```

Le cas suivant peut apparaître dans la deuxième interprétation du standard, mais il est impossible dans notre modèle. Le premier nœud détecte une collision au quatrième bit, le deuxième nœud détecte une collision au troisième bit mais ne s'arrête pas immédiatement.

```
nd1       ------ 0 1 0 1           --> le noeud nd1 s'arrête
nd2         ---- 0 1 1 0           --> le noeud nd2 s'arrête

bus   *latence*  0 1 0 0     --> les deux noeuds s'arrêtent
```

Constantes

Nous définissons d'abord la macro BIT qui correspond au sous-ensemble des entiers $\{0, 1\}$ et nous définissons cinq constantes :

- *CHAR* est l'ensemble des caractères. Un caractère (section 7.1.1.0, page 100) est un vecteur de BITs. Le bit de départ a toujours la valeur 0 et le bit d'arrêt la valeur 1.
- *MAX_BIT_CHAR* est l'index du bit d'arrêt d'un caractère. La valeur n'est pas ici déterminée. Dans le standard, la valeur est de 9, mais nous laissons la valeur indéterminée de façon à laisser le modèle générique par rapport à la valeur de cette constante.
- *MID* est la fonction qui associe à chaque nœud son MID.
- *MESSAGES* est l'ensemble des messages admissibles. Cet ensemble n'est pas vide ($MESSAGES \neq \varnothing$) et un message est un vecteur fini de caractères ($MESSAGES \subseteq \mathbb{N} \rightarrow\!\!\!\!\!\!+ \; CHAR \wedge \forall \; msg.(msg \in MESSAGES \Rightarrow dom(msg){=}0 \; .. \; max(dom(msg))))$.
- *COUNT_FREE* est un nombre entier tel que le bus est considéré comme libre après *COUNT_FREE* 1 consécutifs sur le bus. En effet, puisque tout nœud n'écrit que des caractères (de longueur *MAX_BIT_CHAR*+1)) sur le bus, et puisque le premier bit d'un caractère est toujours 0, on peut dire que "aucun nœud n'est en train d'écrire sur le bus" est impliqué par "le bus a porté la valeur 1 depuis au moins *MAX_BIT_CHAR*+1 cycle". On doit donc poser la valeur de *COUNT_FREE* telle que *COUNT_FREE* - *MAX_BIT_CHAR* > 1. Là encore le modèle est générique par rapport à *COUNT_FREE* puisque d'une part cette constante dépend de *MAX_BIT_CHAR* qui est générique, et on pose seulement la condition > 1 et non pas = 2. Ainsi, une fois la valeur de *MAX_BIT_CHAR* fixée on garde encore une marge de manœuvre pour fixer la valeur de *COUNT_FREE*.

Par rapport au standard, *MAX_BIT_CHAR* aurait la valeur 9, et la plus petite valeur possible de *COUNT_FREE* serait 11.

DEFINITIONS
$BIT == \{0,1\}$
CONSTANTS
$MESSAGES$,
$CHAR$,
MID,
MAX_BIT_CHAR,
$COUNT\_FREE$
PROPERTIES
$MAX\_BIT\_CHAR \in \mathbb{N}_1 \wedge$
$CHAR = 0 \,..\, MAX\_BIT\_CHAR \rightarrow BIT \wedge$
$\forall\, cc.(cc \in CHAR \Rightarrow cc(0) = 0 \wedge cc(MAX\_BIT\_CHAR) = 1) \wedge$
$MID \in ND \rightarrowtail CHAR \wedge$
$COUNT\_FREE \in \mathbb{N} \quad \wedge$
$COUNT\_FREE - MAX\_BIT\_CHAR > 1 \wedge$
$MESSAGES \subseteq \mathbb{N} \nrightarrow CHAR \wedge$
$MESSAGES \neq \varnothing \wedge$
$\forall\, msg.(msg \in MESSAGES \Rightarrow dom(msg)=0 \,..\, max(dom(msg)))$

Nouvelles variables

Ce raffinement introduit beaucoup de nouvelles variables. Nous les présentons en les classant suivant la façon dont elles sont principalement utilisées.

La compétition La phase de compétition est raffinée en remplaçant la comparaison entre les nœuds utilisant la relation nb par une comparaison des MIDs. La comparaison est faite bit par bit (puisque les caractères sont envoyés bit par bit sur le bus). Pour cela, nous avons besoin d'une variable de boucle pos (une pour chaque nœud) pour parcourir le MID.

$$pos \in ND \rightarrow 0 \,..\, MAX\_BIT\_CHAR$$

Durant la compétition, chaque nœud nd écrit le bit de son MID ($MID(nd)$) qui est à la position $pos(nd)$: $MID(nd)(pos(nd))$

Le bus Le bus est simplement modélisé par une variable représentant la valeur qui est portée par le bus. Il s'agit d'une variable de type BIT, c'est-à-dire dans l'ensemble $\{0, 1\}$

$$bus \in BIT$$

L'état du bus dépend des valeurs envoyées par les nœuds. Si un nœud envoie 0 alors la valeur portée par le bus est 0, sinon la valeur portée par le bus est 1. Ainsi, on peut prouver l'invariant ci-dessous.

Le premier invariant assure que si la valeur portée par le bus est 0 ($bus = 0$), alors il existe un nœud qui a envoyé 0 sur le bus. L'ensemble $CP - WRI$ est l'ensemble des nœuds qui ont déjà envoyé une valeur sur le bus pendant le cycle courant. L'ensemble WRI est l'ensemble des nœuds qui doivent encore envoyer une valeur sur le bus. L'hypothèse $CP - WRI \neq \varnothing$ signifie que la période d'écriture a commencé (éventuellement elle peut être terminée et la période de lecture commencée).

Le second invariant ci-dessous assure que lorsqu'un composant détecte une collision ($(MID(nd))(pos(nd)) \neq bus$) alors le bus porte nécessairement la valeur 0. L'ensemble VER est l'ensemble des nœuds qui ont fini leurs phases d'attente et qui doivent encore faire leur période de lecture.

(BusInv)
$$\begin{array}{l}(bus{=}0 \wedge CP\text{-}WRI \neq \varnothing \Rightarrow \exists\, nd.(nd \in CP\text{-}WRI \wedge (MID(nd))(pos(nd))=0)) \wedge \\ (VER \neq \varnothing \Rightarrow (\ \forall\, nd.(nd \in CP\text{-}WRI \Rightarrow ((MID(nd))(pos(nd)) \neq bus \Rightarrow bus = 0\ ))))\end{array}$$

Pour savoir si le bus est libre, les nœuds utilisent un compteur fc.

$$fc \in ND \rightarrow \mathbb{N}$$

Ce compteur est réinitialisé à la valeur de la constante *COUNT_FREE* chaque fois que le bus porte la valeur 0. Lorsque le bus porte la valeur 1, le compteur *fc* est décrémenté, sauf si la valeur 0 était déjà atteinte (ceci permet d'avoir un codomaine fini pour la fonction $fc : 0..COUNT\_FREE$).

$$\text{(S)} \quad \boxed{fc(nd) := (1 - bus) \times COUNT\_FREE + bus \times max(\{fc(nd) - 1, 0\}).}$$

Comme tous les nœuds peuvent a priori vouloir envoyer des messages sur le bus, ils doivent savoir si le bus est libre ou non. Donc ils doivent maintenir à jour leurs compteurs *fc*. Ainsi, la substitution précédente (S) est ajoutée à chaque événement de la période de lecture. Notons que cette substitution peut en fait s'implanter de façon simple puisqu'elle est équivalente à IF $bus = 0$ THEN $fc(nd) := COUNT\_FREE$ ELSIF $fc(nd) = 0$ THEN $fc(nd) := 0$ ELSE $fc(nd) := fc(nd) - 1$. Dans le modèle, nous utilisons plutôt la forme (S) car elle génère moins d'obligations de preuve.

Écriture du message Une fois qu'un nœud gagne, après une phase de compétition, l'accès en écriture au bus, il commence à écrire son message. Le message du nœud *nd* est contenu dans une variable *MESSAGE(nd)*. Il est défini uniquement pour le nœud qui a l'accès au bus. C'est pourquoi la fonction *MESSAGE* a *wr* comme domaine, c'est le cas pour toutes les fonctions définies ici. Nous avons également besoin d'une variable permettant de parcourir le message : la variable *pos_message* est utilisée pour parcourir les caractères d'un message et la variable *pos_msg_char* est utilisée pour parcourir les bits dans un caractère. La variable *LONGMESSAGE* est la longueur du message (nombre de caractères sans le MID).

$$\begin{array}{l} LONGMESSAGE \in wr \rightarrow \mathbb{N}_1 \wedge \\ pos\_message \in wr \rightarrow \mathbb{N} \wedge \\ pos\_msg\_char \in wr \rightarrow 0 \mathinner{..} MAX\_BIT\_CHAR \wedge \\ MESSAGE \in wr \rightarrow MESSAGES \wedge \\ \forall\, nd.(nd \in wr \quad \Rightarrow \quad pos\_message(nd) \leq LONGMESSAGE(nd)) \end{array}$$

Initialisation Les deux variables *VER* et *WRI* sont initialisées à l'ensemble vide, comme cela est expliqué dans la section 7.8.2. Le bus est initialisé à 1 puisqu'aucun nœud n'est en train d'écrire sur le bus à l'initialisation. Les quatre variables *LONGMESSAGE*, *pos_message*, *MESSAGE*, *pos_msg_char* sont initialisées à l'ensemble vide puisqu'aucun nœud n'a encore l'accès au bus. Le compteur fc est initialisé à 0 pour chaque nœud : le bus est considéré comme libre par tous les nœuds.

$$\begin{array}{l} \text{INITIALISATION} \\ \quad wr\ ,\ AT,\ XR,\ CP,\ pos,\ VER,\ WRI,\ bus := \varnothing\ ,\ \varnothing\ ,\ \varnothing\ ,\ \varnothing\ ,ND \times \{0\},\ \varnothing\ ,\ \varnothing,\ 1\ \| \\ \quad LONGMESSAGE,\ pos\_message := \varnothing\ ,\ \varnothing\ \| \\ \quad MESSAGE,\ pos\_msg\_char,\ fc := \varnothing\ ,\ \varnothing\ ,\ ND \times \{0\} \end{array}$$

Environnement

Comme cela a été expliqué dans la section 7.8.2, il est nécessaire de modéliser l'environnement. L'événement *env* permet la transition d'un cycle au suivant. Le modèle itératif que nous avons choisi pour le bus (cf. 7.8.3) nécessite que celui-ci soit initialisé à la valeur 1 en début de cycle, ce qui est fait par l'événement *env*. La valeur 1 est la valeur maximale pour le bus, comme l'écriture sur le bus se fait par la substitution $bus := min(bus, valeur)$, la valeur 1 sera soit écrasée si un composant écrit 0 sur le bus, soit conservée si tous les composants écrivent 1 sur le bus.

$$\begin{array}{l} \textbf{env} = \\ \quad \text{WHEN} \\ \quad\quad VER = \varnothing \wedge WRI = \varnothing \\ \quad \text{THEN} \\ \quad\quad WRI := ND\ \| \\ \quad\quad bus := 1 \\ \quad \text{END} \end{array}$$

Période de lecture

Compétition Le principe de l'algorithme est que la variable de parcours du MID *pos* est incrémentée en même temps pour tous les nœuds qui sont en compétition. A chaque cycle, les bits des MIDs, des nœuds en compétition, se situant à la position *pos* sont comparés. S'ils sont tous identiques (ils envoient tous la même valeur sur le bus), alors il n'est pas possible de distinguer les compétiteurs dans le cycle courant. S'il existe un nœud *nd* tel que *MID(nd)(pos(nd))=0*, chaque compétiteur *ne* tel que *MID(ne)(pos(ne))=1* doit quitter la compétition car son MID est nécessairement plus grand que le MID de *nd* : 0 domine 1 sur le bus ([83, item 4.2]).

Invariant L'algorithme est basé sur deux propriétés. D'une part, à chaque étape, dans la période de lecture les MIDs des nœuds encore en compétition sont égaux de la position 0 à la position $pos - 1$. Ainsi, avant la position *pos*, il n'est pas possible de distinguer les compétiteurs. D'autre part, si deux nœuds en compétition, disons *nd1* et *nd2*, ne peuvent pas être distingués avant que la position *pos* du MID soit atteinte (nous rappelons que nécessairement *pos(nd1)=pos(nd2)* dans la période de lecture) et que *MID(nd1)(pos(nd1))=0* and *MID(nd2)(pos(nd2))=1*, alors *nd1* est plus petit que *nd2* (au sens de la relation *nb* que nous remplaçons ainsi par une relation sur les vecteurs de bits).

La première propriété, (P), est la conséquence de nombreux invariants. Elle exprime que, pendant la période de lecture du cycle ($WRI = \varnothing$), la variable *pos* a la même valeur pour tous les nœuds. Cette propriété assure également que, pendant la période de lecture, il n'est pas possible de distinguer les nœuds qui étaient encore en compétition à la fin du cycle précédent.

$$
\text{(P)}\quad \boxed{\begin{array}{l} (WRI = \varnothing \Rightarrow \forall\ (nd1,\ nd2).(nd1 \in CP \wedge nd2 \in CP \Rightarrow pos(nd1){=}pos(nd2))) \wedge \\ (WRI = \varnothing \Rightarrow \forall\ (nd1,\ nd2,\ pp).(nd1 \in CP \wedge nd2 \in CP \wedge pp \in 0\ ..\ pos(nd1){-}1 \Rightarrow \\ \quad (MID(nd1))(pp) = (MID(nd2))(pp))) \end{array}}
$$

La seconde propriété a été prouvée dans une autre machine B car elle nécessite de prouver de nombreux lemmes. De plus, il s'agit d'une propriété indépendante du modèle du système, elle exprime la relation d'ordre entre des vecteurs de bits. La machine utilisée pour faire cette preuve est donnée en annexe E.

$$
\boxed{\begin{array}{l} \forall\ (nd1,\ nd2,\ posn).(nd1 \in ND \wedge nd2 \in ND \wedge posn \in 0\ ..\ MAX\_BIT\_CHAR \quad \wedge \\ \quad (\ \forall\ pp.(pp \in 0\ ..\ posn{-}1 \Rightarrow (MID(nd1))(pp) = (MID(nd2))(pp))) \wedge \\ \quad (MID(nd1))(posn) = 0 \wedge (MID(nd2))(posn) = 1 \Rightarrow nb(nd1){<}nb(nd2)\ ) \end{array}}
$$

Événements Nous pouvons maintenant raffiner l'événement *check_ko* : si le MID d'un nœud *nd* est 1 à la position courante alors qu'il existe un autre nœud *ne* dont le MID vaut 0 à la même position, *nd* doit quitter la compétition car il est plus grand que *ne*. Nous avons également besoin d'un événement pour modéliser les nœuds qui n'ont pas besoin de quitter la compétition, c'est l'événement *check_ok*.

En utilisant l'invariant (BusInv), on peut déduire qu'un nœud doit quitter la compétition si ce qu'il a envoyé sur le bus diffère de la valeur du bus. Si la valeur portée par le bus est 1, tous les nœuds ont écrit 1, par conséquent aucun nœud ne doit quitter la compétition. Si la valeur portée par le bus est 0, il existe au moins un nœud qui a envoyé la valeur 0 sur le bus, par conséquent tous les nœuds qui ont envoyé 1 sur le bus doivent quitter la compétition (*check_ko*), les autres (qui ont envoyé 0) restent dans la compétition (*check_ok*). Dans tous le cas ils remettent à jour leur compteur *fc*.

$$
\boxed{\begin{array}{l} \textbf{check\_ok} = \\ \quad \text{ANY } nd \text{ WHERE} \\ \quad\quad nd \in VER \wedge WRI = \varnothing \wedge \\ \quad\quad nd \in CP \quad \wedge pos(nd) < MAX\_BIT\_CHAR \wedge \\ \quad\quad (MID(nd))(pos(nd)) = bus \\ \quad \text{THEN} \\ \quad\quad fc(nd) := (1{-}bus) \times COUNT\_FREE + bus \times max(\{fc(nd)\ {-}\ 1{,}0\}) \parallel \\ \quad\quad VER := VER\ {-}\ \{nd\} \\ \quad \text{END} \end{array}}
$$

check_ko =
ANY *nd* WHERE
$nd \in VER \wedge WRI = \varnothing \wedge$
$nd \in CP \quad \wedge pos(nd) < MAX\_BIT\_CHAR \wedge$
$(MID(nd))(pos(nd)) \neq bus$
THEN
$CP := CP\text{-}\{nd\} \parallel$
$pos(nd) := 0 \parallel$
$fc(nd) := (1\text{-}bus) \times COUNT\_FREE + bus \times max(\{fc(nd) - 1,0\}) \parallel$
$VER := VER - \{nd\}$
END

Nous devons également raffiner l'événement *take*. Dans le raffinement précédent, la compétition se terminait lorsqu'il ne restait plus qu'un seul nœud en compétition. Grâce à la propriété (P), appliquée à $pos{=}MAX\_BIT\_CHAR$, nous savons que si $pos(nd){=}MAX\_BIT\_CHAR$ alors *nd* est nécessairement le seul nœud restant en compétition (la fonction *MID* est injective). De plus, on peut prouver qu'il n'y a aucun nœud en phase d'attente ($dom(AT) = \varnothing$) quand $pos{=}MAX\_BIT\_CHAR$. Par conséquence, on peut retirer $dom(AT) = \varnothing$ de la garde de l'événement *take* et remplacer $CP = \{nd\}$ par $pos(nd){=}MAX\_BIT\_CHAR$.

Le message à envoyer est choisi par l'événement *take* en utilisant la fore ANY de l'événement. Le choix du message est donc non déterministe. Ceci pourra se raffiner de façon à obtenir le message par une entrée (c'est-à-dire mise à jour par l'événement *env*). Cette utilisation de la forme ANY est donc considérée comme une façon plus simple de spécifier une entrée du circuit. On peut également faire en sorte d'obtenir le message de façon moins direct, bit par bit par exemple (au lieu de l'ensemble du message en un coup comme c'est le cas ici), mais ceci ne faisait pas partie de l'étude de cas initialement.

take =
ANY *nd*, *LMSG*, *MSG* WHERE
$nd \in VER \wedge WRI = \varnothing \wedge$
$nd \in CP \wedge pos(nd){=}MAX\_BIT\_CHAR \wedge$
$LMSG \in \{nd\} \rightarrow \mathbb{N}_1 \wedge$
$MSG : (\{nd\} \rightarrow (MESSAGES \cap (0 \mathinner{..} LMSG(nd)\text{-}1 \rightarrow CHAR)))$
THEN
$wr := \{nd\} \parallel$
$CP := CP - \{nd\} \parallel$
$pos(nd) := 0 \parallel$
$pos\_message(nd) := 0 \parallel$
$pos\_msg\_char(nd) := 0 \parallel$
$LONGMESSAGE := LMSG \parallel$
$MESSAGE := MSG \parallel$
$XR := \varnothing \quad \parallel$
$fc(nd) := (1\text{-}bus) \times COUNT\_FREE + bus \times max(\{fc(nd) - 1,0\}) \parallel$
$VER := VER - \{nd\}$
END

Phases d'attente et d'élimination Les événements des phases d'attente et d'élimination (*rise*, *rise_count*, *switch_ok* et *switch_ko*) sont raffinés de façon à prendre en compte l'ordonnancement des périodes du cycle et de la mise à jour du compteur *fc* qui est faite à chaque cycle.

La condition $bus{=}1$ est ajoutée à la garde de l'événement *rise* : si le bus porte la valeur 0, il n'est pas libre et donc aucun nœud ne peut commencer une phase d'attente. On verra par la suite qu'on pourra déduire le reste de la garde à partir de cette condition.

rise =
ANY nd, t WHERE
$nd \in VER \wedge WRI = \varnothing \wedge$
$CP = \varnothing \wedge wr = \varnothing \wedge$
$nd \notin dom(AT) \wedge t \in \mathbb{N}_1 \wedge$
$nd \notin XR \wedge$
bus=1
THEN
$AT := AT \leftarrow \{nd \mapsto t\} \parallel$
$pos(nd) := 0 \parallel$
$fc(nd) := (1\text{-}bus) \times COUNT\_FREE + bus \times max(\{fc(nd) - 1,0\}) \parallel$
$VER := VER - \{nd\}$
END

La condition $bus = 1$ est ajoutée à la garde de l'événement *rise_count* : si le bus porte la valeur 0, le nœud doit quitter la compétition (événement *switch_ko*). En effet, si le bus porte la valeur 0 cela signifie qu'un nœud a terminé sa phase d'attente. Si un nœud n'a pas fini sa phase d'attente en même temps, il doit se retirer. On remarquera que nous avons retiré la condition $CP = \varnothing$ de la garde de l'événement *rise_count*. Ceci peut être fait grâce aux invariants suivants :

$(dom(AT) \neq \varnothing \Rightarrow \forall\, nd.(nd \in CP \Rightarrow pos(nd)=0)) \wedge$
$(VER \neq \varnothing \Rightarrow (\forall\, nd.(nd \in CP\text{-}WRI \Rightarrow ((MID(nd))(pos(nd)) \neq bus \Rightarrow bus = 0))))$

En effet, en se rappelant que la propriété $MID(nd)(0) = 0$ est toujours vraie, on peut déduire que si CP n'est pas vide, il doit exister un nœud en compétition et, puisque la position courante dans le MID est 0, la valeur portée par le bus ne peut pas être autre chose que 0. Ainsi, si CP n' est pas vide, la valeur portée par le bus, pendant la période de lecture, ne peut pas être 1. La condition $CP = \varnothing$ peut donc être retirée de la garde de l'événement *rise_count*.

rise_count =
ANY nd WHERE
$nd \in VER \wedge WRI = \varnothing \wedge$
$nd \in dom(AT) \wedge \quad AT(nd) \neq 0 \wedge$
bus=1
THEN
$AT := AT \leftarrow \{nd \mapsto AT(nd)\text{-}1\} \parallel$
$fc(nd) := (1\text{-}bus) \times COUNT\_FREE + bus \times max(\{fc(nd) - 1,0\}) \parallel$
$VER := VER - \{nd\}$
END

switch_ko =
ANY nd WHERE
$nd \in VER \wedge WRI = \varnothing \wedge$
$nd \in dom(AT) \wedge AT(nd) \neq 0 \wedge$
$bus = 0$
THEN
$AT := \{nd\} \lhd AT \parallel$
$VER := VER - \{nd\} \parallel$
$XR := XR \cup \{nd\} \parallel$
$fc(nd) := (1\text{-}bus) \times COUNT\_FREE + bus \times max(\{fc(nd) - 1,0\})$
END

switch_ok =
ANY *nd* WHERE
$nd \in VER \wedge WRI = \varnothing \wedge$
$nd \in dom(AT) \wedge AT(nd)=0 \wedge$
$bus = 0$
THEN
$CP := CP \cup \{nd\} \parallel$
$AT := \{nd\} \lhd AT \parallel$
$fc(nd) := (1\text{-}bus) \times COUNT\_FREE + bus \times max(\{fc(nd) - 1,0\}) \parallel$
$VER := VER - \{nd\}$
END

Phases de message et de latence Pendant la période de lecture, le nœud qui a l'accès en écriture au bus n'a rien à faire à part libérer le bus lorsqu'il a fini d'envoyer son message et que la phase de latence est terminée ($pos\_message(nd) = LONGMESSAGE(nd) \wedge fc(nd) \leq 1$).

free =
ANY *nd* WHERE
$nd \in VER \wedge WRI = \varnothing \wedge$
$nd \in wr \wedge pos\_message(nd) = LONGMESSAGE(nd) \wedge fc(nd) \leq 1$
THEN
$wr := \varnothing \parallel$
$LONGMESSAGE, pos\_message, MESSAGE, pos\_msg\_char := \varnothing , \varnothing , \varnothing , \varnothing \parallel$
$fc(nd) := (1\text{-}bus) \times COUNT\_FREE + bus \times max(\{fc(nd) - 1,0\}) \parallel$
$VER := VER - \{nd\}$
END

L'événement *sender* modélise le comportement d'un nœud dans la période de lecture pendant qu'il envoie son message (lorsqu'un message est en train d'être envoyé : $pos\_message(nd) < LONGMESSAGE(nd)$)

sender =
ANY *nd* WHERE
$nd \in VER \wedge WRI = \varnothing \wedge$
$nd \in wr \wedge pos\_message(nd) < LONGMESSAGE(nd)$
THEN
$fc(nd) := (1\text{-}bus) \times COUNT\_FREE + bus \times max(\{fc(nd) - 1,0\}) \parallel$
$VER := VER - \{nd\}$
END

L'événement *sender_w* modélise le comportement d'un nœud pendant la phase de latence. Rappelons que seul le composant ayant fini d'envoyer un message a accès à cette phase, pour les autres cette phase fait partie de la phase du message. Un tel nœud ne peut savoir qu'il s'agissait d'une phase de latence qu'une fois qu'il détecte que le bus est libre. La phase de latence est caractérisée par la condition $pos\_message(nd) = LONGMESSAGE(nd) \wedge fc(nd)>1$).

sender_w =
ANY *nd* WHERE
$nd \in VER \wedge WRI = \varnothing \wedge$
$nd \in wr \wedge pos\_message(nd) = LONGMESSAGE(nd) \wedge fc(nd)>1$
THEN
$fc(nd) := (1\text{-}bus) \times COUNT\_FREE + bus \times max(\{fc(nd) - 1,0\}) \parallel$
$VER := VER - \{nd\}$
END

Nothing Les nœuds qui n'ont rien à faire (partie du système non modélisée explicitement) doivent tout de même tenir compte de l'ordonnancement et mettre à jour le compteur *fc* pour savoir si le bus est libre ou non.

nothing =
ANY *nd* WHERE
$nd \in VER \wedge WRI = \varnothing \wedge$
$nd \notin dom(AT) \wedge nd \notin CP \wedge nd \notin wr$
THEN
$fc(nd) := (1\text{-}bus) \times COUNT\_FREE + bus \times max(\{fc(nd) - 1,0\}) \parallel$
$VER := VER - \{nd\}$
END

Période d'écriture

Phase de compétition L'événement *bus_write* modélise l'envoi des bits par les nœuds en compétition pendant la phase de compétition. Remarquez que, il est impossible de se retrouver dans une situation où $pos(nd) = MAX\_BIT\_CHAR$ dans ce cas, car, quand la position du nœud atteint MAX_BIT_CHAR, le nœud est nécessairement le seul en compétition et entre dans l'ensemble *wr* (événement *take*).

bus_write =
ANY *nd* WHERE $nd \in CP \wedge pos(nd) < MAX\_BIT\_CHAR \wedge$
$nd \in WRI$
THEN
$pos(nd) := pos(nd) + 1 \parallel$
$bus := \min(\{bus, (MID(nd))(pos(nd)+1)\}) \parallel$
$VER := VER \cup \{nd\} \parallel$
$WRI := WRI - \{nd\}$
END

Un nœud envoie la valeur 0 sur le bus lorsqu'il finit sa phase d'attente. La valeur 0 correspond au bit de départ de son MID. Ceci est modélisé par l'événement *bus_write_fst*.

bus_write_first =
ANY *nd* WHERE
$nd \in dom(AT) \wedge AT(nd)=0 \wedge nd \in WRI$
THEN
$WRI := WRI - \{nd\} \parallel$
$VER := VER \cup \{nd\} \parallel$
$bus := 0$
END

Phase d'attente L'événement *bus_write_nothing* modélise le comportement des nœuds qui sont en phase d'attente : ils ne modifient pas le bus.

bus_write_nothing=
ANY *nd* WHERE
$nd \in dom(AT) \wedge AT(nd) \neq 0 \quad \wedge nd \in WRI$
THEN
$WRI := WRI - \{nd\} \parallel$
$VER := VER \cup \{nd\}$
END

Phase d'envoi d'un message Nous ajoutons deux nouveaux événements pour modéliser l'envoi d'un message. Il faut noter que durant cette phase de message, il n'y a qu'un seul nœud qui écrit sur le bus. Ainsi, il n'est pas nécessaire d'utiliser l'expression avec *min* comme nous l'avons fait pour la phase de compétition.

La transmission du message est faite par l'événement *send*. Le nœud envoie la valeur du message à la position courante sur le bus, et la position est incrémentée.

send =
ANY nd WHERE
$nd \in WRI \wedge$
$nd \in wr \wedge pos\_message(nd) < LONGMESSAGE(nd) \wedge$
$pos\_msg\_char(nd) < MAX\_BIT\_CHAR$
THEN
$pos\_msg\_char(nd) := pos\_msg\_char(nd)+1 \parallel$
$bus := MESSAGE(nd)(pos\_message(nd))(pos\_msg\_char(nd)) \parallel$
$WRI := WRI - \{nd\} \parallel$
$VER := VER \cup \{nd\}$
END

Le dernier bit d'un caractère est traité séparément. Le nœud écrit la valeur du message à la position courante sur le bus comme précédemment, mais la position doit être incrémentée au prochain caractère. Ceci est modélisé par l'événement *send_next_char*.

send_next_char =
ANY nd WHERE
$nd \in WRI \wedge$
$nd \in wr \wedge pos\_message(nd) < LONGMESSAGE(nd) \wedge$
$pos\_msg\_char(nd) = MAX\_BIT\_CHAR$
THEN
$bus := MESSAGE(nd)(pos\_message(nd))(pos\_msg\_char(nd)) \parallel$
$pos\_message(nd) := pos\_message(nd)+1 \parallel$
$pos\_msg\_char(nd) := 0 \parallel$
$WRI := WRI - \{nd\} \parallel$
$VER := VER \cup \{nd\}$
END

Phase de latence Lorsqu'un nœud a fini d'envoyer son message, il attend un certain temps de latence pour libérer le bus. Durant cette phase, le nœud ne modifie pas le bus, qui porte donc constamment la valeur 1.

sender_nothing =
ANY nd WHERE
$nd \in WRI \wedge$
$nd \in wr \wedge pos\_message(nd) = LONGMESSAGE(nd)$
THEN
$WRI := WRI - \{nd\} \parallel$
$VER := VER \cup \{nd\}$
END

Nothing De façon similaire à la période de lecture, nous devons modéliser le comportement des nœuds inactifs, qui ne modifient pas le bus (ils sont en mode réception de message, non modélisé dans cette étude de cas).

bus_nothing=
ANY *nd* WHERE
$nd \in WRI \wedge$
$nd \notin wr \wedge nd \notin CP \wedge nd \notin dom(AT)$
THEN
$WRI := WRI - \{nd\} \parallel$
$VER := VER \cup \{nd\}$
END

7.10 Suppression de la variable XR

Cette section explique comment retirer la variable XR du modèle. En effet, il s'agit d'une variable abstraite qui doit donc être retirée pour aller vers un modèle implantable. Nous montrons comment les gardes utilisant cette variable peuvent être écrites sans utiliser la variable XR. Cette suppression a été faite en deux raffinements. Rappelons que plusieurs raffinements pourraient très bien se faire en un seul mais que le fait de les diviser en plusieurs pas de raffinement permet de simplifier les transformations et les preuves à faire à chaque pas de raffinement.

Le premier raffinement retire XR du modèle et remplace la garde $nd \notin XR$ de l'événement *rise* par une garde exprimée en utilisant les autres variables. Le deuxième raffinement montre que cette nouvelle garde peut en fait être prouvée à partir du contexte et qu'elle peut donc être retirée. Ainsi, en fin de compte, la variable XR est retirée du modèle sans rien ajouter au modèle.

7.10.1 Premier raffinement

Ce raffinement retire la variable XR du modèle. La suppression d'une variable est une forme de raffinement, il faut donc prouver qu'après la suppression de la variable le système continue à avoir un comportement en accord avec le modèle qu'il raffine. La suppression de la variable XR de la partie substitution d'un événement (ici *take* et *switch_ko*) se fait sans difficulté mais retirer XR de la garde de l'événement *rise* requiert que la nouvelle garde implique l'ancienne garde.

Pour faire cela, nous avons prouvé l'invariant suivant : s'il existe un nœud dans l 'ensemble XR, soit il y a au moins un nœud en compétition, soit il existe un nœud nd qui a terminé sa phase d'attente ($AT(nd) = 0$) mais n'a pas encore fait sa période de lecture ($nd \in VER$). Plus formellement ceci peut s'écrire sous la forme suivante :

$$(XR \neq \varnothing \Rightarrow ((dom(AT \rhd \{0\}) \cap VER) \cup CP) \neq \varnothing)$$

Ainsi, si nous avons les deux propriétés $dom(AT \rhd \{0\}) \cap VER = \varnothing$ et $CP = \varnothing$, on peut déduire que l'ensemble XR est vide, et qu'alors la garde $nd \notin XR$ peut être remplacée par $dom(AT \rhd \{0\}) \cap VER = \varnothing$. Les raffinements montrés ci-dessous de *take* et *switch_ko* ne sont affectés que par la suppression de la variable XR.

rise =
ANY *nd*, *t* WHERE
$nd \in VER \wedge WRI = \varnothing \wedge$
$CP = \varnothing \wedge wr = \varnothing \wedge$
$nd \notin dom(AT) \wedge t \in \mathbb{N}_1 \wedge dom(AT \rhd \{0\}) \cap VER = \varnothing \wedge$
$bus=1$
THEN
$AT := AT \Leftarrow \{nd \mapsto t\} \parallel$
$pos(nd) := 0 \parallel$
$fc(nd) := (1\text{-}bus) \times COUNT\_FREE + bus \times max(\{fc(nd) - 1,0\}) \parallel$
$VER := VER - \{nd\}$
END

switch_ko =
ANY *nd* WHERE
$nd \in VER \wedge WRI = \varnothing \wedge$
$nd \in dom(AT) \wedge AT(nd) \neq 0 \wedge$
$bus = 0$
THEN
$AT := \{nd\} \lhd AT \parallel$
$fc(nd) := (1\text{-}bus) \times COUNT\_FREE + bus \times max(\{fc(nd) - 1,0\}) \parallel$
$VER := VER - \{nd\}$
END

take =
ANY *nd*, *LMSG*, *MSG* WHERE
$nd \in VER \wedge WRI = \varnothing \wedge$
$nd \in CP \wedge pos(nd) = MAX\_BIT\_CHAR \wedge$
$LMSG \in \{nd\} \rightarrow \mathbb{N}_1 \wedge$
$MSG : (\{nd\} \rightarrow (MESSAGES \cap (0 \,..\, LMSG(nd)\text{-}1 \rightarrow CHAR)))$
THEN
$wr := \{nd\} \parallel$
$CP := CP - \{nd\} \parallel$
$pos(nd) := 0 \parallel$
$pos\_message(nd) := 0 \parallel$
$pos\_msg\_char(nd) := 0 \parallel$
$LONGMESSAGE := LMSG \parallel$
$MESSAGE := MSG \parallel$
$fc(nd) := (1\text{-}bus) \times COUNT\_FREE + bus \times max(\{fc(nd) - 1,0\}) \parallel$
$VER := VER - \{nd\}$
END

7.10.2 Deuxième raffinement

Ce deuxième raffinement supprime la garde $dom(AT \rhd \{0\}) \cap VER = \varnothing$ de la garde de l'événement *rise*. En effet, on peut prouver l'invariant écrit formellement ci-dessous : si cette garde n'était pas vraie, alors le bus devrait nécessairement porter la valeur 0. Comme la garde de l'événement *rise* assure que le bus porte la valeur 1, la garde $dom(AT \rhd \{0\}) \cap VER = \varnothing$ peut être retirée.

$$(dom(AT \rhd \{0\}) \cap VER \neq \varnothing \Rightarrow bus = 0)$$

rise =
ANY *nd*, *t* WHERE
$nd \in VER \wedge WRI = \varnothing \wedge$
$CP = \varnothing \wedge wr = \varnothing \wedge$
$nd \notin dom(AT) \wedge t \in \mathbb{N}_1 \wedge$
$bus = 1$
THEN
$AT := AT \leftarrow \{nd \mapsto t\} \parallel$
$pos(nd) := 0 \parallel$
$fc(nd) := (1\text{-}bus) \times COUNT\_FREE + bus \times max(\{fc(nd) - 1,0\}) \parallel$
$VER := VER - \{nd\}$
END

7.11 Localisation des gardes

Notre modèle prend en compte l'ensemble des comportements de chaque nœud. Mais pour être acceptables, les gardes des événements ne doivent se référer qu'à des variables modélisant l'état interne du

nœud concerné, et non pas à des variables globales (comme la variable CP par exemple).

Nous n'introduisons pas de nouveaux événements dans ce raffinement. Comme précédemment nous avons séparé cette étape en deux raffinements à cause du nombre d'invariants à prouver.

7.11.1 Gardes locales

Les conditions de la forme $nd \in Set$ sont considérées comme locales si des substitutions de la forme $Set := Set \cup \{nd\}$ apparaissent seulement dans des événements dont la garde est de la forme ANY nd WHERE. C'est toujours le cas dans notre modèle : chaque nœud sait s'il fait partie d'un ensemble ou non.

Ceci permet d'implanter les ensembles par des variables locales à chaque nœud, par exemple par des variables booléennes (une pour chaque nœud). Par exemple, la substitution $Set := Set \cup \{nd\}$ est implantée par la substitution $Set(nd) := true$, et $Set := Set - \{nd\}$ par $Set(nd) := false$. Le test $nd \in Set$ peut alors s'implanter par le test $Set(nd) = true$.

Par ailleurs, des conditions comme $CP = \varnothing$ ne sont pas considérées comme locales car un nœud ne peut pas déduire le résultat de ce test à partir uniquement de son propre état. Nous devons donc raffiner le modèle de façon à ce qu'on puisse déduire l'état global du système à partir de l'état local d'un nœud. Notre but est de supprimer la garde $CP = \varnothing \wedge wr = \varnothing$ de la garde de l'événement *rise*.

7.11.2 Premier raffinement

Ce raffinement supprime la condition $CP = \varnothing$ de la garde de l'événement *rise*. Nous ajoutons la condition $fc(nd)$=0, qui est une condition locale, dans la garde. En utilisant la propriété ci-dessous (déduite d'un nombre important d'invariants) on peut déduire que $CP = \varnothing$ de la nouvelle garde de l'événement *rise*.

$$(CP \neq \varnothing \Rightarrow (\ \forall\ nd.(nd \in CP \Rightarrow fc(nd) \neq 0)))$$

rise =
ANY *nd*, *t*WHERE
$nd \in VER \wedge WRI = \varnothing \wedge$
$wr = \varnothing \wedge nd \notin dom(AT) \wedge t \in \mathbb{N}_1 \wedge fc(nd)=0 \wedge$
$bus=1$
THEN
$AT := AT \Leftarrow \{nd \mapsto t\} \parallel$
$pos(nd) := 0 \parallel$
$VER := VER - \{nd\} \parallel$
$fc(nd) := (1-bus) \times COUNT\_FREE + bus \times max(\{fc(nd) - 1,0\})$
END

7.11.3 Deuxième raffinement

Ce raffinement fait le même travail pour la condition $wr = \varnothing$. La propriété ci-dessous est déduite à partir d'un nombre important d'invariants. Cette propriété permet de retirer la condition $wr = \varnothing$ de la garde de l'événement *rise*.

$$\forall\ nd.(nd \in VER \wedge fc(nd) = 0 \wedge bus = 1 \Rightarrow wr = \varnothing\ )$$

rise =
ANY *nd*, *t* WHERE
$nd \in VER \wedge WRI = \varnothing \wedge$
$nd \notin dom(AT) \wedge t \in \mathbb{N}_1 \wedge fc(nd)=0 \wedge$
$bus=1$
THEN
$AT := AT \Leftarrow \{nd \mapsto t\} \parallel$
$pos(nd) := 0 \parallel$
$VER := VER - \{nd\} \parallel$
$fc(nd) := (1-bus) \times COUNT\_FREE + bus \times max(\{fc(nd) - 1,0\})$
END

En utilisant la même propriété, on peut aussi remplacer la condition $fc(nd) \leq 1$ par la condition $fc(nd)=1$ dans la garde de l'événement *free*.

free =
ANY *nd* WHERE
$nd \in VER \wedge$
$nd \in wr \wedge WRI = \varnothing \wedge pos\_message(nd) = LONGMESSAGE(nd) \wedge fc(nd)=1$
THEN
$wr := \varnothing \parallel$
$VER := VER - \{nd\} \parallel$
$LONGMESSAGE, pos\_message, MESSAGE, pos\_msg\_char := \varnothing , \varnothing , \varnothing , \varnothing \parallel$
$fc(nd) := (1-bus) \times COUNT\_FREE + bus \times max(\{fc(nd) - 1,0\})$
END

7.12 Concrétisation

Cette section résume quatre pas de raffinements. Il s'agit premièrement de normaliser l'écriture sur le bus en utilisant systématiquement l'expression *min*. Deuxièmement, nous rendons explicites les entrées des nœuds. Troisièmement, nous concrétisons les types des variables de façon à n'utiliser que des types implantables. Enfin, nous rendons explicite la sortie *out* des nœuds vers le bus.

7.12.1 Normalisation de l'envoi d'un bit sur le bus

Dans ce raffinement il n'y a quasiment pas de modification. Nous modifions simplement les substitutions qui modifient le bus de façon à ce qu'elles soient toutes sur le modèle suivant (cf. section 7.8.3) :

bus := min(bus, output)

7.12.2 Concrétisation des types de variables

Jusqu'à présent nous n'avons pas modélisé explicitement les entrées du système. Nous les avons modélisées, par exemple dans l'événement *take*, par un choix non déterministe en utilisant la forme ANY des événements. Ce raffinement rend explicite les entrées des nœuds en créant les variables correspondantes.

Un nœud possède quatre entrées :

- Un bit indiquant si le composant raccordé au nœud souhaite envoyer un message ou non sur le bus. Cette entrée était modélisée de façon très abstraite jusqu'ici en utilisant le non déterminisme dans le choix entre les événements *rise* (envoi d'un message) et *nothing* (pas d'envoi). Cette entrée est nommée *in_wri*.

- La durée d'attente pour la phase d'attente précédent l'accès au bus (pour la compétition). Cette durée d'attente est une décision du composant associé au nœud, c'est pourquoi c'est une entrée

du nœud. Jusqu'ici cette entrée était modélisée par le choix non déterministe d'une valeur t dans l'événement *rise*.

- Et, bien sûr, le message *in_msg* et la longueur *in_lmsg* qui étaient jusqu'à présent choisis par l'événement *take*

Dans le modèle B, les entrées sont de simples variables. Cependant on ne leur assigne jamais une valeur particulière (ou alors ce serait une entrée constante). Par ailleurs, une variable modélisant une entrée ne peut être modifiée qu'en deux endroits dans le modèle : dans l'initialisation (puisque toute variable du modèle doit être initialisée), et dans l'événement *env* qui modélise de façon abstraite l'environnement et qui est donc chargé de fournir les entrées.

Comme nous venons de le dire, l'événement *env* modélise l'environnement, nous devons donc modéliser la façon dont l'environnement modifie les entrées du système. De plus, nous avons déjà dit que l'événement *env* modélise également le passage d'un cycle à l'autre. Cela signifie que, au niveau du circuit physique, ce modèle n'autorise les entrées à ne changer qu'au *front montant* de l'horloge. C'est un bon modèle puisque, même si dans la réalité une entrée peut être modifiée à tout instant, les nouvelles valeurs se propagent dans le circuit à chaque modification, et par conséquent les nouvelles valeurs des registres internes (au prochain front montant de l'horloge) ne dépendent que des dernières valeurs des entrées. Tout se passe comme si ces dernières entrées avaient été présentes dès le front montant de l'horloge. Rappelons que dans un circuit physique les délais de propagation ne sont pas nuls et que la durée de la période de l'horloge est généralement calculée de façon à ce que tous les signaux du circuit aient le temps de se propager pendant le cycle. En général, on fait l'hypothèse que les entrées changent en début de cycle, au front montant de l'horloge, ou très peu de temps après.

Les variables modélisant les entrées doivent être initialisées car toute variable doit l'être dans un modèle B. Cette initialisation est non déterministe pour les entrées car lorsque le système s'initialise, le premier événement à se déclencher est l'événement *env* qui donne aux entrées leurs valeurs.

L'invariant pour ces nouvelles variables est le suivant :

CONSTANTS
  $LONG\_MAX\_MESSAGE$
PROPERTIES
  $LONG\_MAX\_MESSAGE \in \mathbb{N}_1 \wedge$
  $\forall\, nn.(\, nn \in \mathbb{N}_1 \wedge nn \in 1 \,..\, LONG\_MAX\_MESSAGE \Rightarrow 0 \,..\, nn\text{-}1 \rightarrow CHAR \subseteq MESSAGES)$
INVARIANT
  $in\_wri \in ND \rightarrow BOOL \wedge$
  $in\_tt \in ND \rightarrow \mathbb{N}_1 \wedge$
  $in\_lmsg \in ND \rightarrow 1 \,..\, LONG\_MAX\_MESSAGE \wedge$
  $in\_msg \in ND \rightarrow MESSAGES \wedge$
  $\forall\, nd.(nd \in ND \Rightarrow in\_msg(nd) \in (0 \,..\, LONG\_MAX\_MESSAGE\text{-}1 \rightarrow CHAR))$

On notera que désormais le message a une longueur fixée, une longueur de taille maximum *LONG_MAX_MESSAGE*. La variable *in_lmsg* est la longueur réelle du message. Les données de *in_msg* après *in_lmsg* n'ont aucune signification et ne doivent pas être lues. C'est le cas dans notre modèle puisque la variable *MESSAGE* qui sauvegarde le message a exactement la taille *in_lmsg* (cf. événement *take* ci-dessous).

L'initialisation des nouvelles variables est ajoutée :

INITIALISATION
  ... ||
  $in\_wri :\in ND \rightarrow BOOL \,\|$
  $in\_tt :\in ND \rightarrow \mathbb{N}_1 \,\|$
  $in\_msg :\in (ND \rightarrow (0 \,..\, LONG\_MAX\_MESSAGE\text{-}1 \rightarrow CHAR)) \,\|$
  $in\_lmsg :\in ND \rightarrow 1 \,..\, LONG\_MAX\_MESSAGE$

Et les mêmes substitutions non déterministes (pas de contrainte supplémentaire dans ce modèle) sont ajoutées dans l'événement *env*.

env =
WHEN
$VER = \varnothing \wedge WRI = \varnothing$
THEN
$WRI := ND \parallel bus := 1 \parallel$
$in\_wri :\in ND \rightarrow BOOL \parallel$
$in\_tt :\in ND \rightarrow \mathbb{N}_1 \parallel$
$in\_msg :\in (ND \rightarrow (0 \,..\, LONG\_MAX\_MESSAGE\text{-}1 \rightarrow CHAR)) \parallel$
$in\_lmsg :\in ND \rightarrow 1 \,..\, LONG\_MAX\_MESSAGE$
END

Les gardes des événements *rise* et *nothing* sont modifiées de façon à prendre en compte l'entrée *in_wri*. Si *in_wri* est vraie alors le composant souhaite envoyer un message (*rise*), sinon il est en mode récepteur (*nothing*).

rise =
ANY *nd* WHERE
$nd \in VER \wedge WRI = \varnothing \wedge$
$nd \notin dom(AT) \wedge fc(nd)=0 \wedge \quad bus=1 \wedge$
$in\_wri(nd)= TRUE$
THEN
$AT := AT \Leftarrow \{nd \mapsto in\_tt(nd)\} \parallel$
$pos(nd) :=0 \parallel$
$VER := VER - \{nd\} \parallel$
$fc(nd) := ...$
END

nothing =
ANY *nd* WHERE
$nd \in VER \wedge WRI = \varnothing \wedge$
$nd \notin dom(AT) \wedge$
$nd \notin CP \wedge$
$nd \notin wr \wedge$
$(nd \notin dom(AT) \Rightarrow$
$\neg (fc(nd)=0 \wedge bus=1 \wedge in\_wri(nd)= \text{TRUE}))$
THEN
$VER := VER - \{nd\} \parallel$
$fc(nd) := ...$
END

L'événement *take* est modifié de façon à ce que le message soit celui obtenu en entrée. Notons, que la variable *in_msg* est contrainte avant d'être assignée à *MESSAGE* car *in_msg* est un message de longueur totale *LONG_MAX_MESSAGE* alors que *MESSAGE* doit avoir exactement la longueur *LONGMESSAGE*(= *in_lmsg*).

take =
ANY *nd* WHERE
$nd \in VER \wedge WRI = \varnothing \wedge$
$nd \in CP \wedge pos(nd)=MAX\_BIT\_CHAR$
THEN
$wr :=\{nd\} \parallel$
$CP := CP - \{nd\} \parallel$
$pos(nd) :=0 \parallel$
$pos\_message(nd) := 0 \parallel$
$pos\_msg\_char(nd) := 0 \parallel$
$LONGMESSAGE := \{nd\} \lhd in\_lmsg \parallel$
$MESSAGE := \{nd\} \times \{((0 \,..\, in\_lmsg(nd)\text{-}1) \lhd in\_msg(nd))\} \parallel$
$VER := VER - \{nd\} \parallel$
$fc(nd) := (1\text{-}bus) \times COUNT\_FREE + bus \times max(\{fc(nd) - 1{,}0\})$
END

Le modèle obtenu est alors déterministe en fonction des entrées. Tous les événements, à part *env*, sont de la forme ANY *nd* WHERE.

7.12.3 Concrétisation des variables d'état des nœuds

Le modèle que nous avons obtenu ne peut pas être traduit tel quel vers un langage de description de circuits car certains types utilisés ne sont pas directement implantables.

Par ailleurs, les variables doivent être modélisées comme des fonctions de type $ND \rightarrow aType$ parce que tous les nœuds ont le même ensemble de variables d'état (factorisation des variables). Cependant certaines variables du modèle sont jusqu'ici modélisées par des fonctions partielles : c'est parce que dans certaines situations, les valeurs de ces variables n'ont pas d'importance (et faire en sorte que ces variables n'aient pas de valeur dans ces cas rend les preuves plus simples). Maintenant notre but est d'obtenir un modèle plus proche de l'implantation : nous devons faire en sorte que toutes les variables soient des fonctions totales. De la même façon, les ensembles doivent être remplacés par des variables locales (modélisées par des fonctions totales).

Les ensembles sont remplacés par des valeurs booléennes, suivant le principe ci-dessous, où $BooleanSet$ est une fonction totale qui représente les états locaux des nœuds (cf. section 5.1) : $BooleanSet \in ND \rightarrow Bool$.

$$nd \in Set \equiv BooleanSet(nd) = true$$

Ensuite, les fonctions partielles sont remplacées par des fonctions totales en ajoutant une variable booléenne pour dire que le nœud est dans le domaine de la fonction partielle originale ou non. Par exemple, la fonction AT est remplacée par $TotalAT$ et $IsInDomAT$ suivant le principe ci-dessous.

$$nd \in dom(AT) \equiv IsInDomAT(nd) = true$$
$$\forall nd \cdot (nd \in dom(AT) \Rightarrow AT(nd) = TotalAT(nd))$$

De cette façon, les variables AT, CP, $LONGMESSAGE$, $MESSAGE$, $pos\_message$, pos_msg_char et wr sont raffinées. On donne ci-dessous l'invariant qui donne les types des nouvelles variables et les invariants de collage.

INVARIANT
$TotalAT \in ND \rightarrow \mathbb{N} \wedge$
$IsInDomAT \in ND \rightarrow BOOL \wedge$
$AT = dom((IsInDomAT \rhd \{TRUE\})) \lhd TotalAT \wedge$
$dom(AT) = dom(IsInDomAT \rhd \{TRUE\}) \wedge$
$TotalLONGMESSAGE \in ND \rightarrow \mathbb{N}_1 \wedge$
$LONGMESSAGE = wr \lhd TotalLONGMESSAGE \wedge$
$TotalMESSAGE \in ND \rightarrow (0 \,..\, LONG\_MAX\_MESSAGE\text{-}1 \rightarrow CHAR) \wedge$
$\forall nd \cdot (nd \in wr \Rightarrow$
$MESSAGE = \{nd\} \times \{((0 \,..\, LONGMESSAGE(nd)\text{-}1) \lhd TotalMESSAGE(nd))\}) \wedge$
$Totalpos\_message \in ND \rightarrow \mathbb{N} \wedge$
$pos\_message = wr \lhd Totalpos\_message \wedge$
$Totalpos\_msg\_char \in ND \rightarrow 0 \,..\, MAX\_BIT\_CHAR \wedge$
$pos\_msg\_char = wr \lhd Totalpos\_msg\_char \wedge$
$Booleanwr \in ND \rightarrow BOOL \wedge$
$wr = dom(Booleanwr \rhd \{\mathbf{TRUE}\}) \wedge$
$BooleanCP \in ND \rightarrow BOOL \wedge$
$CP = dom(BooleanCP \rhd \{TRUE\})$

La clause INITIALISATION correspondante est la suivante :

INITIALISATION
pos, VER, WRI, bus, $fc := ND \times \{0\}, \varnothing, \varnothing, 1, ND \times \{0\} \parallel$
$Booleanwr := ND \times \{FALSE\} \parallel$
$BooleanCP := ND \times \{FALSE\} \parallel$
$TotalLONGMESSAGE :\in ND \rightarrow \mathbb{N}_1 \parallel$
$TotalMESSAGE :\in ND \rightarrow (0 \,..\, LONG\_MAX\_MESSAGE\text{-}1 \rightarrow CHAR) \parallel$
$Totalpos\_message :\in ND \rightarrow \mathbb{N} \parallel$
$Totalpos\_msg\_char :\in ND \rightarrow 0 \,..\, MAX\_BIT\_CHAR \parallel$
$TotalAT$, $IsInDomAT := ND \times \{0\}, ND \times \{FALSE\} \parallel$
$in\_wri :\in ND \rightarrow BOOL \parallel$
$in\_tt :\in ND \rightarrow \mathbb{N}_1 \parallel$
$in\_msg :\in ND \rightarrow (0 \,..\, LONG\_MAX\_MESSAGE\text{-}1 \rightarrow CHAR) \parallel$
$in\_lmsg :\in ND \rightarrow 1 \,..\, LONG\_MAX\_MESSAGE$

Tous les événements doivent être modifiés pour tenir compte du raffinement des données. Notons que l'événement *take* n'a plus besoin de contraindre son message d'entrée avant de l'assigner à son message *TotalMESSAGE* car , pour tout nœud *nd*, *TotalMESSAGE(nd)* est aussi une fonction totale de longueur 0 .. *LONG_MAX_MESSAGE*-1, même si nous avons vu dans un précédent raffinement que la totalité de ce message ne sera pas lu. Pour illustrer cela, nous ne montrons que l'événement *take*, tous les autres événements étant modifiés sur le même modèle. On constatera que les variables *VER* et *WRI* restent abstraites : ce ne sont pas des variables d'état des nœuds, elles servent à ordonnancer les événements, cet ordonnancement sera introduit différemment lors du passage à BHDL.

take =
ANY *nd* WHERE
$nd \in VER \wedge WRI = \varnothing \wedge$
$BooleanCP(nd) = TRUE \wedge pos(nd) = MAX\_BIT\_CHAR$
THEN
$Booleanwr(nd) := TRUE \parallel$
$BooleanCP(nd) := FALSE \parallel$
$pos(nd) := 0 \parallel$
$Totalpos\_message(nd) := 0 \parallel$
$Totalpos\_msg\_char(nd) := 0 \parallel$
$TotalLONGMESSAGE := in\_lmsg \parallel$
$TotalMESSAGE := in\_msg \parallel$
$VER := VER - \{nd\} \parallel$
$fc(nd) := (1\text{-}bus) \times COUNT\_FREE + bus \times max(\{fc(nd) - 1, 0\})$
END

7.12.4 Explicitation de la sortie

Ce raffinement rend explicite la sortie des nœuds. La sortie est nommée *out*, c'est une variable modélisée par une fonction totale car chaque nœud a sa propre sortie.

INVARIANT
$out \in ND \rightarrow BIT$

Notons que la variable de sortie *out* est initialisée de façon non déterministe.

INITIALISATION
pos, VER, WRI, bus, $fc := ND \times \{0\}, \varnothing, \varnothing, 1, ND \times \{0\}$ ||
$Booleanwr := ND \times \{FALSE\}$ ||
$BooleanCP := ND \times \{FALSE\}$ ||
$TotalLONGMESSAGE :\in ND \rightarrow \mathbb{N}_1$ ||
$TotalMESSAGE :\in ND \rightarrow (0 \,..\, LONG\_MAX\_MESSAGE\text{-}1 \rightarrow CHAR)$ ||
$Totalpos\_message :\in ND \rightarrow \mathbb{N}$ ||
$Totalpos\_msg\_char :\in ND \rightarrow 0 \,..\, MAX\_BIT\_CHAR$ ||
$TotalAT$, $IsInDomAT := ND \times \{0\}, ND \times \{FALSE\}$ ||
$in\_wri :\in ND \rightarrow BOOL$ ||
$in\_tt :\in ND \rightarrow \mathbb{N}_1$ ||
$in\_msg :\in ND \rightarrow (0 \,..\, LONG\_MAX\_MESSAGE\text{-}1 \rightarrow CHAR)$ ||
$in\_lmsg :\in ND \rightarrow 1 \,..\, LONG\_MAX\_MESSAGE$ ||
$out :\in ND \rightarrow BIT$

Tous les événements de la période d'écriture du cycle doivent être raffinés de façon à ce que l'envoi d'un bit se fasse sur la variable *out*, et *après* le bus calcule sa nouvelle valeur à partir de la sortie *out*. Nous utilisons ici le point virgule qui dénote une composition séquentielle. Cette composition existait en B classique lorsque nous avons débuté nos travaux, elle n'existe plus dans le nouveau langage du B événementiel.

ANY nd WHERE
....
THEN
.... ||
$out\ (nd) := ...$ ||
$WRI := WRI - \{nd\}$ ||
$VER := VER \cup \{nd\}$
;
$bus := \min(\{bus, out(nd)\})$
END

A titre d'exemple nous donnons l'événement *send_next_char*, tous les autres événements de la période d'écriture sont modifiés sur le même modèle.

send_next_char =
ANY nd WHERE
$nd \in WRI \wedge$
$Booleanwr(nd) = TRUE \wedge$
$\neg\ (Totalpos\_message(nd) = TotalLONGMESSAGE(nd)) \wedge$
$Totalpos\_msg\_char(nd) = MAX\_BIT\_CHAR \wedge$
THEN
$out(nd) := TotalMESSAGE(nd)(Totalpos\_message(nd))(Totalpos\_msg\_char(nd))$ ||
$Totalpos\_message(nd) := Totalpos\_message(nd)+1$ ||
$Totalpos\_msg\_char(nd) := 0$ ||
$WRI := WRI - \{nd\}$ ||
$VER := VER \cup \{nd\}$;
$bus := \min(\{bus, out(nd)\})$
END

7.13 Introduction explicite du composant "bus"

Au point où nous sommes arrivés, il n'y a pas d'événement spécifique pour modéliser le bus. Celui-ci est modélisé par la substitution *min* dans la période d'écriture du cycle. Nous modélisons explicitement le bus comme un composant à part entière du système en le sortant des événements de la période d'écriture

et en créant un événement spécifique (appelé *bus_p*) modélisant le bus et en raffinant la variable *bus* par la variable *bbus*.

Cet événement se déclenche après la première période (après que tous les nœuds aient envoyé leurs bits) et avant la seconde (avant que le bus ne soit lu). Nous introduisons donc une troisième période dans le cycle qui prend place entre les deux autres. À ce titre, la variable *VER* est raffinée par les trois variables $E4$, $E2$ et $VER2$.

La machine que nous obtenons est donnée dans l'annexe F. Nous donnons ci-dessous l'invariant de collage et l'événement *bus_p*.

$$
\begin{array}{l}
bbus \in BIT \wedge \\
\\
E4 \subseteq ND - VER \wedge \\
(E4 \neq \varnothing \Rightarrow WRI = \varnothing) \\
(E4 \cup WRI \cup VER = ND) \\
\\
E2 \subseteq ND \wedge \\
E2 \subseteq VER \wedge \\
VER2 \subseteq VER \wedge \\
\\
(VER2 \neq \varnothing => WRI = \varnothing) \wedge \\
((WRI \cup E2) \neq \varnothing \Rightarrow (WRI \cup E2) = ND) \wedge \\
((VER2 \cup E4) \neq \varnothing \Rightarrow (VER2 \cup E4) = ND) \wedge \\
(WRI \cup E2 \cup VER2 \cup E4 = ND) \wedge \\
\\
(E2 = \varnothing \ \wedge VER \neq \varnothing \Rightarrow bbus = min(out[ND])) \\
(WRI = \varnothing \ \wedge VER \neq \varnothing \Rightarrow bus = min(out[ND]))
\end{array}
$$

bus_p =
WHEN
$WRI = \varnothing \wedge E2 \neq \varnothing$
THEN
$E2 := \varnothing \parallel$
$VER2 := ND \parallel$
$bbus := \min(out[ND])$
END

7.14 Statistiques sur les preuves

Chaque pas de raffinement est justifié formellement par des preuves. Les obligations de preuve sont générées automatiquement par les outils. Certaines des preuves sont prouvées automatiquement (48% dans cette étude de cas) et d'autres nécessitent une interaction avec l'utilisateur. Quelques statistiques sur les preuves sont résumées dans le tableau de la figure 7.6.

Deux des raffinements (le 6ème et le 10ème) génèrent plus d'obligations de preuve que les autres. Le premier est le raffinement où l'ordonnancement est ajouté. Le deuxième est un raffinement où les gardes de deux événements sont transformées : ce développement modélise un protocole distribué, modélisé d'abord en utilisant un état global, ce 10ème raffinement distribue concrètement le protocole.

Cette modélisation a permis de montrer formellement que le protocole résout correctement les conflits d'accès au bus et qu'il était possible de l'implanter de façon à ce que le système ne soit jamais totalement surchargé, c'est-à-dire qu'il y a toujours des composants qui peuvent envoyer leur message. Nous avons également montré qu'en cas de conflit d'accès au bus, le choix du composant qui obtient le bus est déterministe, ce qui signifie que le protocole induit une relation de priorité entre les composants.

Le principe de raffinement permet de modéliser le système pas à pas en introduisant les détails au fur et à mesure. Beaucoup de preuves peuvent ainsi être faites à un niveau abstrait. Il est beaucoup plus simple de faire des preuves sur un système contenant peu de variables et en manipulant des données abstraites

FIG. 7.6 – (1) Nombre de commandes du prouveur interactif utilisées pour faire l'ensemble preuves qui n'ont pas été déchargées automatiquement. (2) Nombre d'appels au prouveur automatique dans les preuves interactives. (3) nombre moyen de commandes (y compris appels au prouveur automatique) utilisées pour les preuves.

Raffinement	nombre d'obligations de preuve	nombre de preuves interactives	nombre de preuves automatiques	taille script de preuve (1)	appels au prouveur automatique (2)	taille moyenne (3)
0	2	0	2			
1er	5	2	3	9	1	4,5
2ème	19	5	14	31	5	6,2
3ème	5	5	0	28	4	5,6
4ème	18	13	5	58	15	4,5
5ème	14	5	9	32	3	6,4
6ème	504	211	293	2660	744	12,6
7ème	64	50	14	354	78	7,1
8ème	36	28	8	150	46	5,4
9ème	52	30	22	225	51	7,5
10ème	380	176	204	1646	441	9,4
11ème	31	23	8	109	15	4,7
12ème	24	20	4	96	19	4,8
13ème	75	33	42	673	172	20,4
14ème	55	30	25	324	79	10,8
15ème	46	34	12	167	32	4,9
16ème	72	46	26	233	40	5
17ème	87	59	28	445	132	7,5
Total	1489	52%	48%			9,4

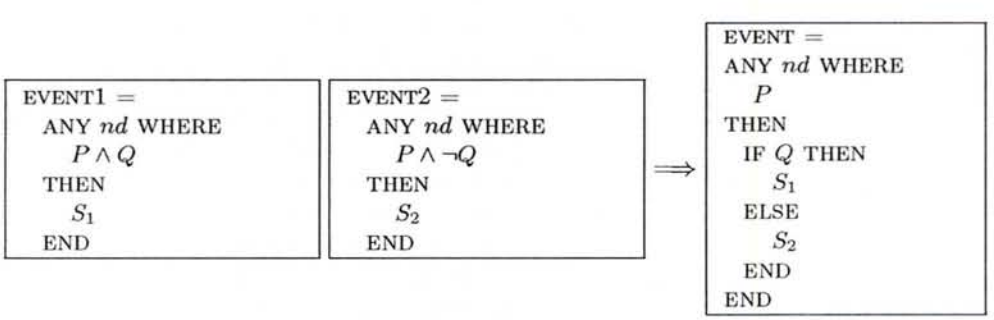
(ensembles, fonctions ...) que sur un système décrit au niveau *transfert de registres* en manipulant des bits.

Par ailleurs, la modélisation étape par étape par raffinements successifs permet à la personne en charge de la modélisation d'appréhender le système progressivement, en raffinant seulement une partie du système à chaque pas. Il est plus facile de trouver et corriger une erreur dans un tel système que dans une description bas niveau, noyé sous une montagne de détails. De plus, posséder un modèle abstrait du système permet une meilleure compréhension de son comportement. Le système aura une meilleure chance d'être compris par une autre personne que le développeur, ce qui facilitera son intégration et sa réutilisation.

De plus, le modèle implantable étant dérivé formellement du modèle du protocole, on est assuré que l'implantation correspond au modèle prouvé du protocole.

7.15 Regroupement des événements

Pour chaque période du cycle (période de lecture et période d'écriture), la collection d'événements prend en compte tous les cas possibles : la disjonction de toutes les gardes est *vraie*. De plus, la collection des gardes est une partition de tous les cas possibles, deux gardes ne peuvent pas être vraies au même moment. Plusieurs événements peuvent être regroupés en un seul événement en suivant le principe suivant :



La description d'un nœud doit être regroupée car la traduction vers d'autres formalismes a été définie pour un circuit regroupé en seul événement. Le regroupement a été effectué en utilisant un petit outil écrit

pour l'occasion qui travaille purement syntaxiquement. Les regroupements sont obtenus par de simples ensembles de règles qui spécifient la façon de faire le regroupement. Par exemple, la règle qui correspond à l'exemple donné ci-dessus est la suivante :

```
EVENT =
    BOTH EVENT1 AND EVENT2 FOR
      Q
    END
```

La règle ci-dessus signifie que les deux événements EVENT1 et EVENT2 sont regroupés en un seul événement appelé EVENT. On donne le prédicat Q qui doit figurer dans la garde du premier événement EVENT1 et $\neg Q$ doit figurer dans la garde du deuxième événement EVENT2.

Un regroupement est obtenu à partir du dernier raffinement (donné en annexe F) pour obtenir un système contenant quatre événements (*write*, *read*, *env* (inchangé)et *bus_p* (inchangé)) en appliquant les règles suivantes :

```
RECOMPOSITION
    recomp0
RECOMPOSES
    mm13
PRODUCE
    write, read, env, bus_p
RULES
ess1 = BOTH bus_write_first AND bus_write_nothing FOR TotalAT(nd)=0 END;
ess2 = BOTH send AND send_next_char FOR Totalpos_msg_char(nd) = MAX_BIT_CHAR END;
ess3 = BOTH sender_nothing AND ess2 FOR Totalpos_message(nd) = TotalLONGMESSAGE(nd) END;
ess3bis = BOTH bus_nothing AND ess3 FOR Booleanwr(nd) = TRUE END;
ess3ter = BOTH bus_write AND ess3bis FOR BooleanCP(nd) = TRUE END;
write = BOTH ess1 AND ess3ter FOR IsInDomAT(nd) = TRUE END;
ess4 = BOTH rise_count AND switch_ko FOR bus=1 END;
ess5 = BOTH switch_ok AND ess4 FOR TotalAT(nd)=0 END;
ess6 = BOTH check_ok AND check_ko FOR (MID(nd))(pos(nd)) = bus END;
ess7 = BOTH take AND ess6 FOR pos(nd)=MAX_BIT_CHAR END;
ess8 = BOTH free AND sender_w FOR fc(nd)=1 END;
ess9 = BOTH ess8 AND sender FOR Totalpos_message(nd) = TotalLONGMESSAGE(nd) END;
ess10 = BOTH rise AND nothing FOR fc(nd)=0 ∧ bus=1 ∧ in_wri(nd)= TRUE END;
ess11 = BOTH ess10 AND ess9 FOR Booleanwr(nd) = TRUE END;
ess12 = BOTH ess11 AND ess7 FOR BooleanCP(nd) = TRUE END;
read = BOTH ess12 AND ess5 FOR IsInDomAT(nd) = TRUE END
END
```

Les événements *env* et *bus_p* sont inchangés, l'événement *write* est l'événement obtenu par regroupement des événements de la période d'écriture et l'événement *read* est l'événement obtenu par regroupement des événements de la période de lecture. Les événements *read* et *write* sont montrés ci-dessous. Cette étape de regroupement constitue un raffinement en lui-même que nous ne faisons pas par la preuve mais par un outil mécanique dont la sortie est supposée correcte par construction. Cet outil est mécanique mais pas automatique puisqu'il faut lui fournir les règles à appliquer. Le fait qu'il réussi à appliquer les règles prouve que le résultat est correct.

```
write = ANY nd WHERE
      nd ∈ WRI
    THEN
      IF IsInDomAT(nd) = TRUE THEN
        IF TotalAT(nd) = 0 THEN
          out(nd) :=0
        ELSE
          out(nd) :=1
        END
```

```
      ELSE
        IF BooleanCP(nd) = TRUE THEN
          pos(nd) :=pos(nd)+1 ||
          out(nd) :=MID(nd)(pos(nd)+1)
        ELSE
          IF Booleanwr(nd) = TRUE THEN
            IF Totalpos_message(nd) = TotalLONGMESSAGE(nd) THEN
              out(nd) :=1
            ELSE
              IF Totalpos_msg_char(nd) = MAX_BIT_CHAR THEN
                Totalpos_message(nd) :=Totalpos_message(nd)+1 ||
                Totalpos_msg_char(nd) :=0
              ELSE
                Totalpos_msg_char(nd) :=Totalpos_msg_char(nd)+1
              END ||
              out(nd) :=TotalMESSAGE(nd)(Totalpos_message(nd))(Totalpos_msg_char(nd))
            END
          ELSE
            out(nd) :=1
          END
        END
      END ||
      WRI :=WRI-{nd} ||
      E2 :=E2 ∪ {nd}
    END

read = ANY nd WHERE
    nd ∈ VER2 ∧
    E2 = ∅
  THEN
    IF IsInDomAT(nd) = TRUE THEN
      IF TotalAT(nd) = 0 THEN
        BooleanCP(nd) :=TRUE ||
        IsInDomAT(nd) :=FALSE
      ELSE
        IF bbus = 1 THEN
          TotalAT(nd) :=TotalAT(nd)-1
        ELSE
          IsInDomAT(nd) :=FALSE
        END
      END
    ELSE
      IF BooleanCP(nd) = TRUE THEN
        IF pos(nd) = MAX_BIT_CHAR THEN
          Booleanwr(nd) :=TRUE ||
          BooleanCP(nd) :=FALSE ||
          pos(nd) :=0 ||
          Totalpos_message(nd) :=0 ||
          Totalpos_msg_char(nd) :=0 ||
          TotalLONGMESSAGE :=in_lmsg ||
          TotalMESSAGE :=in_msg
        ELSE
          IF ¬ (MID(nd)(pos(nd)) = bbus) THEN
            BooleanCP(nd) :=FALSE ||
            pos(nd) :=0
          END
        END
      ELSE
```

IF $Booleanwr(nd) = TRUE$ THEN
IF $Totalpos\_message(nd) = TotalLONGMESSAGE(nd) \wedge fc(nd) = 1$ THEN
$Booleanwr(nd) := FALSE$
END
ELSE
IF $fc(nd) = 0 \wedge bbus = 1 \wedge in\_wri(nd) = TRUE$ THEN
$TotalAT(nd) := in\_tt(nd)$ ||
$IsInDomAT(nd) := TRUE$ ||
$pos(nd) := 0$
END
END
END
END ||
$VER2 := VER2 - \{nd\}$ ||
$E4 := E4 \cup \{nd\}$ ||
$fc(nd) := (1 - bbus) \times COUNT\_FREE + bbus \times max(\{fc(nd) - 1, 0\})$
END

Les événements *bus_p* et *env* n'ont aucun effet sur l'état d'un nœud. Du point de vue d'un nœud, ils se comportent comme *skip*. Attention, ceci ne signifie pas qu'on puisse remplacer ces événements par *skip* dans le modèle, mais simplement qu'ils ne manipulent pas directement les variables d'état des nœuds. Ces événements peuvent donc être traduits séparément.

L'ordonnancement modélisé par les variables *WRI*, *VER2*, *E2* et *E4* est en fait un ordre sur le chemin des données du circuit : le calcul effectué par l'événement *read* dépend du calcul effectué par l'événement *write*. C'est exactement la sémantique du point-virgule en B. On peut supprimer l'ordonnancement en composant les événements *write* et *read* en une substitution séquentielle. Attention, il ne s'agit pas là d'un regroupement réalisant un raffinement du modèle. En effet, dans le modèle l'événement *bus_p* doit se tenir entre ces deux événements ; en composant ces deux événements en une seule substitution, aucun autre événement ne peut se déclencher en même temps, ce n'est pas un raffinement du modèle à proprement parler. Il s'agit d'un raffinement si on supprime l'événement *bus_p* ce qui est le cas puisque nous avons expliqué que celui-ci pouvait être traduit à part. Ainsi, nous pouvons composer les deux événements *read* and *write* en un seul événement *circuit* comme montré ci-dessous, où *sl_write* et *sl_read* sont les substitutions des événements respectifs *write* et *read* dans lesquelles toutes les références aux variables d'ordonnancement ont été retirées.

circuit =
ANY *nd* WHERE
$nd \in ND$
THEN
sl_write ; *sl_read*
END

7.16 Traduction

Nous présentons dans cette section le code BHDL obtenu après le développement que nous avons expliqué dans les sections précédentes. Nous donnons également un morceau du code VHDL généré automatiquement par l'outil développé par KeesDA. Dans nos modèles, le signal d'horloge n'est pas modélisé explicitement. L'hypothèse implicite qui est faite est que tous les composants sont gouvernés par la même horloge.

7.16.1 Code BHDL

Avant de traduire le modèle du nœud vers un langage de description de circuits (ici VHDL), nous avons besoin de l'adapter :

- Nous ne traduisons qu'un seul nœud (pas l'ensemble du système) [3], donc

  - toute variable d'état (fonction totale) est remplacée par une simple variable. Par exemple, $BooleanCP \in ND \rightarrow BOOL$ est remplacée par $BooleanCP \in Bool$,

  - les substitutions sont adaptées en conséquence ; en fait toute référence à *nd* est retirée.

  Pour ce système qui modélise un composant (disons M1), nous pouvons obtenir un raffinement (disons M) du système original en "important" le modèle d'un composant. De la même façon pour un système contenant plusieurs composants, en faisant plusieurs importations : chaque variable $v \in E$ dans M1 est transformée en $v \in ND \rightarrow E$ où ND est l'ensemble des composants "importés".

- L'événement *env* n'est pas traduit, seulement l'événement *circuit*.

- Les variables *bus* et *out* sont respectivement renommées *bbus* et *bout* car elles sont des mots réservés en VHDL.

- Puisque le type *BIT* existe déjà en VHDL (et qu'il n'est pas exactement le même type), nous remplaçons le type *BIT* par le type *BOOL*.

- La substitution

$$fc(nd) := (1\text{-}bus) \times COUNT\_FREE + bus \times max(\{fc(nd)\text{-}1, 0\})$$

  est répétée dans tous les sous cas dans la partie "read". Elle peut être factorisée et mise en fin de la partie "read".

  De plus, pour obtenir une expression plus praticable, et en accord avec le fait que maintenant le type de la variable *bbus* n'est plus *BIT* mais *BOOL*, on peut réécrire la substitution de cette façon :

```
IF bbus = FALSE
THEN
   fc := COUNT_FREE
ELSE
   IF fc ≠ 0
   THEN
      fc := (fc -1) mod 16
   END
END
```

- Les opérations arithmétiques sur les entiers non signés sont faites modulo. Cela est nécessaire pour éviter les erreurs dues à des "overflows" lors de calculs intermédiaires. Par exemple, quand on utilise la substitution IF en B, on prouve la correction pour le cas qui va se produire. Hors dans un circuit physique, même la partie "inutile" est calculée (bien que inutilisée). Le choix d'utiliser des opérations modulo est en relation avec le résultat obtenu par un additionneur physique. Comme les modulo n'étaient pas présents dans le modèle B, cela signifie que nous avons prouvé que le résultat "utile" était toujours correct, c'est-à-dire dans les bornes du type, donc insensible au modulo.

- Tous les sous types d'entiers sont remplacés par des types prédéfinis dans une machine appelée BHDL. Nous utilisons ici deux types : *UINT4* et *UINT5*, nombres entiers non signés sur quatre et cinq bits.

- Les paramètres sont fixés : période de latence, longueur maximale d'un message et longueur d'un caractère. Ceci est fait dans une machine séparée :

[3] En fait, un système peut contenir plusieurs composants qui seront décrits dans des machines BHDL séparées, et ensuite traduits séparément

MACHINE
PARAM
SEES
BHDL
CONSTANTS
CHAR, MAX_BIT_CHAR, COUNT_FREE,LONG_MAX_MESSAGE,TMESSAGE
PROPERTIES
$MAX\_BIT\_CHAR = 9 \wedge \quad COUNT\_FREE = 11 \wedge \quad LONG\_MAX\_MESSAGE = 5 \wedge$

$INTCHAR = 0 \,..\, MAX\_BIT\_CHAR \wedge CHAR = INTCHAR \rightarrow BOOL \wedge$
$INTMESSAGE = 0 \,..\, LONG\_MAX\_MESSAGE\text{-}1 \wedge TMESSAGE = INTMESSAGE \rightarrow CHAR$
END

Ci-dessous le code BHDL obtenu après toutes les adaptations énumérées ci-dessus.

REFINEMENT

circuit

REFINES

NOTHING

SEES

BHDL, PARAM

INPUTS

bbus,

in_tt ,

in_lmsg,

in_msg,

in_wri,

MID

OUTPUTS

bout

VARIABLES

Booleanwr, *TotalAT*,
IsInDomAT, *BooleanCP*,
pos, *TotalLONGMESSAGE*,
Totalpos_message, *Totalpos_msg_char*,
TotalMESSAGE, *fc*

INVARIANT

$pos \in UINT4 \wedge$ $fc \in UINT4 \wedge$
$TotalAT \in UINT5 \wedge$ $IsInDomAT \in BOOL \wedge$
$Booleanwr \in BOOL \wedge$ $BooleanCP \in BOOL \wedge$
$TotalLONGMESSAGE \in UINT4 \wedge$ $TotalMESSAGE \in TMESSAGE \wedge$
$Totalpos\_message \in UINT4 \wedge$ $Totalpos\_msg\_char \in UINT4 \wedge$

$bbus \in BOOL \wedge$ $in\_wri \in BOOL \wedge$
$in\_tt \in UINT5 \wedge$ $in\_lmsg \in UINT4 \wedge$
$in\_msg \in TMESSAGE \wedge$ $MID \in CHAR \wedge$

$bout \in BOOL$

INITIALISATION

```
  pos := 0 ||                          fc := 0 ||
  TotalAT := 0 ||                      IsInDomAT := FALSE ||
  Booleanwr := FALSE ||                BooleanCP := FALSE ||
  TotalLONGMESSAGE :∈ UINT4 ||         TotalMESSAGE :∈ TMESSAGE ||
  Totalpos_message :∈ UINT4 ||         Totalpos_msg_char :∈ UINT4 ||
  bbus :∈ BOOL ||                      in_wri :∈ BOOL ||
  in_tt :∈ UINT5 ||                    in_msg :∈ TMESSAGE ||
  in_lmsg :∈ UINT4 ||                  MID :∈ CHAR ||
bout :∈ BOOL

OPERATIONS
circuit =
BEGIN

      BEGIN
        IF IsInDomAT = TRUE THEN
          IF TotalAT = 0 THEN
            bout :=FALSE
          ELSE
            bout :=TRUE
          END
        ELSE
          IF BooleanCP = TRUE THEN
            pos :=pos+1 ||
            bout :=MID((pos+1) mod 16)
          ELSE
            IF Booleanwr = TRUE THEN
              IF Totalpos_message = TotalLONGMESSAGE THEN
                bout :=TRUE
              ELSE
                IF Totalpos_msg_char = MAX_BIT_CHAR THEN
                  Totalpos_message :=(Totalpos_message+1) mod 16 ||
                  Totalpos_msg_char :=0
                ELSE
                  Totalpos_msg_char :=(Totalpos_msg_char+1) mod 16
                END ||
                bout :=TotalMESSAGE(Totalpos_message)(Totalpos_msg_char)
              END
            ELSE
              bout :=TRUE
            END
          END
        END
      END

;
      BEGIN
        IF IsInDomAT = TRUE THEN
          IF TotalAT = 0 THEN
            BooleanCP :=TRUE ||
            IsInDomAT :=FALSE
          ELSE
            IF bbus = TRUE THEN
```

```
            TotalAT :=(TotalAT-1) mod 16
          ELSE
            IsInDomAT :=FALSE
          END
        END
      ELSE
        IF BooleanCP = TRUE THEN
          IF pos = MAX_BIT_CHAR THEN
            Booleanwr :=TRUE ||
            BooleanCP :=FALSE ||
            pos :=0 ||
            Totalpos_message :=0 ||
            Totalpos_msg_char :=0 ||
            TotalLONGMESSAGE :=in_lmsg ||
            TotalMESSAGE :=in_msg
          ELSE
            IF ¬ (MID(pos) = bbus) THEN
              BooleanCP :=FALSE ||
              pos :=0
            END
          END
        ELSE
          IF Booleanwr = TRUE THEN
            IF Totalpos_message = TotalLONGMESSAGE ∧ fc = 1 THEN
              Booleanwr :=FALSE
            END
          ELSE
            IF fc = 0 ∧ bbus = TRUE ∧ in_wri = TRUE THEN
              TotalAT :=in_tt ||
              IsInDomAT :=TRUE ||
              pos :=0
            END
          END
        END
      END

      ||

    IF bbus = FALSE
    THEN
      fc := COUNT_FREE
    ELSE
      IF fc ≠ 0
      THEN
        fc := (fc -1) mod 16
      END
    END
    END

END

END
```

7.16.2 Code VHDL

Nous devons traduire les deux machines *PARAM* et *circuit*. Ces deux traductions sont faites automatiquement vers VHDL en utilisant le traducteur développé par KeesDA.

Traduction de PARAM

La traduction de la machine *PARAM* donne le résultat suivant. Il s'agit d'une suite de définitions de types et de constantes.

```
use BHDL.all;

package PARAM is

        constant MAX_BIT_CHAR : INTEGER := 9;
        constant COUNT_FREE : INTEGER := 11;
        constant LONG_MAX_MESSAGE : INTEGER := 5;
        subtype INTCHAR is INTEGER range 0 to MAX_BIT_CHAR;
        type CHAR is array (INTCHAR) of BOOL;
        subtype INTMESSAGE is INTEGER range 0 to (LONG_MAX_MESSAGE - 1);
        type TMESSAGE is array (INTMESSAGE) of CHAR;

end PARAM;
```

Traduction du circuit

Pour des raisons de confidentialité, mais aussi parce que le code VHDL obtenu automatiquement est relativement illisible, nous ne donnons pas l'intégralité de ce code VHDL obtenu automatiquement à partir du code BHDL par le traducteur développé par KeesDA. À titre d'exemple, nous donnons la partie du code VHDL correspondant à la partie suivante du code BHDL :

```
IF Totalpos_message = TotalLONGMESSAGE THEN
  bout :=TRUE
ELSE
  IF Totalpos_msg_char = MAX_BIT_CHAR THEN
    Totalpos_message :=(Totalpos_message+1) mod 16 ||
    Totalpos_msg_char :=0
  ELSE
    Totalpos_msg_char :=(Totalpos_msg_char+1) mod 16
  END ||
  bout :=TotalMESSAGE(Totalpos_message)(Totalpos_msg_char)
END
```

Le code VHDL obtenu à partir du code BHDL ci-dessus est le suivant :

```
CD_22 : block
          signal bout_23 : BOOL;
          signal bout_25 : BOOL;
          signal Totalpos_msg_char_25 : UINT4;
          signal Totalpos_message_25 : UINT4;
          signal gd_23 : BOOL;
        begin
          gd_23 <= (Totalpos_message_0=TotalLONGMESSAGE_0);
          bout_23 <= true;

          CD_27 : block
            signal Totalpos_msg_char_28 : UINT4;
            signal Totalpos_message_28 : UINT4;
```

```
    signal Totalpos_msg_char_32 : UINT4;
    signal gd_28 : BOOL;
  begin
    gd_28 <= (Totalpos_msg_char_0=MAX_BIT_CHAR);
    Totalpos_message_28 <= ((Totalpos_message_0 + 1) mod 16);
    Totalpos_msg_char_28 <= 0;
    Totalpos_msg_char_32 <= ((Totalpos_msg_char_0 + 1) mod 16);
    Totalpos_msg_char_25 <= Totalpos_msg_char_28 when gd_28 else Totalpos_msg_char_32;
    Totalpos_message_25 <= Totalpos_message_28 when gd_28 else Totalpos_message_0;
  end block;

  bout_25 <= TotalMESSAGE_0(Totalpos_message_0)(Totalpos_msg_char_0);
  bout_21 <= bout_23 when gd_23 else bout_25;
  Totalpos_msg_char_21 <= Totalpos_msg_char_0 when gd_23 else Totalpos_msg_char_25;
  Totalpos_message_21 <= Totalpos_message_0 when gd_23 else Totalpos_message_25;
end block;
```

7.17 Simulation

La section précédente a montré comment obtenir le code VHDL d'un circuit à partir de sa description en B. Le code ainsi généré a été utilisé pour faire de la simulation. Nous présentons dans cette section la simulation d'un système composé de deux nœuds qui vont être en compétition pour obtenir l'accès au bus.

7.17.1 Le système

Le système est composé d'une horloge, d'un signal de réinitialisation asynchrone (qui n'apparaît qu'en début de simulation), du bus et de deux nœuds. Chaque nœud a cinq entrées (plus les signaux d'horloge et de réinitialisation asynchrone) et une sortie.

- L'entrée *in_tt* est le temps d'attente (exprimé en nombre de cycles d'horloge) que le nœud doit respecter avant de tenter d'accéder au bus.
- Les entrées *in_lmsg* et *in_msg* sont respectivement la longueur du message et le message lui-même.
- L'entrée *in_wri* est le signal permettant de dire au contrôleur s'il doit ou non envoyer un message.
- L'entrée *MID* est le numéro d'identification du nœud, il s'agit d'une entrée constante.
- La sortie *bout* est la seule sortie d'un nœud, elle est utilisée pour envoyer des bits vers le bus ; elle est connectée au bus.

Les entrées et la sorties sont indicées par 1 ou 2 selon le nœud auquel ils correspondent (nœud 1 et nœud 2).

7.17.2 Simulation

Dans la simulation représentée par les chronogrammes ci-dessous (obtenus grâce au logiciel de simulation ActiveHDL [11]), deux nœuds tentent d'accéder au bus au même moment. Il y a donc une compétition, et finalement seulement l'un des deux peut effectivement envoyer son message. Après qu'une période de latence ait été observée, l'autre nœud peut enfin envoyer son propre message (après sa période d'attente).

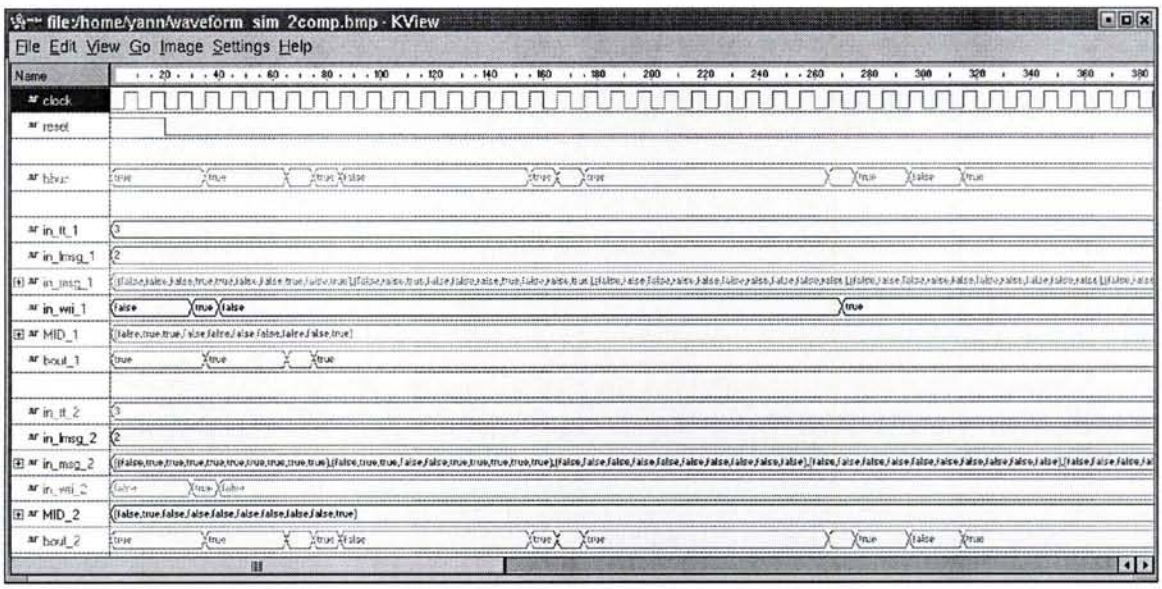
Jusqu'à 20ns, rien ne se passe sur le bus car le signal d'initialisation asynchrone est actif.

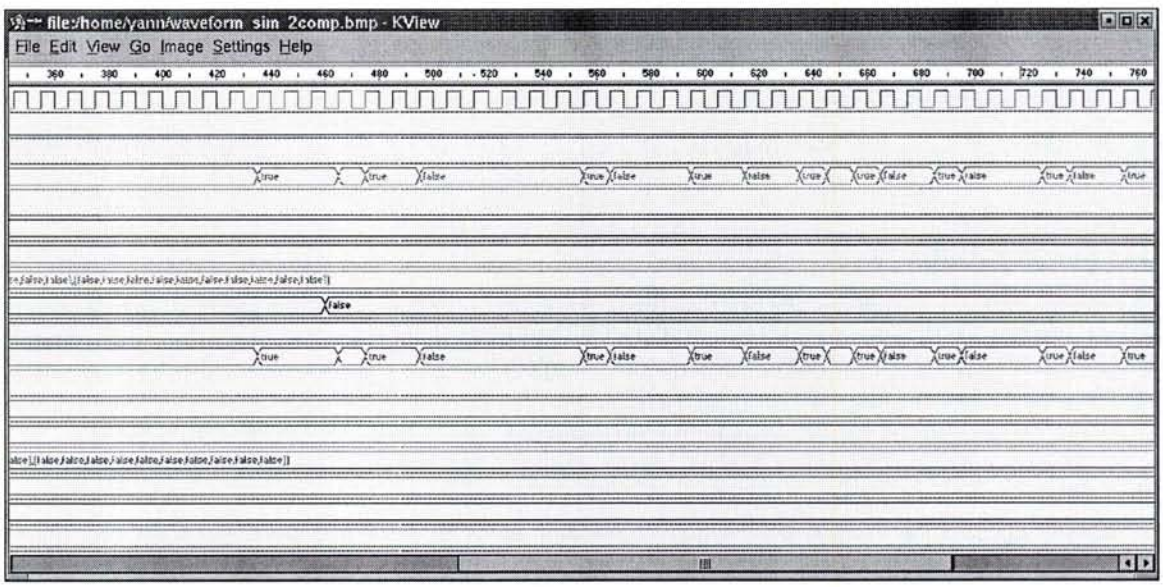
Au front montant de l'horloge à 35ns, les deux nœuds ont leurs entrées *in_wri* positionnées à *true* et commencent donc leurs phases d'attente.

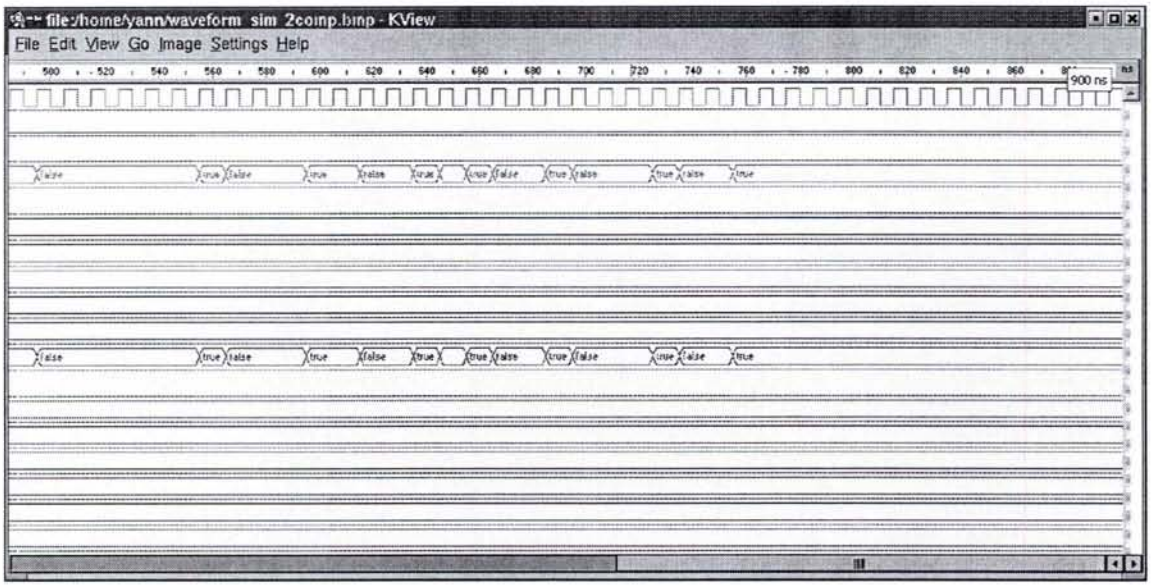
Trois cycles plus tard, les deux nœuds ont terminé leurs phases d'attente (ils ont tous les deux leurs entrées *in_tt* positionnées à 3) et commencent à envoyer leurs MIDs.

Les deux premiers bits de leurs MIDs sont les mêmes, il n'y a donc pas de collision sur le bus. Lorsque les deux nœuds écrivent leurs troisièmes bits de leurs MIDs respectifs, il y a une collision car leurs MIDs n'ont pas la même valeur à cette position, "false" domine le bus. Le premier nœud (qui a envoyé "true") détecte la collision et quitte la compétition (il envoie continuellement "true" sur le bus). Le deuxième nœud ne détecte pas de collision et continue à envoyer le reste de son MID et ensuite son message (le chronogramme du bus est identique au chronogramme de la sortie du deuxième nœud.

Une période de latence est observée entre 315ns et 425ns. Après avoir observé cette période de latence, le premier nœud , qui a toujours son entrée *in_wri* positionnée à "true", commence sa période d'attente (3 cycles). Il n'y a aucun conflit et il peut donc envoyer son message complètement.







7.17.3 Remarques

Dans cette simulation nous n'avons pas modélisé un comportement complexe pour retenter l'accès au bus après un échec (comme c'est le cas pour le premier nœud dans cette simulation). Les valeurs pour les entrées *in_wri* et *in_tt* ont été choisies de façon à ce que les chronogrammes de simulation ne soient pas trop difficiles à lire.

De plus, dans cette simulation le système n'est composé que de deux nœuds et une seule compétition apparaît. Si on augmente le nombre de nœuds, de collisions et qu'on modélise le calcul des nouvelles valeurs pour les entrées *in_tt* après qu'une collision soit détectée, la simulation deviendrait très complexe. De la même façon, choisir des cas de test et la vérification des résultats de simulation est un travail difficile.

Pour la vérification de la fonctionnalité d'un contrôleur de bus, la vérification formelle semble donc une meilleure solution que la simulation, car il n'est pas nécessaire de faire l'effort de comprendre des résultats complexes de simulation.

Si on observe bien les chronogrammes du bus et des sorties des nœuds au temps 35ns, on observe un curieux phénomène : les valeurs passent de "true" à "true". Autrement dit il n'y a pas fonctionnellement de changement de la valeur mais il y a une petite instabilité du signal. C'est ce qu'on appelle un *glitch*. Pour éviter un tel phénomène, les ingénieurs de conception de circuits recommandent de faire en sorte que la sortie d'un circuit (en l'occurrence la sortie d'un nœud) soit toujours directement la sortie d'un registre interne : de cette façon le signal est beaucoup plus stable. Il s'agit typiquement d'un problème de spécification, il devrait être écrit dans la spécification si on souhaite qu'un signal soit stable pendant toute la durée d'un cycle. Ce problème a pu être corrigé facilement. Nous ne montrons pas le résultat ici car il n'a pas d'intérêt sinon de montrer la disparition du glitch.

Ce problème rencontré lors de cette étude de cas permet de mettre l'accent sur le fait qu'une méthode formelle n'a pas pour vocation à supprimer totalement la simulation, mais au contraire à lui donner tout son sens en la réservant aux vérifications de niveau physique. La méthode formelle peut valider la fonctionnalité d'un composant alors que la simulation peut servir à vérifier les performances par exemple.

7.17.4 Synthèse

Le code VHDL ainsi obtenu a été passé dans un synthétiseur. Ce travail a été fait par des ingénieurs de la société Evatronix en Pologne. La synthèse elle-même n'a pas posé de problème. Nous ne donnons pas ici la sortie complète de l'outil de synthèse car elle est illisible pour un non spécialiste. Nous en donnons une petite partie ci dessous. Le premier tableau calcul la longueur du chemin critique, c'est-à-dire le

chemin qui prend le plus de temps à être parcouru par les signaux électriques. Ceci permet de déterminer la fréquence maximale pour l'horloge.

```
Point                                                    Incr       Path
  ---------------------------------------------------------------------------
  clock clock (rise edge)                                0.00       0.00
  clock network delay (ideal)                            0.00       0.00
  pos_0_reg[0]/CK (DFFRPQ1)                              0.00       0.00 r
  pos_0_reg[0]/Q (DFFRPQ1)                               0.30       0.30 f
  U572/Z (INVD1)                                         0.13       0.43 r
  U441/Z (OR3D1)                                         0.12       0.55 r
  U523/Z (OR2D1)                                         0.11       0.66 r
  U580/Z (OR2D4)                                         1.25       1.91 r
  TotalLONGMESSAGE_0_reg[0]/CEB (DFEPQ1)                 0.00       1.92 r
  data arrival time                                                 1.92

  clock clock (rise edge)                               10.00      10.00
  clock network delay (ideal)                            0.00      10.00
  TotalLONGMESSAGE_0_reg[0]/CK (DFEPQ1)                  0.00      10.00 r
  library setup time                                    -0.23       9.77
  data required time                                                9.77
  ---------------------------------------------------------------------------
  data required time                                                9.77
  data arrival time                                                -1.92
  ---------------------------------------------------------------------------
  slack (MET)                                                       7.85
```

Ci-dessous nous donnons une autre partie de la sortie de l'outil de synthèse. Elle donne le nombre de ports d'entrée/sortie (286), le nombre de portes logiques dans le circuit (677), le nombre de connexions (977) ainsi que quelques informations sur la surface du circuit.

```
Number of ports:              286
Number of nets:               977
Number of cells:              677
Number of references:          35

Combinational area:       9737.113281 (square micrometers)
Noncombinational area:    30353.806641
Net Interconnect area:      undefined  (Wire load has zero net area)

Total cell area:          40090.832031 (? 3286 logic gates)
Total area:                 undefined
```

La seule remarque que nous puissions faire sur ce résultat est le nombre important de ports d'entrée/sortie du circuit. Cela est en fait dû au message qui est pris dans son entier d'un seul coup (256 entrées).

Pour corriger cela, nous avons poursuivi le raffinement de façon à faire en sorte que le message soit obtenu bit par bit lorsque cela est nécessaire. Pour cela nous avons introduit un petit protocole entre le nœud et le composant englobant qui fournit le message.

7.18 Conclusion

Nous avons présenté dans ce chapitre la modélisation en utilisant la méthode B du protocole standard SAE J1708 défini dans [83]. La technique du raffinement a permis d'introduire les différents concepts et les phases du protocole les uns après les autres, en plusieurs étapes. Ceci permet une meilleure compréhension des modèles et des preuves de correction plus simples que si le modèle final avait été écrit en un coup. L'utilisation du raffinement a aussi permis de représenter le système de façon abstraite, facilitant ainsi l'écriture de propriétés au niveau système, ce qui devient très difficile à faire lorsque qu'on se trouve au niveau de l'implantation.

Cette étude de cas nous a permis de présenter comment spécifier un modèle de circuit synchrone en B jusqu'au niveau où le cycle d'horloge lui-même est pris en compte. Le modèle cyclique est obtenu en utilisant des variables supplémentaires pour ordonnancer les composants. Nous avons vu que cet ordonnancement par des variables se traduit en fait par un ordonnancement du chemin des données du circuit modélisé. L'ordonnancement autorise les communications entre les composants (par les entrées et les sorties).

Nous avons présenté les étapes de raffinement les unes après les autres. Chaque raffinement ajoute des détails au modèle du système. Ce développement incrémental permet une meilleure compréhension et donc un dialogue plus facile avec les donneurs d'ordre au niveau de la spécification. Les modèles les plus abstraits constituent une spécification formelle du système, et c'est une partie non négligeable du développement car il s'agit du passage de la spécification en langue naturelle (avec tous les défauts que cela comporte) à la spécification formelle. C'est pendant cette étape que la spécification initiale est interprétée. Par ailleurs, le développement en plusieurs étapes successives évite au modeleur de crouler sous une montagne de détails.

Nous avons également montré comment modéliser le système de façon à ce qu'il puisse comporter un nombre arbitraire de composants identiques sans faire exploser la taille du modèle en "factorisant" le modèle. Les composants sont indépendants : leurs états sont représentés de telle façon qu'ils puissent être manipulés séparément. Nous avons également montré comment modéliser l'envoi de bits sur le bus, ce qui est une partie analogique du système.

La méthode pour regrouper les événements du système a été présentée. Ce regroupement prend en compte l'ordonnancement et il est nécessaire pour obtenir un modèle conforme au langage d'entrée du traducteur développé par KeesDA.

La machine BHDL que nous avons obtenu après regroupement a été traduite automatiquement en VHDL en utilisant le traducteur. Nous avons exposé une petite partie du code obtenu par traduction. Ce code VHDL a ensuite été utilisé pour faire une simulation du système dont nous avons présenté les chronogrammes de simulation. Cette simulation, d'un système relativement simple pour faciliter la compréhension, a concerné un système composé de deux nœuds. Cet exemple illustre qu'il est difficile de vérifier les résultats de simulation, et donc l'intérêt d'une méthode formelle permettant le développement de circuits fonctionnellement corrects par construction. Cependant nous avons expliqué que la simulation ne doit pas être abandonnée car la vérification formelle ne s'adresse qu'à la fonctionnalité du système ; la simulation peut être utilisée pour vérifier d'autres facteurs, comme les performances temporelles.

Pour résumé, le modèle a été développé étape par étape en utilisant la technique de raffinement. Ensuite les événements ont été regroupés pour obtenir le code BHDL d'un nœud. Ce code BHDL a été traduit automatiquement vers VHDL en utilisant le traducteur de KeesDA. Ce code VHDL a été utilisé pour faire des simulations du système.

Dans notre première tentative de regroupement nous avons trouvé des glitches sur les signaux de sortie des nœuds. Ceci a été corrigé en faisant un autre regroupement en suivant les recommandations données par les ingénieurs de Volvo.

Nous sommes allés un peu plus loin en faisant une synthèse du système. Les ingénieurs ayant réalisé cette synthèse nous ont indiqués que de récupérer le message d'un seul coup en entrée représentait trop de fils d'entrée (256 entrées uniquement pour le message). Nous avons donc poursuivi le raffinement de façon à ce que le message soit obtenu bit par bit, en introduisant un petit protocole entre le nœud et le composant associé qui fournit le message. Ceci montre qu'une fois une implantation obtenue on peut si nécessaire continuer à raffiner vers une autre implantation selon les recommandations données par les experts en systèmes électroniques, sans avoir à refaire tout le développement.

Troisième partie

BHDL : sémantique et traduction

Chapitre 8

Le langage BHDL

L'objectif de ce chapitre est de présenter le langage BHDL, que nous considérons comme le niveau implantable RTL pour le langage B. Nous présentons la syntaxe que nous illustrons sur l'exemple d'un compteur.

Nos travaux ont débuté par l'étude de cas du contrôleur de bus série SAEJ1708, étude de cas présentée dans le chapitre précédent. Cette étude de cas nous a permis de définir des règles de modélisation présentée dans le chapitre 5. Dans le même temps, nous avons défini le langage BHDL et sa traduction vers VHDL et SystemC et enfin nous avons défini la sémantique du langage BHDL. La définition du langage BHDL et de sa traduction a été faite en collaboration avec Stefan Hallerstede, dans le cadre de la société KeesDA. Pendant la durée de nos travaux, le langage de la méthode B a évolué, partant du B "classique" pour aller vers le langage B événementiel. Le langage BHDL et sa sémantique sont basés sur le langage B classique. C'est pourquoi nous utilisons dans le langage BHDL (et parfois dans nos développements) la composition séquentielle (notée par un point virgule) ainsi que la composition conditionnelle (IF) qui n'existent plus dans le langage B événementiel. Nous avons cependant voulu développer nos modèles dans le langage B événementiel, c'est pourquoi nous avons utilisé les deux règles présentée dans la section 6.7 pour obtenir un modèle BHDL à partir d'un modèle B événementiel.

Le langage BHDL est un sous-ensemble du langage B et sa sémantique est également basée sur celle de B. Il y a cependant certaines différences. Deux nouvelles clauses sont ajoutées au langage pour les variables représentant les entrées et les sorties du système. Des conditions de bonne formation sur ces variables sont ajoutées : les entrées peuvent être seulement lues et les sorties seulement écrites. Ces conditions sont nécessaires pour que le modèle représente correctement un circuit électronique (les signaux électriques ne se propageant que dans un seul sens, les signaux d'entrée se propagent de l'extérieur vers l'intérieur du système (lecture) alors que les signaux de sortie se propagent de l'intérieur vers l'extérieur (écriture). Nous imposons également des restrictions plus fortes sur les types des expressions. Les types correspondent à des structures bien définies dans un circuit électronique (par exemple, un entier sur 4 bits correspond à une nappe de 4 fils) et les liaisons doivent être bien définies : une expression codée sur 5 bits ne peut pas a priori être affectée à un entier de 4 bits, à moins de définir explicitement la conversion.

On distingue deux types de machines BHDL : les machines déclaratives qui définissent des types et des constantes utilisés par d'autres machines, et les modèles BHDL définissant des circuits électroniques synchrones. Dans ce chapitre nous définissons d'abord le langage BHDL (les deux types de machines) et sa syntaxe (cf. annexe A pour la grammaire complète).

8.1 Le langage

Cette section introduit le langage BHDL. Nous définissons d'abord la grammaire des deux types de machines BHDL puis nous donnons deux exemples afin d'illustrer comment modéliser un circuit en BHDL et le réutiliser par importation.

8.1.1 Grammaire

Nous présentons ici les grammaires des machines BHDL. Il y deux types de machines : les machines déclaratives et les modèles BHDL. Une machine déclarative définit des types et des constantes et sont vues (en utilisant la clause SEES) par d'autres machines qui utilisent ces types et ces constantes. Les modèles BHDL définissent des circuits électroniques synchrones. Les modèles BHDL ne définissent pas de types ou de constantes, ceux-ci doivent être déclarés dans des machines déclaratives uniquement. Il est très utile de séparer les déclarations de types et de constantes des modèles eux-mêmes : d'une part, ceci permet à plusieurs modèles d'utiliser les mêmes machines déclaratives (et donc d'être certain d'utiliser les mêmes définitions), et, d'autre part, ceci permet au modèle de ne pas être dépendant des valeurs des constantes et des types. Il est possible de modifier les définitions des constantes et des types sans modifier le modèle lui-même. Les modifications qui peuvent intervenir dans ces machines doivent cependant être en accord avec les contraintes exprimées sur ces constantes (dans la clause PROPERTIES) dans les modèles prouvés de façon à ce que la correction du modèle soit maintenue.

Modèles BHDL

Un modèle BHDL est organisé en neuf clauses, la figure 8.1 donne une vue générale d'un modèle BHDL. Cinq des clauses sont identiques aux clauses d'une machine traditionnelle : la clause MACHINE donne le nom de la machine, la clause SEES donne la liste des machines déclaratives qui sont vues par le modèle, la clause VARIABLES donne la liste des variables de la machine, la clause INVARIANT est l'invariant de la machine (uniquement le type des variables dans notre cas) et la clause INITIALISATION qui spécifie de quelle manière les variables sont initialisées. La clause OPERATIONS est légèrement modifiée puisque qu'elle ne peut définir qu'une seule opération, en fait un événement unique auquel on ne donne pas de nom. Ainsi la clause OPERATIONS ne contient qu'une substitution. On ajoute trois nouvelles clauses : IMPORTS donne la liste des modèles BHDL qui sont importés, et les clauses INPUTS et OUTPUTS donnent respectivement les listes des variables d'entrée et de sortie. Ces variables sont typées dans l'invariant.

FIG. 8.1 – Structure d'un modèle BHDL

MACHINE *D*
IMPORTS *imp*
SEES *param*
INPUTS *i*
OUTPUTS *o*
VARIABLES *v*
INVARIANT *inv*
INITIALISATION *init*
OPERATIONS *C*
END

Les neuf clauses sont expliquées plus en détail ci-dessous, une grammaire complète est donnée dans l'annexe A.

- La première clause (MACHINE) donne le nom du modèle BHDL. Les noms donnés aux modèles sont supposés uniques et permettent d'identifier de manière unique chaque modèle. Le nom est utilisé en particulier lorsque l'on souhaite importer le modèle dans un autre modèle. Si les noms ne sont pas attribués de manière unique il y a un risque de conflit.

- La clause IMPORTS spécifie la liste des modèles qui sont importés. Chaque élément de la liste est composé de deux identificateurs. Le premier est le nom que l'on attribue au modèle importé et le second est le nom du modèle que l'on importe. Le premier nom que l'on donne sert à identifier le modèle de manière unique lorsque l'on importe plusieurs fois le même modèle BHDL. Chaque nom correspond donc à une unique instance du modèle importé. Nous donnons ci-dessous la forme

que prend syntaxiquement la clause IMPORTS, c'est une liste de couples séparés par des virgules, chaque couple étant constitué de deux identificateurs séparés par le caractère "deux points" (rejoint la notation de l'appartenance à un ensemble en B).

$$imp \quad = \quad name_1 : design_1, ..., name_n : design_n$$

- La clause SEES indique la liste des machines déclaratives qui sont vues par le modèle. Le modèle peut utiliser directement (sans renommage) tous les types et toutes les constantes qui sont définis dans ces machines. Pour éviter tout conflit de nom, un modèle ne doit pas voir deux machines qui définissent deux constantes ou deux types ayant le même nom. La liste des machines vues est une liste d'identificateurs séparés par des virgules.

$$param \quad = \quad pack_1, ..., pack_p$$

- La clause INPUTS (resp. OUTPUTS) définit la liste des entrées (resp. des sorties) du modèle BHDL. Les entrées et le sorties sont appelées ports. Les noms des ports sont utilisés directement par le modèle lui-même et par d'autres modèles lorsqu'ils importent ce modèle. A l'intérieur du modèle, les ports sont considérés comme des variables : ils sont lus et écrits de la même manière que toute autre variable, à la différence qu'une entrée ne peut être que lue et une sortie ne peut être qu'écrite.

- La clause VARIABLES déclare la liste des variables qui sont locales au modèle. Ces variables ne sont pas accessibles à un autre modèle (lors d'une importation). Ces variables modélisent aussi bien des fils que des registres, le langage BHDL ne fait pas de distinction explicite entre les deux. C'est la façon dont une variable est utilisée qui détermine si elle modélise un fil ou un registre. La classification en fils et registres peut s'effectuer automatiquement en utilisant les ensembles supports (cf. section 9.2).

- Les types des variables, des entrées et des sorties sont spécifiés dans la clause INVARIANT. Cette clause est limitée par rapport à la clause traditionnelle de B. Elle ne peut contenir que des invariants de typage. Cette clause consiste en une conjonction de tous les typages (*nom_variable* $\in$ *type_variable*) des variables, des entrées et des sorties. Les types sont définis dans les machines qui sont vues via la clause SEES. Cette clause INVARIANT d'un modèle BDHL doit être extraite de la clause INVARIANT du modèle B correspondant. Ces invariants sont donc déjà prouvés dans le modèle B, il n'est alors pas nécessaire de reproduire des obligations de preuve au niveau du modèle BHDL. Chaque variable ne doit être typée qu'une seule fois.

$$inv \quad = \quad i \in I \wedge o \in O \wedge v \in V$$

- La clause INITIALISATION est la même qu'en B. Elle contient une substitution qui s'applique une seule fois lorsque le modèle est initialisé. Les entrées et les sorties doivent toujours être initialisées de façon non déterministe par $i :\in I \| o :\in O$ où I et O sont leurs types respectifs. Les variables peuvent être initialisées de manière déterministe par $v := c$, où c est une constante, ou de façon non déterministe par $v :\in V$. Le seul opérateur de composition autorisé dans la clause initialisation est l'opérateur parallèle ($\|$). Lorsqu'une variable est initialisée de manière déterministe, elle sera automatiquement considérée comme modélisant un registre (initialisé un fil n'aurait pas de sens). Un registre peut éventuellement ne pas être initialisé, cependant ce type de modèle doit être manipulé avec précaution car un registre a physiquement toujours un état initial. On donne ci-dessous la forme générale de la clause initialisation. La variable i (resp. o) représente une entrée et I (resp. O) est son type. La variable v_u est une variable initialisée de façon non déterministe, son type est V_u, et v_d est une variable initialisée, de manière déterministe, à la constante c.

$$init \quad = \quad i :\in I \parallel o :\in O \parallel v_u :\in V_u \parallel v_d := c$$

- La description BHDL du comportement du système se trouve dans la clause OPERATIONS. Cette clause contient une seule substitution. Le modèle étant synchrone (par définition du langage BHDL), cette substitution représente la façon dont l'état du système évolue à chaque cycle d'horloge. Les substitutions de base (:= et :∈) peuvent être composées en parallèle (||), en séquence (;) ou par une composition conditionnelle (IF). La figure 8.2 résume le langage des substitutions qui peut être utilisé.

  Par rapport au langage B, on ajoute la substitution de connexion d'un modèle BHDL importé :

$$(o_1 : ol_1, ..., o_n : ol_m) \leftarrow instname(i_1 : il_1, ..., i_n : il_n)$$

  Le nom de l'instance est donné par l'identificateur *instname* (il doit être tel qu'il a été défini dans la clause IMPORTS). A gauche de la flèche se trouve la liste des connexions de sortie et dans la parenthèse la liste des connexions d'entrée. Une liste de connexions est une liste de couples séparés par des virgules. Chaque couple est de la forme $p : v$ où p est le nom d'un port (entrée ou sortie, suivant le cas) du modèle importé, et v le nom d'une variable locale.

  On ajoute également la possibilité de créer un bloc sous-modèle en utilisant le mot clef BLOCK. Dans un bloc sous-modèle on peut déclarer des variables, qui doivent être initialisées dans le bloc, dans une clause INITIALISATION, et une substitution dans une clause OPERATIONS. L'invariant *binv* donne les types des variables locales au bloc. Le bloc possède donc son propre état et évolue tel que la substitution locale le spécifie. Un bloc peut utiliser les variables du modèle globale mais ce dernier ne peut pas utiliser les variables du bloc. Un bloc sous-modèle modélise un sous-composant du système qui est modélisé 'sur place', c'est-à-dire qu'on donne directement le code du composant sans avoir recours à une importation. Un bloc ne contient pas de liste d'entrées/sorties ni de connexion, il lit et écrit directement les variables du modèle BHDL dans lequel il se trouve. Les variables lues peuvent être considérées comme les entrées et les variables écrites comme des sorties.

FIG. 8.2 – Grammaire des substitutions utilisées par BHDL

$S ::=$
- $S \parallel S$
- $S ; S$
- BEGIN S END
- IF $\mathcal{P}$ THEN S^1 ELSE S^2 END
- IF $\mathcal{P}$ THEN S END
- *var* := *exp*
- *var* :∈ *exp*
- BLOCK *listvar*
  INVARIANT *binv*
  INITIALISATION S^1
  OPERATIONS S^2
  END
- *lport* ← *ident* (*lport*)

Machine déclarative

Une machine déclarative se compose de quatre clauses qui ont la même définition que celles du B traditionnel. La figure 8.3 donne une vue globale d'une machine déclarative.

Nous détaillons les quatre clauses d'une machine déclarative ci-dessous. Une grammaire complète est donnée dans l'annexe A.

- La première clause MACHINE indique le nom de la machine. C'est ce nom qui sera utilisé par les autres machines via la clause SEES pour voir cette machine. Une machine est identifiée de manière unique par son nom, il faut donc éviter de donner le même nom à deux machines.

FIG. 8.3 – Structure d'une machine déclarative

MACHINE P
SEES *param*
CONSTANTS *const*
PROPERTIES *Prop*
END

- Comme pour un modèle BHDL, une machine déclarative peut voir d'autres machines déclaratives. La liste des machines qui sont vues et dont on souhaite utiliser les définitions est donnée par la clause SEES.

- La clause CONSTANTS contient la liste des constantes et des types qui sont définis par la machine. Ce que nous appelons *type* est en fait une simple constante en B, un ensemble défini comme constante. En BHDL, on ne peut pas manipuler les ensembles d'une façon aussi générale qu'il est possible de le faire en B. Seules des constantes peuvent être des ensembles et ils sont utilisés uniquement pour typer des variables (ou des constantes) ou définir d'autres types. C'est pourquoi nous faisons dans notre vocabulaire la distinction entre les *constantes* qui sont des valeurs (non ensemblistes) qui peuvent être assignées à des variables, et les *types* qui servent uniquement pour le typage ; en BHDL un type ne peut pas être affecté à une variable. Par conséquence, en BHDL, un type (c'est-à-dire, en B, une constante ensembliste) ne peut pas être défini comme un ensemble d'ensemble ; sinon une variable de ce type serait un ensemble, ce qui n'a pas de sens en BHDL.
  D'un point de vue formel, il y a cependant une exception à cette règle. En effet, en BHDL les vecteurs sont modélisés comme des fonctions B, et les fonctions B sont en fait définies comme des ensembles. Cela dit, les vecteurs ne sont pas manipulés comme des ensembles en BHDL. Les vecteurs sont créés en utilisant des λ-expressions et sont lus en utilisant la notation $v(i)$, où v est un vecteur et i l'index pour lequel on souhaite connaître la valeur.

- Les définitions elles-mêmes des constantes et des types sont spécifiées dans la clause PROPERTIES. Les définitions sont des égalités, il y en a de quatre sortes qui sont détaillées ci-dessous. La notation c_i (resp. t_i) correspond à la constante (resp. type) qui est définie, pc_i (resp. pt_i) dénote une constante (resp. type) prédéfinie plus haut dans la même machine ou dans une autre machine dont le nom figure dans la clause SEES, *num* correspond à une constante numérique littérale et les éléments e_i sont des identificateurs.

$$Prop \quad = \quad c_1 = num \wedge t_2 = pc_2..pc_3 \wedge t_3 = \{e_1, ..., e_j\} \wedge t_4 = pt_4 \rightarrow pt_5$$

  - Constante numérique :

$$c_1 = num$$

    La constante c_1 est définie avec une valeur égale à la constante numérique littérale *num*.

  - Type défini comme un intervalle :

$$t_2 = pc_2..pc_3$$

    Le type t_2 est le type correspondant à l'intervalle définit par les deux nombres entiers *pc*1 et *pc*2. Les deux constantes pc_1 et pc_2 peuvent être des constantes définies précédemment dans la même machine, ou définies dans une des machines vues listées dans la clause SEES, ou encore des valeurs numériques (entières) littérales.

  - Type défini en extension :

$$t_3 = \{e_1, ..., e_j\}$$

Le type t_3 est défini en extension en donnant la liste des éléments (e_i) contenus dans ce type. Les éléments e_i sont des identificateurs. La valeur d'une variable de ce type sera toujours un des éléments de la liste. Il faut noter que cette manière de définir un type en extension est accepté en BHDL mais n'est pas complètement correct en B. En effet, un outil B signalera que les identificateurs e_i ne sont pas définis. Dans ce cas. Pour assurer la compatibilité avec le langage B, un type défini en extension peut être déclaré dans une clause SETS (en répétant la définition donnée dans la clause PROPERTIES) au lieu de la clause CONSTANTS. Pour être bien formé, l'ensemble des définitions de types en extension doit utiliser des identificateurs différents pour nommer leurs éléments.

- Type vecteur (défini comme une fonction totale) :

$$t_4 = pt_4 \rightarrow pt_5$$

Le type t_4 représente l'ensemble des vecteurs dont les index sont du type pt_4 et les valeurs du type pt_5. Les types pt_4 et pt_5 sont des types prédéfinis dans la même machine où dans une des machines vues. Une variable de ce type est un vecteur.

8.1.2 Syntaxe abstraite

Pour des raisons de commodité d'écriture nous utiliserons parfois une syntaxe plus abstraite pour noter la composition conditionnelle et la structure de bloc. Elle permet de raccourcir l'écriture de ces substitutions, mais aussi de les considérer comme des opérateurs binaires de composition.

- La composition conditionnelle IF P THEN A ELSE B END sera parfois notée $\mathcal{C}_P(A, B)$ ou encore $A\,\mathcal{C}_P\,B$.
- le constructeur de bloc BLOCK v INITIALISATION I OPERATIONS S END sera parfois noté $\mathcal{B}_v(I, S)$ ou encore $I\,\mathcal{B}_v\,S$.

8.1.3 Exemples

Nous présentons deux exemples de modèle BHDL. Le premier est un compteur qui compte à rebours de 7 à 0. Lorsqu'il atteint 0, l'état du compteur reste inchangé jusqu'à ce qu'un signal de réinitialisation soit détecté. Lorsque c'est le cas il recommence à compter à rebours. Cet exemple illustre la modélisation d'un circuit simple en BHDL et comment utiliser les machines déclaratives. Le second exemple est un circuit qui a un comportement proche du premier mais qui se réinitialise automatiquement à 7 lorsque le compteur atteint 0. Ce second exemple illustre l'importation du circuit en réutilisant le premier exemple.

Compteur 3-bits

Nous prenons pour premier exemple de modèle de circuit synchrone en BHDL un compteur 3-bits. Ce compteur compte à rebours de 7 jusqu'à 0. Il a une seule entrée, deux sorties et une variable interne :

- L'entrée *rst* peut être utilisée pour réinitialiser le compteur à 7. Si l'entrée *rst* est positionnée à *TRUE* le compteur est réinitialisé, si elle positionnée à *FALSE* le compteur continue à compter à rebours (à moins qu'il ait déjà atteint 0). Il s'agit d'un signal de réinitialisation *synchrone* , c'est-à-dire que, comme pour toute entrée, sa valeur ne doit changée qu'une fois en début de cycle. Il ne s'agit pas du signal de réinitialisation *asynchrone* qui initialise l'état du circuit lors du démarrage du système : la réinitialisation asynchrone (le signal peut changer indépendamment de l'horloge, pour redémarrer le système) est spécifiée dans la clause INITIALISATION du modèle BHDL ; le signal de réinitialisation asynchrone n'est jamais modélisé explicitement dans un modèle BHDL.
- Le compteur positionne la sortie *alm* à *TRUE* lorsqu'il atteint 0.
- La sortie *val* montre à l'extérieur du circuit la valeur du compteur.

- La variable interne n contient la valeur du compteur, elle est décrémentée à chaque cycle jusqu'à atteindre 0.

La figure 8.4 page 166 présente le modèle BHDL du compteur à rebours 3-bits. La machine déclarative *PARAM* définit les constantes *seven3*, *one3* et *zero3*, et la machine déclarative *BHDL* définit quelques types implantables utiles. Dans l'exemple du compteur 3-bits nous utilisons $UINT3$ qui est le type des nombres entiers non signés représentables sur trois bits. Ce type correspond à l'intervalle des entiers compris entre 0 et 7. Nous utilisons également le type *BOOL*, représentant les valeurs booléennes, définit en extension, il contient les deux éléments *TRUE* et *FALSE*.

On remarquera l'utilisation de l'opérateur de composition parallèle ||. Sa sémantique est la même qu'en B : la valeur de la sortie *val* est la valeur *avant* de la variable n (la valeur de n avant que la substitution soit appliquée), et pas la valeur de n résultant de l'application de la substitution appliquée en parallèle par la substitution conditionnelle.

Interprétation en termes de circuit (cf. figure 8.4 page 166) Interprétée en termes de séquence de cycles, la valeur de la sortie *val* pendant un cycle donné est la valeur de la variable n calculée au cycle précédent. D'un point de vue architectural, cela signifie que n est un registre et que la sortie *val* est connectée directement à la sortie de ce registre. Ceci a pour effet que la sortie *val* est stable pendant toute la durée du cycle. On obtient le même résultat pour la sortie *alm*. La figure 8.4 page 166 montre l'évolution des signaux obtenue lors d'une simulation de ce circuit.

Compteur 3-bits modulo

L'exemple précédent est un compteur qui compte à rebours et qui reste bloqué à 0 tant que le signal de réinitialisation synchrone *rst* n'est pas positionné à *TRUE*). Nous illustrons maintenant comment importer un modèle BHDL existant en modélisant un compteur à rebours modulo 8, c'est-à-dire que lorsque le compteur atteint 0, il redémarre à 7. Pour cela nous réutilisons le compteur *counter3bits* précédent. Notre nouvelle machine BHDL positionne à *TRUE* le signal d'entrée *rst* du compteur importé lorsque le compteur à atteint 0. Bien que le nouveau compteur *counter3bitsmodulo* ait un comportement différent, il a la même interface (mêmes entrées et mêmes sorties) que le compteur *counter3bits* précédent. La machine BHDL de ce nouveau compteur est donnée sur la figure 8.5).

L'interface du compteur 3-bits modulo 8 est la suivante :

- une entrée de réinitialisation synchrone *rst* pour remettre le compteur à 7,
- une sortie *alm* qui est positionnée à *TRUE* lorsque le compteur atteint 0, et,
- une sortie *val* qui fournit la valeur du compteur.

Deux variables internes sont utilisées :

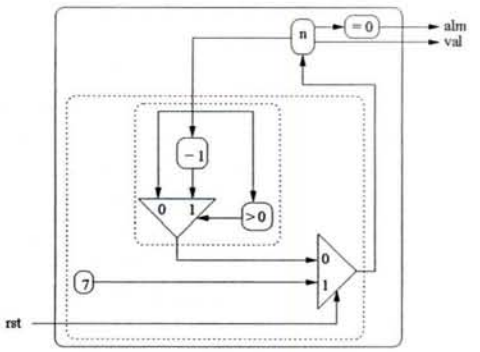
- La variable *rval* qui est connectée à la sortie du compteur importé sert à récupérer la valeur de ce compteur. Notez que lorsque la variable *rval* est lue pour la première fois, par la substitution $rstbis := \text{bool}(rval = one3 \vee rst = TRUE)$, aucune valeur ne lui a encore été assignée (dans le cycle courant). Du point de vue de la séquence des cycles, ceci signifie que la valeur de la variable *rval* dans cette première substitution est celle qui lui a été assignée au cycle précédent par la substitution $(alm := alm,\ val := rval) \leftarrow c3(rst : rstbis)$. Au premier cycle, la valeur de *rval* est celle qui lui est assignée par la clause INITIALISATION. D'un point de vue architectural cela signifie que *rval* est un registre.
- La variable *rstbis* porte la valeur qui est donnée en entrée au compteur importé. Cette variable est positionnée à *TRUE* quand l'entrée *rst* est elle-même positionnée à *TRUE* ou lorsque le compteur atteint la valeur 0. Dans la substitution $rstbis := \text{bool}(rval = one3 \vee rst = TRUE)$, la variable *rstbis* est positionnée à *TRUE* lorsque *rval* vaut 1 et non pas 0. Ceci est nécessaire car, lorsque cette substitution est appliquée, *rval* porte la valeur du compteur du cycle précédent (c'est un registre, comme expliqué au point précédent) et elle vaut 0 lorsque le compteur est à 1 : la sortie *val* du compteur importé est reliée à un registre interne, sa valeur durant un cycle est la valeur calculée au cycle précédent (cf. explications dans l'exemple précédent). Pour résumé, lorsque la

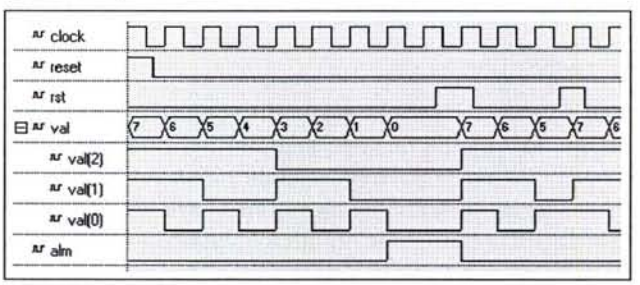
FIG. 8.4 – Modèle BHDL, représentation graphique et chronogramme d'une simulation d'un compteur à rebours 3-bits

```
MACHINE
    counter3bits
SEES
    BHDL,PARAM
INPUT
    rst
OUTPUT
    val,alm
VARIABLES
    n
INVARIANT
    rst ∈ BOOL ∧
    val ∈ UINT3 ∧
    alm ∈ BOOL ∧
    n ∈ UINT3
INITIALISATION
    rst :∈ BOOL ||
    val :∈ UINT3 ||
    alm :∈ BOOL ||
    n := seven3
OPERATIONS
    IF rst = TRUE THEN
        n := seven3
    ELSE
        IF n > zero3 THEN
            n := n − one3
        END
    END ||
    val := n ||
    alm := bool(n = zero3)
END
```

```
MACHINE
    PARAM
CONSTANTS
    seven3,
    one3,
    zero3
PROPERTIES
    seven3 = 7 ∧
    one3 = 1 ∧
    zero3 = 0
END
```

```
MACHINE
    BHDL
CONSTANTS
    ..., UINT3, BOOL, ...
PROPERTIES
    ...
    UINT3 = 0..7 ∧
    BOOL = {TRUE, FALSE}
    ...
END
```





première substitution est appliquée, *rval* porte la valeur obtenue au cycle précédent de la sortie *val* du compteur importé, sortie qui elle-même portait la valeur du compteur obtenue encore un cycle plus tôt. Nous faisons donc le test $rval = 1$ pour assurer une cohérence, à la fin du cycle, entre la sortie *val* et la sortie *alm* : la sortie *alm* fournit la valeur *TRUE* pendant le même cycle où la sortie *val* fournit la valeur 0. Il s'agit d'un choix de modélisation afin d'obtenir un comportement similaire à celui du premier exemple.

On remarquera l'utilisation de l'opérateur de composition séquentielle ';', il a la même sémantique qu'en B : la valeur de *rstbis* qui est envoyée au composant *c*3 est celle qui résulte de l'application de la première substitution $rstbis := \text{bool}(rval = one3 \vee rst = TRUE)$. De la même façon, la valeur de la

FIG. 8.5 – Modèle BHDL du compteur à rebours modulo 8

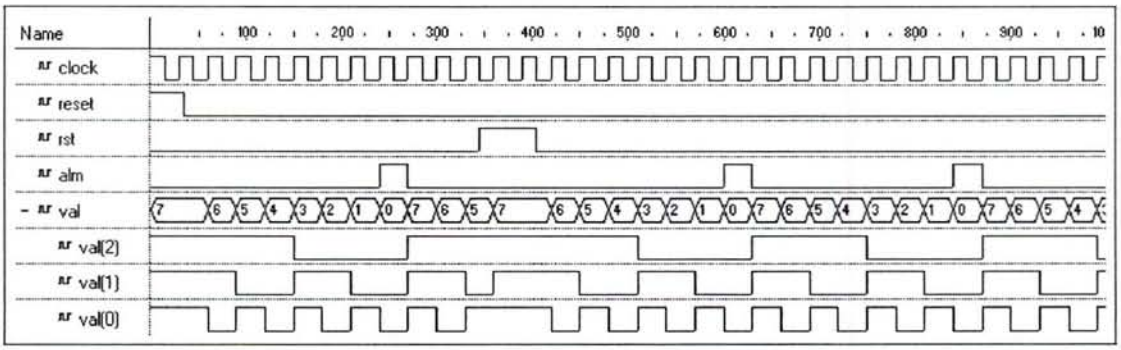
```
MACHINE
    counter3bitsmodulo
IMPORTS
    c3 : counter3bits
SEES
    BHDL,PARAM
INPUT
    rst
OUTPUT
    val,alm
VARIABLES
    rstbis, rval
INVARIANT
    rst ∈ BOOL ∧
    val ∈ UINT3 ∧
    alm ∈ BOOL ∧
    rval ∈ UINT3 ∧
    rstbis ∈ BOOL
INITIALISATION
    rst :∈ BOOL ||
    val :∈ UINT3 ||
    alm :∈ BOOL ||
    rval := 0 ||
    rstbis :∈ BOOL
OPERATIONS
    rstbis := bool(rval = one3 ∨ rst = TRUE)
    ;
    (alm :alm, val :rval ) ← c3(rst : rstbis)
    ;
    val := rval
END
```

variable *rval* est la valeur donnée en sortie par *c*3 et la sortie *val* du compteur modulo 8 est égale à la sortie du compteur *c*3. La figure 8.6 montre les signaux obtenus lors d'une simulation de ce circuit.

Remarque Un utilisateur VHDL pourrait être tenté d'écrire quelque chose dans le style $rstbis \Leftarrow alm$. C'est-à-dire de connecter directement l'entrée du composant *counter3bits* à sa propre sortie dans le but d'éviter d'introduire le registre supplémentaire *rval*. Il se trouve que cela fonctionne en VHDL car dans le modèle de *counter3bits* il y a un registre dans le chemin entre l'entrée *rst* et la sortie *val*, cela n'introduit donc pas de cycle dans la partie combinatoire. Ceci est tout de même dangereux car on ne peut pas nécessairement toujours savoir s'il y a un registre dans le composant que l'on réutilise et on risque d'introduire des cycles dans la partie combinatoire, ce qui est inacceptable (cf. Circuits combinatoires, section 1.1).

En BHDL, ceci est syntaxiquement impossible. Une fois qu'un composant a été utilisé, il est impossible de réutiliser la même instance, il n'y a donc aucune possibilité de connecter les sorties sur les entrées de la même instance d'un composant. Si on écrivait quelque chose du style $(alm : alm,\ val : rval) \leftarrow c3(rst : alm)$, ou encore $(alm : rstbis,\ val : rval) \leftarrow c3(rst : rstbis)$, on n'obtiendrait pas l'effet désiré. Dans le premier cas, la valeur de *alm* qui est envoyée en entrée de *c*3 est la valeur *avant* de *alm*, qui serait en l'occurrence la valeur renvoyée par *c*3 au cycle précédent (cela introduirait un registre), ce qui n'est pas ce que nous souhaitons faire. Dans le deuxième cas, la valeur de la variable *rstbis* serait modifié par cette substitution (elle porterait la même valeur que la sortie *alm*

FIG. 8.6 – Chronogramme d'une simulation utilisant le compteur à rebours modulo 8



de $c3$) mais la valeur qui est envoyée en entrée à $c3$ est toujours celle qui résulte de l'application de la substitution précédente.

8.2 Résumé

Dans ce chapitre nous avons d'abord introduit le langage BHDL. C'est un langage basé sur le langage B. La différence principale réside dans le fait qu'un modèle BHDL ne contient qu'un seul événement qui modélise la façon dont le circuit électronique évolue pendant un cycle d'horloge. La clause INITIALISATION contient les initialisations des registres. Les deux clauses INPUTS et OUTPUTS sont ajoutées afin de spécifier quelles sont les variables qui modélisent les entrées et les sorties du circuit. Nous avons également ajouté deux substitutions : l'une permettant la réutilisation de composants existants, l'autre, sous forme d'un bloc, permettant la modélisation 'sur place' (sans avoir à recourir à une importation) d'un sous-composant du circuit.

Chapitre 9

Éléments de sémantique

L'objectif de ce chapitre est de présenter comment la sémantique du langage B est définie en terme de prédicats et en terme de relations. Ces définitions servent de base à nos sémantiques pour le langage BHDL définies dans les chapitres suivants. Nous proposons également d'ajouter deux substitutions au langage permettant de modéliser l'importation de composants extérieurs. Nous définissons la notion d'ensembles supports d'une substitution qui nous sera très utile pour définir des conditions de bonne formation ainsi que la sémantique relationnelle.

La méthode B a été initialement définie par J.-R. Abrial dans le but de modéliser des programmes informatiques et de les développer de façon formelle. Plus récemment la méthode B a évolué (en donnant naissance au B événementiel) pour modéliser des systèmes de toute nature. La méthode ainsi que ses fondements formels sont décrits en détail dans le BBook [1]. Le principe fondamental de la méthode est le raffinement : plusieurs descriptions du système sont données à des niveaux d'abstraction différents et la méthode génère automatiquement les obligations de preuve nécessaires pour assurer que l'un est le raffinement de l'autre. La description d'un système se fait de manière incrémentale, en partant de sa spécification, puis, raffinement après raffinement on finit par obtenir une implantation du système. Le langage utilisé est basé sur les substitutions généralisées que nous présentons dans ce chapitre de manière syntaxique et en donnant leur sémantique. La sémantique peut s'exprimer de deux manières : soit de manière prédicative, soit de manière ensembliste. Nous présentons formellement la notion de raffinement d'une substitution généralisée. Le lecteur trouvera toutes les démonstrations dans le BBook [1]. Nous définissons ensuite la notion d'*ensemble support* d'une machine B puis nous donnons les conditions de bonne formation d'une machine BHDL, qui s'ajoutent aux conditions de la méthode B.

9.1 Substitutions généralisées

Les substitutions généralisées [1] sont une généralisation de la notion mathématique de substitution. L'application de la substitution mathématique $x := E$ sur une formule F, notée $[x := E]F$ ou $F(E/x)$, consiste à remplacer toutes les occurrences libres de la variable x par l'expression E dans F. La nature de F peut dépendre du contexte :

- F peut être simplement une expression, dans ce cas le résultat de la substitution est une expression
- F peut être un prédicat et le résultat est un prédicat,
- F peut être l'état d'une machine d'états, dans ce cas la substitution fait évoluer la machine vers un nouvel état dans lequel la valeur de x est E et les valeurs des autres variables sont inchangées.

Le langage des substitutions généralisées généralise la notion de substitution en ajoutant des substitutions et en proposant des opérateurs permettant de composer les substitutions entre elles. La sémantique du langage des substitutions généralisées est définie dans le calcul des plus faibles préconditions [37, 38] par un transformateur de prédicat $\lambda(P \in \mathfrak{P}) \cdot ([S]P)$ (fonction qui associe $[S]P$ à un prédicat P, $\mathfrak{P}$ est l'ensemble de tous les prédicats) où $[S]P$ est la plus faible précondition telle que P soit établi par

l'application de la substitution S [25]. C'est-à-dire que l'application de S, à partir de n'importe quel état v qui satisfait $[S]P$, amène le système dans un état $[S]v$ qui satisfait P.

$$\forall v \cdot (\ ([S]P)(v) \Rightarrow P([S]v)\ )$$

Par exemple, si on prend la substitution $S \equiv x := x + 1$ et le prédicat $P \equiv x \leq 10$. La plus faible précondition pour que S établisse P est que $x \leq 9$ $(x + 1 \leq 10)$.

Les définitions inductives de quelques substitutions généralisées sont données sur la figure 9.1. La substitution $x :\in T$, où T est un ensemble, positionne la valeur de la variable x à un des éléments de T. Les substitutions peuvent être composées en parallèle, ce qui est noté $S_1 \| S_2$ (les substitutions S_1 et S_2 sont appliquées simultanément), ou séquentiellement en écrivant $S_1; S_2$ (la substitution S_1 est d'abord appliquée, et seulement ensuite S_2 est appliquée). Des substitutions plus complexes sont obtenues en utilisant la composition conditionnelle, notée IF P THEN S_1 ELSE S_2 END, où P est un prédicat.

FIG. 9.1 – Plus faibles préconditions des substitutions généralisées

Nom	Substitution S	Plus faible précondition $[S]R$
Substitution simple	$x := E$	$R(E/x)$
Substitution multiple	$x, y := E, F$	$[z := F][x := E][y := z]R$
Choix non déterministe dans un ensemble	$x :\in C$	$\forall v.(v \in C \Rightarrow R(v/x))$
Substitution sans effet	*skip*	R
Composition parallèle[4]	$S \| T$	$trm(S \| T) \quad \wedge$ $\forall x', y'(prd_{x,y}(S \| T) \Rightarrow [x, y := x', y']R)$
Composition séquentielle	$S; T$	$[S]([T]R)$
Composition conditionnelle	IF $\mathcal{P}$ THEN S ELSE T END	$(\mathcal{P} \Rightarrow [S]R) \wedge (\neg\mathcal{P} \Rightarrow [T]R)$
Substitution gardée	WHEN $\mathcal{P}$ THEN S END	$\mathcal{P} \Rightarrow [S]R$

Le langage B propose une composition parallèle, notée $\|$ (similaire au mot-clef 'par' de HandleC, mais sans canal de communication). Ceci permet à deux blocs indépendants d'être appliqués en même temps. Une variable peut être lue dans les deux blocs mais ne peut être écrite que par au plus l'un des deux. Sur la figure 9.1, la composition parallèle n'est pas définie ici en donnant la plus faible précondition mais en donnant deux prédicats qui sont suffisant pour caractériser une substitution. Le premier prédicat, *trm*, est un prédicat qui est évalué positivement si la substitution termine. Le deuxième prédicat *prd* est un prédicat, dit prédicat *avant-après*, qui lie les valeurs des variables *avant* (notées x) l'application de la substitution aux valeurs des variables *après* (notées x') l'application de la substitution. Ces deux prédicats sont définis ci-dessous. Dans les définitions données, x est l'ensemble des variables qui sont considérées par S (cf. les ensembles supports dans la section 9.2) et y est l'ensemble des variables considérées par T ; les ensembles x et y sont supposés disjoints. Notons que par définition $(x' \neq x) \equiv \neg(x' = x)$. C'est-à-dire que $x', y' \neq x, y \equiv \neg(x', y' = x, y) \equiv x' \neq x \vee y' \neq y$.

[4] Le transformateur de prédicat correspondant à la composition parallèle n'est pas défini directement mais en définissant les deux prédicats *trm* (terminaison) et *prd* (prédicat *avant-après*, c.f. section 9.1.3). Il est prouvé que ces deux prédicats caractérisent complètement la substitution. On trouvera dans le BBook [1] que la forme normale d'une substitution S est $S = trm(S) | @x' \cdot (pred_x(S) \Rightarrow x := x')$

$$trm(S) \equiv [S](x = x)$$
$$trm(S\|T) \equiv trm(S) \wedge trm(T)$$
$$prd_x(S) \equiv \neg[S](x' \neq x)$$
$$prd_{x,y}(S\|T) \equiv prd_x(S) \wedge prd_y(T)$$

En remarquant que pour toute valeur x de l'espace des états s nous avons de façon évidente $x = x \Leftrightarrow x \in s$ ($x \in s$ est un invariant) et $x' \neq x \Leftrightarrow x \in \overline{\{x'\}}$ (où, pour tout ensemble a, $\overline{a} = s - a$), les définitions de trm et prd peuvent s'écrire également de la façon suivante :

$$trm(S) \equiv [S](x \in s)$$
$$prd_x(S) \equiv \neg[S](x \in \overline{\{x'\}})$$

Nous ne définissons pas ici la substitution WHILE (contenue dans le sous ensemble du langage B utilisé pour la modélisation de logiciels) puisque nous ne l'utilisons pas dans la modélisation de circuits, cette substitution n'est pas implantable "telle qu'elle" dans un circuit électronique. La seule façon d'obtenir des boucles de calcul dans un circuit synchrone est d'utiliser le comportement cyclique du circuit.

9.1.1 Modèle ensembliste

Les définitions précédentes s'appuient sur les transformateurs de prédicat, ce qui peut conduire à définir des notions (comme celle de raffinement, cf. section 9.1.2) en utilisant des quantificateurs sur des prédicats. La méthode B s'appuie sur la logique du premier ordre, dans laquelle on ne peut pas écrire de quantification sur les prédicats. Pour garder une certaine cohérence, on peut vouloir éviter d'avoir à écrire des quantifications d'ordre supérieur. Pour éviter d'avoir à faire des quantifications sur les prédicats, un modèle ensembliste des substitutions généralisées a été créé (cf. [1]). De la même façon qu'une substitution peut être caractérisée en donnant les deux prédicats trm et prd, on définit pour chaque substitution un ensemble pre qui est le sous-ensemble de l'espace des états pour lequel la substitution termine, et la relation rel qui lie l'état dans lequel se trouve le système avant d'appliquer la substitution et l'état après. Dans la suite, l'ensemble s correspond à l'espace des états.

$$pre(S) = \{x | x \in s \wedge trm(S)\} \quad \text{c'est-à-dire} \quad \{x | x \in s \wedge [S](x \in s)\}$$
$$rel(S) = \{x, x' | x, x' \in s \times s \wedge prd_x(S)\} \quad \text{c'est-à-dire} \quad \{x, x' | x, x' \in s \times s \wedge \neg[S](x \in \overline{\{x'\}})\}$$

On peut également définir une substitution S par un *transformateur d'ensemble* $str(S)$ qui est l'interprétation du transformateur de prédicat dans le modèle ensembliste. Un prédicat est interprété par le plus grand sous-ensemble de l'espace des états dans lequel le prédicat est vérifié ($\mathcal{P} = \{x | x \in s \wedge P(x)\}$). Dans la suite, $\mathbb{P}(s)$ est l'ensemble de tous les sous-ensembles de s.

Définition 1 (Transformateur d'ensemble d'une substitution) *Le transformateur d'ensemble associé à une substitution S, noté $str(S)$, est une fonction qui associe à un ensemble d'états p l'ensemble (le plus grand) des états à partir desquels l'application de S emmène dans p.*

$$str(S) \equiv \lambda p \cdot (p \in \mathbb{P}(s) \mid \{x | x \in s \wedge [S](x \in p)\})$$

Propriété 1 (Décomposition du transformateur d'ensemble) *Le transformateur d'ensemble possède la propriété suivante qui divise le transformateur en deux parties : la première correspond à la terminaison de la substitution et la deuxième partie est la relation* avant-après *de la substitution.*

$$x \in str(S)(p) \quad \Leftrightarrow \quad x \in str(S)(s) \wedge \forall x' \cdot (x' \in \overline{p} \Rightarrow x \in str(S)(\overline{\{x'\}}))$$

Le transformateur de prédicat $str(S)$ définit la substitution S de la même façon que le transformateur de prédicat définit dans la section 9.1. Les deux ensembles pre et rel peuvent se définir à partir de $str(S)$.

$$pre(S) = str(S)(s), \text{ s'obtient directement en appliquant } str(S) \text{ sur } s,$$
$$rel(S) = \{x, x' | (x, x') \in s \times s \wedge x \in \overline{str(S)(\overline{\{x'\}})}\}$$

La preuve pour la relation rel se fait de la façon suivante :

Pour tout x et x' dans s, nous avons $x \in \overline{str(S)(\overline{\{x'\}})} \Leftrightarrow \neg(x \in str(S)(\overline{\{x'\}}))$,
en appliquant $str(S)$ à $\overline{\{x'\}}$ on obtient $\{x | x \in s \wedge [S](x \in \overline{\{x'\}})\}$,
et on déduit $x \in \overline{str(S)(\overline{\{x'\}})} \Leftrightarrow \neg[S](x \in \overline{\{x'\}})$

Théorème 1 (La donnée de $pre(S)$ et $rel(S)$ caractérise complètement la substitution S) *Le transformateur d'ensemble $str(S)$ peut être défini en utilisant les ensembles $pre(S)$ et $rel(S)$:*

$$\forall p \cdot (p \subseteq s \Rightarrow str(S)(p) = pre(S) \cap \overline{rel(S)^{-1}[\overline{p}]})$$

La preuve est faite en développant les définitions.

On calcule d'abord $\overline{rel(S)^{-1}[\overline{p}]}$,

$\overline{rel(S)^{-1}[\overline{p}]} = \{x | x \in s \wedge \exists x' \cdot (x' \in \overline{p} \wedge x', x \in rel(S)^{-1})\}$
$\overline{rel(S)^{-1}[\overline{p}]} = \{x | x \in s \wedge \neg(\exists x' \cdot (x' \in \overline{p} \wedge x', x \in rel(S)^{-1}))\}$
$\overline{rel(S)^{-1}[\overline{p}]} = \{x | x \in s \wedge \forall x' \cdot (x' \in \overline{p} \Rightarrow \neg(x, x' \in rel(S)))\}$
$\overline{rel(S)^{-1}[\overline{p}]} = \{x | x \in s \wedge \forall x' \cdot (x' \in \overline{p} \Rightarrow \neg(x \in \overline{str(S)(\overline{\{x'\}})}))\}$
$\overline{rel(S)^{-1}[\overline{p}]} = \{x | x \in s \wedge \forall x' \cdot (x' \in \overline{p} \Rightarrow x \in str(S)(\overline{\{x'\}}))\}$

On en déduit l'égalité suivante :
$pre(S) \cap \overline{rel(S)^{-1}[\overline{p}]} = str(S)(s) \cap \{x | x \in s \wedge \forall x' \cdot (x' \in \overline{p} \Rightarrow x \in str(S)(\overline{\{x'\}})\}$
La conclusion vient en utilisant la propriété 1.

Remarque Toutes les substitutions que nous utiliserons dans le langage BHDL que nous avons défini terminent toujours (pas de substitution gardée). Ainsi, pour toute substitution S que nous utiliserons en BHDL, l'ensemble $pre(S)$ sera toujours égal à l'espace entier des états. Ainsi, les substitutions de BHDL seront entièrement caractérisées par la relation rel.

D'une manière plus générale, en B événementiel on a l'habitude de séparer l'événement en deux : d'une part la garde et de l'autre une substitution non gardée. La substitution non gardée est donc pleinement définie par le prédicat *avant-après*.

9.1.2 Raffinement d'une substitution généralisée

Le raffinement d'une substitution est un processus permettant de transformer une substitution dite *abstraite* en une substitution plus *concrète*. La substitution concrète est un raffinement de la substitution abstraite si elle est conforme à la spécification que constitue la substitution abstraite.

Le principe intuitif du raffinement est le suivant. On se donne deux substitutions S et T. On dit que T est un raffinement de S si pour toute transformation du système obtenue par l'application de T, il est *possible* d'obtenir la même transformation en appliquant S. Lorsque la substitution T est appliquée, un observateur extérieur attentif à l'évolution du système ne peut pas dire si c'est la substitution S ou la substitution T qui a été appliquée.

On différencie les applications *observables* de substitutions des applications non observables. Lorsqu'une substitution est gardée et que, étant donné l'état du système, sa garde est fausse, la substitution est applicable mais nous dirons que son application n'est pas observable car l'état du système n'est pas touché.

Définition 2 (Substitution observable) *Une substitution est dite observable, à un instant donné, si sa garde est telle que son application est susceptible de modifier l'état du système. Un substitution est observable si et seulement si sa garde peut être vraie.*

Définition 3 (Définition intuitive du raffinement) *Une substitution T est un raffinement d'une substitution S si à chaque fois que l'application de la substitution T est observable, la substitution S est applicable et* peut *produire la même observation.*

D'une manière générale, un raffinement réduit le non déterminisme par rapport à l'abstraction : il peut supprimer des observations possibles, mais il n'en ajoute pas.

On peut illustrer ce principe sur un exemple simple. Prenons la substitution $x :\in \{1, 2\}$ qui, lorsqu'elle est appliquée, donne à la variable x *une* des deux valeurs 1 ou 2. Cette substitution peut être raffinée par la substitution $x := 1$ qui, lorsqu'elle est appliquée, donne à la variable x la valeur 1. Un autre raffinement possible est la substitution $x := 2$.

Outre la réduction du non déterminisme, le raffinement autorise également le renforcement des gardes (pour les substitutions gardées) : c'est-à-dire qu'on peut limiter le nombre de cas dans lesquels la substitution est observable. Ce renforcement des gardes suit bien le principe énoncé ci-dessus : dans tous les cas où la substitution raffinée est observable (c'est-à-dire qu'elle modifie l'état du système), la substitution abstraite l'est aussi (puisqu'elle a une garde moins restrictive).

Sur ce principe, la substitution WHEN $0 \neq 0$ THEN *skip* END, qui n'est jamais observable est un raffinement correct de toute substitution. Un autre exemple serait de raffiner la substitution WHEN $x \in \{1, 2\}$ THEN $y := 0$ END par WHEN $x = 1$ THEN $y := 0$ END.

On donne une définition plus formelle du raffinement en utilisant les transformateurs de prédicats. La plus faible précondition $[S]R$ est un prédicat qui représente l'ensemble des états qui conduisent à un état satisfaisant R par la substitution S. On peut interpréter la définition suivante de cette manière : lorsque la substitution T mène à un état donné (un état satisfaisant un prédicat R), S doit également pouvoir mener à cet état.

Définition 4 (Raffinement - transformateurs de prédicat) *Soient S et T deux substitutions, T est un raffinement de S, et on note $S \sqsubseteq T$ si :*

$$\forall R \cdot ([S]R \Rightarrow [T]R)$$

Pour éviter d'utiliser une quantification sur des prédicats, on peut reformuler la définition précédente dans la théorie ensembliste. L'ensemble des états prédécesseurs d'un ensemble d'état a par la substitution S est un sous-ensemble de l'ensemble des états prédécesseurs de a par la substitution T.

Définition 5 (Raffinement - transformateurs d'ensemble) *Soient S et T deux substitutions, T est un raffinement de S, et on note $S \sqsubseteq T$ si :*

$$\forall a \cdot (a \subseteq s \Rightarrow str(S)(a) \subseteq str(T)(a))$$

9.1.3 Prédicat *avant-après*

La sémantique du langage des substitutions est formellement définie en utilisant les transformateurs de prédicat qui donnent la plus faible précondition pour que l'application d'une substitution conduise le système dans un état qui satisfait un prédicat donné. Cette sémantique *en arrière* ("dis moi ce que tu veux et je te dis ce que tu dois avoir") est bien adaptée pour définir la correction d'un modèle et générer les obligations de preuve mais pas pour modéliser l'évolution d'un système.

Pour définir la façon dont une substitution fait évoluer un système, nous utilisons un prédicat *avant-après*. Un tel prédicat lie l'état du système *avant* l'application de la substitution à l'état du système *après*. Pour toute variable d'état x, la notation x à l'intérieur du prédicat désigne la valeur de la variable avant l'application de la substitution, et la notation x' désigne la valeur de la variable après l'application de la substitution.

Par exemple, le prédicat *avant-après* de la substitution $x := x + 1$ spécifie que la valeur *après* de la variable x est égale à la valeur *avant* de x à laquelle on ajoute 1 ($x' = x + 1$), et que les valeurs des autres variables sont inchangées (les valeurs *après* sont égales aux valeurs *avant*, $\bigwedge_{y \in F - \{x\}} (y' = y)$). Dans la formule ci-dessous, l'ensemble F est l'ensemble de toutes les variables.

$$prd_F(x := x + 1) \equiv x' = x + 1 \wedge \bigwedge_{y \in F - \{x\}} (y' = y)$$

Pour une ensemble de variables donné v, le prédicat prd_v est défini en utilisant les transformateur de prédicat de la façon suivante :

$$prd_v \equiv \neg[S](v' \neq v)$$

Cette définition s'interprète de cette façon : la valeur *après* de v est v' lorsque la substitution S n'établit pas que v et v' sont différents. L'intérêt de la double négation se voit principalement dans les cas où on utilise le non déterminisme, ou quand on utilise des substitutions gardées.

Nous expliquons cette définition sur deux exemples. Dans un premier exemple sur le cas d'une substitution simple $x := E$ pour montrer comment on obtient le prédicat *avant-après* en utilisant le transformateur de prédicat. Ensuite nous expliquerons la nécessité de la double négation ($\neg$ et $\neq$) sur l'exemple d'une substitution non déterministe.

On se place dans un contexte où il n'y a que deux variables d'état, x et y. Dans le cas d'une substitution simple $x := E$, l'effet du transformateur de prédicat $[x := E]$ est de remplacer toute occurrence libre de la variable x par l'expression E (qui peut éventuellement utiliser la variable x). L'expression $[x := E](x', y' \neq x, y)$ peut donc se récrire simplement $x', y' \neq E, y$, et ainsi $\neg[x := E](x', y' \neq x, y) \equiv x', y' = E, y$. Nous obtenons donc un prédicat qui spécifie que la valeur *après* de x doit être la valeur de E et que la valeur de y n'est pas affectée. Remarquons que si l'expression E utilise la variables x (par exemple $E \equiv x + 1$ dans l'exemple précédent), celle-ci reste utilisée et désigne donc la valeur *avant* de x. Le résultat obtenu correspond donc bien au résultat auquel on s'attendait. Dans ce cas, la double négation n'a eu aucun effet. Voyons son intérêt dans le cas d'une substitution non déterministe.

Prenons le cas simple d'une substitution $x :\in T$ où T est un ensemble (disons l'ensemble correspondant au type de x — ce qui implique que T n'est pas vide — pour être certain que la substitution soit bien formée). L'effet du transformateur de prédicat $[x :\in T]$ sur un prédicat donné R est de dire que R doit être vrai quelle que soit la valeur choisie pour x dans T : $[x :\in T](x', y' \neq x, y) \equiv \forall z \cdot (z \in T \Rightarrow x', y' \neq z, y)$. En ajoutant la deuxième négation devant ce prédicat, la quantification universelle se transforme en quantification existentielle : $\neg[x :\in T](x', y' \neq x, y) \equiv \exists z \cdot (z \in T \wedge x', y' = z, y)$. Le prédicat *avant-après* de la substitution non déterministe spécifie donc qu'il existe une valeur qui peut être choisie pour valeur *après* de x. Si on n'avait pas de double négation dans la définition du prédicat *avant-après*, on obtiendrait un prédicat spécifiant que la valeur *après* de x soit égale à toute valeur de T, ce qui n'est en général pas possible (à moins que T soit un singleton).

9.1.4 Exemple

A titre d'exemple, nous calculons le prédicat *avant-après* de l'exemple d'un compteur donné sur la figure 9.2. Les calculs pour la clause INITIALISATION et l'unique événement de la clause OPERATIONS sont faits séparément.

Le prédicat *avant-après* de la clause INITIALISATION spécifie uniquement que la variable n modélisant la valeur interne du compteur est initialisée à 0 et que l'initialisation des autres variables est non déterministe. Cela signifie que ces variables peuvent prendre à l'initialisation n'importe quelle valeur en correspondance avec leur type. En effet, si on considère que les variables $o1$ et $o2$ modélisent des sorties d'un composant, et que rst modélise une entrée, cela n'aurait pas de sens de leur donner une initialisation. Le fait d'utiliser le non déterministe permet de prouver que le système est correct quelles que soient les valeurs de ces variables à l'initialisation. Le prédicat *avant-après* correspondant à cette initialisation est donné ci-dessous.

$$prd_{\{n,rst,o1,o2\}}(\text{INITIALISATION}) = (n' = 0 \wedge rst' \in \{0,1\} \wedge o1' \in \mathbb{N} \wedge o2' \in \mathbb{N})$$

La définition du prédicat *avant-après* pour la composition parallèle est donnée page 171 ($prd_{x,y}(S \| T) \equiv prd_x(S) \wedge prd_y(T)$). Pour obtenir le résultat ci-dessus, le prédicat à été décomposé de cette manière :

$$prd_{\{n,rst,o1,o2\}}(\text{INITIALISATION}) = \begin{cases} prd_n(n := 0) \wedge \\ prd_{rst}(rst :\in \{0,1\}) \wedge \\ prd_{o1}(o1 :\in \mathbb{N}) \wedge \\ prd_{o2}(o2 :\in \mathbb{N}) \end{cases}$$

FIG. 9.2 – Modélisation du compteur par des substitutions

```
INITIALISATION
    n := 0 || rst :∈ {0,1} || o1 :∈ ℕ || o2 :∈ ℕ
OPERATIONS
circuit =
    BEGIN
        IF rst = 0 THEN
            o2 := n + 1
        ELSE
            o2 := 1
        END
        ;
        n := o2
    END
    ||
    o1 := n
```

Pour la clause OPERATIONS c'est un peu plus compliqué, on se référera au BBook [1] pour avoir l'ensemble des règles de calcul. La décomposition de $prd_{\{n,rst,o1,o2\}}$ en utilisant la définition à partir des transformateurs de prédicat produit la formule suivante :

$$\neg \left[\begin{array}{l}\text{IF } rst = 0 \text{ THEN} \\ \quad o2 := n+1 \\ \text{ELSE} \\ \quad o2 := 1 \\ \text{END}\end{array}\right] ([n := o2](o2', n' \neq o2, n)) \ \wedge \ \neg[o1 := n](o1' \neq o1)$$

La partie droite $\neg[o1 := n](o1' \neq o1)$ est le prédicat *avant-après* de la substitution $o1 := n$ et la première partie du prédicat ci-dessus est le prédicat *avant-après* de la composition séquentielle de la substitution conditionnelle IF et de la substitution simple $n := o2$. Notons que les variables sont différentes dans la partie gauche ($o2$ and n) et dans la partie droite, ceci est dû à la composition parallèle : chaque substitution peut modifier une certain ensemble de variables, et les deux ensembles doivent être disjoints.

En appliquant les substitutions dans la formule ci-dessus, nous obtenons le prédicat *avant-après* de l'événement de la clause OPERATIONS. La partie droite $\neg[o1 := n](o1' \neq o1)$ conduit à $o1' = n$. Dans la partie gauche $[n := o2](o2', n' \neq o2, n)$ se simplifie en $o2', n' \neq o2, o2$. L'application de la substitution IF amène à $o2', n' \neq n+1, n+1$ ($o2$ est remplacée par $n+1$) dans le cas où $rst = 0$ et à $o2', n' \neq 1, 1$ ($o2$ est remplacée par 1) dans l'autre cas. Ainsi, [IF ...]$[n := o2](o2', n' \neq o2, n)$ produit le prédicat $(rst = 0 \Rightarrow o2', n' \neq n+1, n+1) \wedge (rst \neq 0 \Rightarrow o2', n' \neq 1, 1)$. Il ne nous reste plus qu'à appliquer la négation sur ce résultat. On rappelle que la formule logique $(a \wedge \neg b) \vee (\neg a \wedge \neg c)$ est équivalente à la formule $(a \Rightarrow \neg b) \wedge (\neg a \Rightarrow \neg c)$.

$$prd(\text{OPERATIONS}) = \begin{pmatrix} rst = 0 \Rightarrow o2', n' = n+1, n+1 \ \wedge \\ rst = 1 \Rightarrow o2', n' = 1, 1 \end{pmatrix} \wedge o1' = n$$

9.1.5 Proposition d'introduction de la modularité en BHDL

Le langage BHDL est à l'origine définit comme un sous langage du langage B. Nous proposons d'ajouter deux nouveaux constructeurs au langage BHDL : les blocs sous-modèles et une substitution d'importation de modèle. Un bloc sous-modèle est un moyen de cacher des variables en restreignant leur visibilité à l'intérieur d'un bloc. Un bloc sous-modèle sert de spécification pour un sous-module et est destiné à être remplacé par une substitution d'importation. Une substitution d'importation permet à un modèle BHDL d'importer un autre modèle BHDL.

Bloc sous-modèle

Un bloc sous-modèle a pour but de servir de spécification pour un sous-module du circuit modélisé par le modèle BHDL. Concrètement, le bloc sous-modèle permet de réaliser cela en restreignant la portée d'une variable. Ces variables constituent l'état interne du module modélisé par le bloc et sont invisibles pour le reste du modèle, la portée des variables de bloc est restreinte au bloc dans lequel elles sont déclarées. Ces variables ne sont pas comparables aux variables locales introduites par la substitution ANY du langage B. Pour une substitution ANY, les valeurs des variables locales sont indépendantes d'une application de la substitution à l'autre. La sémantique d'un bloc sous-modèle est qu'à chaque application de la substitution, les valeurs des variables de l'état interne de bloc sont conservées d'une application à l'autre : la valeur d'une variable *avant* la substitution est la valeur de cette variable *après* la dernière application de la substitution. Le chapitre 13 explique comment les blocs sont introduits dans le modèle.

Nous utilisons les blocs sous-modèles uniquement dans un modèle BHDL, et pas dans un modèle B plus abstrait. Rappelons qu'un modèle BHDL est constitué d'un seul événement (pouvant contenir un ou plusieurs blocs sous-modèles). Dans cette hypothèse, les variables cachées par les blocs sous-modèles se conduisent exactement de la même façon que si elles étaient déclarées globalement comme toutes les variables sauf qu'elles ne peuvent être lues ou écrites qu'à l'intérieur du bloc. Le bloc lui-même peut interagir avec le reste du système en lisant ou en modifiant les variables globales du système.

La syntaxe d'un bloc sous-modèle est montrée ci-dessous. Un bloc s'écrit en donnant la liste des variables internes au bloc avec leurs types. Le bloc contient deux substitutions : l'initialisation des variables internes qui est appliquée une seule fois à l'initialisation du système, et une clause OPERATIONS qui est la substitution appliquée à chaque cycle. La substitution de la clause OPERATIONS peut elle-même contenir d'autres blocs sous-modèles.

```
BLOCK
    v
TYPES
    types
INITIALISATION
    init
OPERATIONS
    S
END
```

Substitution d'importation

La substitution d'importation permet de réutiliser des modèles BHDL existants. La signification de cette substitution est de présenter certaines valeurs (li) aux entrées (pi) d'un composant C et d'affecter les valeurs présentées par les sorties (po) de C aux variables locales (lo). La syntaxe de la substitution est la suivante :

$$(po : lo) \leftarrow C(pi : li)$$

Pour ces deux nouvelles substitutions, il n'est pas possible de définir leurs sémantiques par un transformateur de prédicat comme les autres substitutions car ils permettent d'introduire des mémoires cachées. Par exemple, si on importe un composant qui contient certaines mémoires, la relation entre les entrées et les sorties n'est pas nécessairement la même à chaque cycle. Plus exactement, ce type de relation sur les variables ne peut pas s'exprimer dans la logique classique. Une logique temporelle ou l'expression d'une relation sur les traces séquentielles des variables est nécessaire. Le chapitre 11 donne une sémantique de BHDL est définissant des prédicats sur les traces des variables. La chapitre 12 définit une sémantique relationnelle sur les traces des variables et discute les problèmes introduits par les blocs sous-modèle et les substitutions d'importation.

Ces deux substitutions sont introduites pour permettre la modularité, la réutilisation de modèles de composants existants. Dans un processus de raffinement formel B, ces deux substitutions ne sont pas

utilisées. Un bloc sous-modèle peut être retiré du modèle sans en changer la sémantique en déclarant les variables internes comme des variables globales et en insérant au même endroit que le bloc dans le modèle BHDL la substitution de la clause OPERATIONS du bloc sous-modèle. Le bloc sous-modèle est introduit en fin de développement comme intermédiaire pour introduire les substitutions d'importation. Le chapitre 13 montre comment un bloc sous-modèle peut être remplacé par une substitution d'importation d'un modèle BHDL ayant la même sémantique que le bloc. Le bloc sous-modèle est utilisé en fin de processus de raffinement afin de séparer les variables et la description des sous-composants qui seront importés.

9.2 Ensembles supports d'un modèle BHDL

Un ensemble support est une collection de variables. Nous étendons la notion d'ensemble support introduite par Dunne dans [40] (*frames* en anglais) en définissant trois ensembles supports pour une substitution. Nous définissons les ensembles supports en lecture, en écriture et les ensembles supports total en écriture. L'ensemble support en écriture (resp. en lecture) est l'ensemble des variables qui sont écrites (resp. lues) par la substitution. L'ensemble support en écriture d'une substitution S est noté $write(S)$ et l'ensemble support en lecture est noté $read(S)$. L'ensemble support total en écriture est l'ensemble des variables qui sont toujours écrites par une substitution (dans tous les cas qui peuvent être introduits par une composition conditionnelle).

Dans [40], les ensembles supports correspondent aux variables qui sont écrites par une substitution et sont utilisés pour définir une théorie des substitutions généralisées permettant de différencier, par exemple, les substitution $skip$ et $x := x$ (ces deux substitutions ne sont pas différenciées dans la sémantique basée sur le calcul de plus faible précondition). Nous utilisons les ensembles supports pour vérifier la bonne formation des machines BHDL (utilisation correcte des entrées et des sorties par exemple). Nous utilisons également les ensembles supports pour interpréter le modèle BHDL comme un modèle de circuit physique, les ensembles supports permettent de calculer une partition des variables en plaçant d'un côté la variables qui modélisent des registres et de l'autre celles qui modélisent des fils.

Les variables écrites d'une substitution sont les variables qui sont potentiellement modifiées par S. Nous considérons comme potentiellement modifiée toute variable se situant en partie gauche d'une substitution simple : dans la substitution $x := E$, x est considérée comme une variable écrite, même si l'expression E est en fait égale à la valeur de x (par exemple, dans $x := x$ la variable x est considérée comme modifiée). L'ensemble support en écriture de l'exemple précédent est $\{x\}$. La définition est établie de manière purement syntaxique, une variable peut donc éventuellement être abusivement considérée comme écrite alors qu'elle n'est en fait pas modifiée (comme dans l'exemple $x := x$). Nous choisissons tout de même cette définition syntaxique car nous souhaitons pouvoir calculer automatiquement les ensembles supports et, dans le cas général, il n'est pas possible d'identifier automatiquement si une variable est effectivement modifiée ou pas (cela reviendrait à faire une preuve automatique d'une des propositions $x = E$ ou $\neg(x = E)$, ce qui n'est pas possible dans le cas général).

De la même manière, les variables lues sont les variables qui sont potentiellement lues par une substitution. Dans le cas simple $x := E$, les variables considérées comme lues sont les variables libres de l'expression E. Ici, encore, la définition est purement syntaxique.

Les ensembles supports ne sont pas définis que pour les substitutions simples mais pour toutes les substitutions de BHDL. On donne dans la suite de cette section les définitions formelles des ensembles supports et on montre comment ils peuvent être calculés de manière compositionnelle.

Afin de montrer comment les ensembles supports peuvent évoluer en fonction de la façon dont les substitutions sont composées, nous présentons d'abord deux exemples simples de substitutions dont nous calculons les ensembles supports. Ces exemples permettent en particulier de préciser ce qu'on appelle une variable lue par une substitution dans le cas général. Nous disons qu'une variable v est lue lorsque, pour pouvoir appliquer la substitution, il est nécessaire de connaître la valeur *avant* de v : les valeurs *après* des variables écrites dépendent, a priori, des valeurs *avant* des variables lues. Considérons les deux

substitutions S_1 et S_2 suivantes :

$$S_1 \equiv \begin{cases} a := 2; \\ b := a \end{cases} \qquad S_2 \equiv \begin{cases} b := a; \\ a := 2 \end{cases}$$

Nous avons $write(S_1) = \{a, b\}$ et $read(S_1) = \varnothing$ parce que la seule variable utilisée dans la partie droite d'une substitution simple est a dans la deuxième substitution, mais a est d'abord écrite par la première substitution : la valeur *avant* (avant S_1) de a n'est donc pas utilisée par S_1. En fait, lorsque dans une substitution une variable est écrite avant toute utilisation elle n'est pas considérée comme lue. La substitution S_1 positionne la variable a à 2 et la variable b également à 2.

La situation est différente dans le cas de la substitution S_2. Les ensembles supports sont $write(S_2) = \{a, b\}$ et $read(S_2) = \{a\}$ car la substitution utilise la valeur *avant* de a. En supposant que la valeur *avant* de a est a_0, S_2 positionne a à 2 et b à a_0.

Dans les sections suivantes nous définissions formellement les ensembles supports d'une machine BHDL. Nous donnons d'abord les définitions dans le cas, plus simple, où la composition conditionnelle (IF) n'est pas utilisée. Puis nous définissons la notion de chemin syntaxique afin de donner une définition des ensembles supports dans le cas général, en s'appuyant sur les premières définitions données dans le cas sans composition conditionnelle. Nous proposons ensuite une solution pour calculer les ensembles supports de manière automatique et compositionnelle. Pour cela nous définissons un troisième ensemble support, qui est l'ensemble des variables qui sont écrites dans tous les chemins syntaxiques.

9.2.1 Cas sans composition conditionnelle

Nous définissons ici les ensembles supports pour la plupart des substitutions. Le cas de la composition conditionnelle est discuté dans la section suivante. La substitution multiple $x, y := E, F$ est considérée comme une abréviation de $x := E \| y := F$. L'exposant s dans les notations $write^s$ et $read^s$ signifie qu'il s'agit ici de définitions dans le cas de la syntaxe simplifiée des substitutions sans composition conditionnelle.

Dans la définition des ensembles supports, pour la substitution de connexion à un composant importé, nous utilisons l'opérateur *local* qui fournit l'ensemble des variables locales d'une liste de connexions O. Nous donnons sa définition ci-dessous.

$$local(fp : lv) = \{lv\}$$
$$local(A, B) = local(A) \cup local(B)$$

Plus généralement, nous définissons l'opérateur $PortLoc$ qui donne une paire d'ensembles : l'ensemble des ports formels (entrées et sorties du composant importé), et l'ensemble des variables locales qui sont connectées à ces ports.

$$PortLoc(fp : lv) = (\{fp\}, \{lv\})$$
$$PortLoc(A, B) = \begin{cases} \text{let} \\ \quad (pa, la) = PortLoc(A) \\ \quad (pb, lb) = PortLoc(B) \\ \text{in} \\ \quad (pa \cup pb, la \cup lb) \end{cases}$$

Nous utilisons également l'opérateur *free* qui donne l'ensemble des variables libres d'une expression.

$free(E_1\ op\ E_2) = free(E_1) \cup free(E_2)$, où $op \in \{+, -, *, /, mod, <, >, \leq, =, \geq, \wedge, \vee, \Rightarrow\}$
$free(bool(E)) = free(E)$
$free(\lambda x.(P|E)) = (free(P) \cup free(E)) - \{x\}$

Définition 6 (Ensemble support en écriture sans composition conditionnelle) *L'ensemble support en écriture d'une substitution est l'ensemble des variables qui sont potentiellement modifiées par la substitution.*

$$\begin{aligned} write^s(x := E) &\equiv \{x\} \\ write^s(x :\in E) &\equiv \{x\} \\ write^s(A \| B) &\equiv write^s(A) \cup write^s(B) \\ write^s(A; B) &\equiv write^s(A) \cup write^s(B) \\ write^s((O) \leftarrow ic(I)) &\equiv local(O) \\ write^s(\mathcal{B}_v(I, S)) &\equiv write^s(S) - v \end{aligned}$$

Définition 7 (Ensemble de support en lecture sans composition conditionnelle) *L'ensemble support en lecture d'une substitution est l'ensemble des variables dont la substitution utilise la valeur* avant.

Il faut prêter attention au cas de la composition séquentielle $A;B$: si une variable v est lue par B mais est aussi écrite par A alors elle n'est pas considérée comme lue par la substitution $A;B$: l'effet de la substitution ne dépend pas de la valeur avant *de v, B utilise pour v la valeur qui lui est assignée par A.*

Finalement nous verrons que cette définition pour la composition séquentielle ne sera plus valable dans le cas où nous introduirons la composition conditionnelle.

$$
\begin{aligned}
read^s(x := E) &\equiv free(E) \\
read^s(x :\in E) &\equiv \varnothing \text{ , } E \text{ doit être une constante (un type)} \\
read^s(A \| B) &\equiv read^s(A) \cup read^s(B) \\
read^s(A;B) &\equiv (read^s(B) - write^s(A)) \cup read^s(A) \\
read^s((O) \leftarrow ic(I)) &\equiv local(I) \\
read^s(\mathcal{B}_v(I,S)) &\equiv read^s(S) - v
\end{aligned}
$$

9.2.2 La composition conditionnelle

Le cas de la composition conditionnelle IF n'est pas si évident car elle offre le *choix* entre deux alternatives qui n'ont pas, a priori, les mêmes ensembles supports.

Par ailleurs, il faut être conscient que l'implantation d'une composition conditionnelle dans un circuit électronique est légèrement différente d'une implantation en logiciel. Lorsqu'un logiciel est exécuté, dans la plupart des cas, la condition est évaluée puis le bloc de code adéquat est exécuté. Dans un circuit, les deux branches font partie du chemin de donnée du circuit et sont physiquement implantées, mais le résultat d'une seule des deux branches est utilisé, un multiplexeur est utilisé pour implanté le choix en fonction de l'évaluation de la condition, lorsqu'une variable est lue (resp. écrite) par une des deux branches, elle est toujours lue (resp. écrite), même si c'est le résultat de l'autre branche qui est utilisé. Cependant il ne suffit pas de définir les ensembles supports d'une composition conditionnelle comme l'union des ensembles supports des deux branches : la possibilité de composer séquentiellement des substitutions qui peuvent contenir des compositions conditionnelles complique la définition des ensembles supports. On peut l'observer sur l'exemple suivant :

$$\text{IF } x = 1 \text{ THEN } v := 2 \text{ ELSE } u := 3 \text{ END } ; t := v$$

Il est clair que la variable v est écrite par la substitution compositionnelle ; mais dans ce cas, en utilisant les définitions précédentes des ensembles supports d'une composition séquentielle, l'ensemble support en lecture de la substitution complète ne contient pas v. Or dans le cas où $x \neq 1$, la valeur *avant* de v est utilisée, donc v doit être considérée comme lue.

La composition conditionnelle crée plusieurs alternatives, plusieurs *chemins*. Nous utilisons le terme *chemin* en référence aux langages impératifs dans lesquels un chemin est une séquence d'opérations qui sont exécutées lors d'une exécution d'un programme. Il peut exister plusieurs chemins lorsqu'on utilise des opérations conditionnelles (comme IF), le chemin qui est exécuté par le programme dépend de la valeur des variables.

L'idée dans le calcul des ensembles supports d'une substitution est de considérer à chaque fois le pire chemin. Une variable est considérée écrite (resp. lue) lorsqu'il existe un chemin dans lequel la variable est écrite (resp. lue). Une variable n'est donc pas considérée comme écrite si elle n'est écrite (resp. lue) dans aucun chemin. Évidemment, le pire chemin ne sera a priori pas le même pour chaque variable ou pour les deux ensembles supports.

Cette approche a cependant une limitation. Il n'est généralement pas possible de calculer statiquement quel est exactement l'ensemble des chemins effectivement possibles. Ceci est dû en particulier au fait que les conditions des substitutions conditionnelles peuvent être éventuellement très complexes et il n'est généralement pas possible de savoir de quelles manières elles dépendent les unes des autres (ont-elles une intersection nulle ? Y a-t-il des implications ? des équivalences ? ...). Nous donnons un exemple en fin de section.

Nous nous basons alors sur des définitions s'appuyant uniquement sur la syntaxe. Nous définissons la notion de *chemin syntaxique*. Intuitivement, l'ensemble des chemins syntaxiques est l'ensemble des chemins qui sont *a priori* possibles d'un point de vue strictement syntaxique : on considère tous les cas produits par les substitutions conditionnelles, même si certains cas ne sont en fait pas possibles si on considère la sémantique. C'est-à-dire que nous n'analysons pas les conditions des substitutions conditionnelles. Un chemin est exprimé comme une substitution qui ne contient pas de composition conditionnelle.

Par exemple, les deux chemins syntaxiques de la substitution $y := 0$; IF $y = 0$ THEN $u := 1$ ELSE $u := 2$ END sont, d'une part, le chemin $y := 0; \xi := bool(y = 0); u := 1$ et, d'autre part, le chemin $y := 0; \xi := bool(y = 0); u := 2\}$, bien que le second ne puisse en fait jamais se produire puisque y est d'abord positionné à 0 et c'est donc toujours la première branche du IF qui est choisie. On remarquera l'utilisation d'une variable supplémentaire ξ et la substitution $\xi := bool(...)$ de façon à éviter de perdre l'information concernant les variables lues par les conditions des substitutions conditionnelles dans l'expression des chemins.

Définition 8 (Chemins syntaxiques d'une substitution) *Nous définissons l'opérateur* **spaths** *qui donne l'ensemble des chemins syntaxiques d'une substitution. Cet opérateur est défini inductivement sur la structure d'une substitution.*

$spaths(x := E) = \{x := E\}$
$spaths(x :\in E) = \{x :\in E\}$
$spaths((O) \leftarrow ic(I)) = \{(O) \leftarrow ic(I)\}$
$spaths(\mathcal{B}_v(I, S)) = \{\mathcal{B}_v(I, S_p) | S_p \in spaths(S)\}$
$spaths(\text{IF } \mathcal{C} \text{ THEN } S \text{ ELSE } T \text{ END})$
$= \{\xi := bool(\mathcal{C}); S_p | S_p \in spaths(S)\} \cup \{\xi := bool(\mathcal{C}); T_p | T_p \in spaths(T)\}$
$spaths(\text{IF } \mathcal{C} \text{ THEN } S \text{ END})$
$= \{\xi := bool(\mathcal{C}); S_p | S_p \in spaths(S)\} \cup \{\xi := bool(\mathcal{C})\}$

Nous utilisons la notation $\star$ pour désigner une composition binaire de substitutions (composition séquentielle ; ou composition parallèle $\|$).

$spaths(S \star T) = \{S_p \star T_p | S_p \in spaths(S) \wedge T_p \in spaths(T)\}$

Propriété 2 (Quantifications sur les chemins)

A partir des définitions ci-dessus nous pouvons prouver les propriétés suivantes, pour toutes les substitutions S et T, toute composition $\star$ ($\star$ est ; ou $\|$) et tout prédicat $\mathcal{P}$:

$$\exists p \cdot (p \in spaths(S \star T) \wedge \mathcal{P}(p)) \Leftrightarrow \exists (S_p, T_p) \cdot (S_p \in spaths(S) \wedge T_p \in spaths(T) \wedge \mathcal{P}(S_p \star T_p))$$

$$\forall p \cdot (p \in spaths(S \star T) \Rightarrow \mathcal{P}(p)) \Leftrightarrow \forall (S_p, T_p) \cdot (S_p \in spaths(S) \wedge T_p \in spaths(T) \Rightarrow \mathcal{P}(S_p \star T_p))$$

La composition conditionnelle a les propriétés équivalentes :

$$\exists p \cdot (p \in spaths(\text{IF } \mathcal{C} \text{ THEN } S \text{ ELSE } T \text{ END}) \wedge \mathcal{P}(p)) \Leftrightarrow \exists (S_p, T_p) \cdot (S_p \in spaths(S) \wedge T_p \in spaths(T) \wedge (\mathcal{P}(\xi := bool(\mathcal{C}); S_p) \vee \mathcal{P}(\xi := bool(\mathcal{C}); T_p))$$

$$\forall p \cdot (p \in spaths(\text{IF } \mathcal{C} \text{ THEN } S \text{ ELSE } T \text{ END}) \Rightarrow \mathcal{P}(p)) \Leftrightarrow \forall (S_p, T_p) \cdot (S_p \in spaths(S) \wedge T_p \in spaths(T) \Rightarrow (\mathcal{P}(\xi := bool(\mathcal{C}); S_p) \wedge \mathcal{P}(\xi := bool(\mathcal{C}); T_p))$$

Les substitutions composant un chemin ne contiennent pas de composition conditionnelle. Ainsi, les ensembles supports d'un chemin peuvent être définis comme cela est expliqué dans la section 9.2.1. Cependant, nous faisons l'hypothèse implicite que la variable supplémentaire ξ n'est jamais utilisée, nous pouvons donc la retirer systématiquement des ensembles supports calculés à partir de chemins.

Définition 9 (Ensemble support en lecture) *L'ensemble support en lecture d'une substitution est un sous-ensemble des variables d'état du système. Une variable v fait partie de l'ensemble support en lecture d'une substitution S si S a un chemin dont l'ensemble support en lecture contient v (la variable v est lue dans au moins un des chemins de S).*

$$v \in read(S) \equiv v \neq \xi \wedge \exists p \cdot (p \in spaths(S) \wedge v \in read^s(p))$$

Définition 10 (Ensemble support en écriture) *L'ensemble support en écriture d'une substitution est un sous-ensemble des variables d'état du système. Une variable v fait partie de l'ensemble support en écriture d'une substitution S si S a un chemin dont l'ensemble support en écriture contient v (la variable v est écrite dans au moins un des chemins de S).*

$$v \in write(S) \equiv v \neq \xi \wedge \exists p \cdot (p \in spaths(S) \wedge v \in write^s(p))$$

En faisant l'hypothèse implicite que la variable supplémentaire ξ est automatiquement retirée (on aura toujours $v \neq \xi$) des ensembles supports, on écrira plus simplement $v \in read(S) \equiv \exists p \cdot (p \in spaths(S) \wedge v \in read^s(p))$, et $v \in write(S) \equiv \exists p \cdot (p \in spaths(S) \wedge v \in write^s(p))$. Pour adopter une définition plus ensembliste, nous écrirons également $read(S) = \{v | \exists p \cdot (p \in spaths(S) \wedge v \in read^s(p))\}$, et $write(S) = \{v | \exists p \cdot (p \in spaths(S) \wedge v \in write^s(p))$.

La principale différence avec les ensembles supports $read^s$ et $write^s$, définis sur la syntaxe simplifiée dans la section 9.2.1, se situe au niveau de l'ensemble support en écriture pour la composition séquentielle. En effet, pour deux substitutions S et T, l'ensemble support simplifié $read^s(S;T)$ est défini en fonction de l'ensemble support en écriture de S, le principe consistant à prendre les variables lues par S et les variables lues par T qui ne sont pas écrites par S. Nous verrons que dans la syntaxe non simplifiée, on ne peut pas raisonner de manière aussi simple. En effet une variable est écrite par S s'il y a un chemin de S dans laquelle elle est écrite, il peut donc y avoir des chemins dans lesquels elle n'est pas écrite. Si cette variable est lue par T (on suppose, pour l'exemple, qu'elle n'est pas lue par S), on ne peut l'enlever de l'ensemble support en lecture de $S;T$ que si on est certain que T n'utilisera jamais la valeur *avant* $S;T$ de la variable, ce qui n'est pas le cas s'il existe des chemins de S qui n'écrivent pas cette variable, même si elle est dans l'ensemble support en écriture de S. Nous aurons donc besoin de connaître l'ensemble des variables qui sont *toujours* écrites (écrites dans tous les chemins) par une substitution.

En prenant en compte ce principe, les ensembles supports peuvent être calculés de façon compositionnelle. Pour montrer cela nous avons besoin de définir un troisième ensemble support, noté *writeall*, qui est l'ensemble des variables qui sont écrites dans tous les chemins de la substitution.

Définition 11 (Ensemble support total en écriture) *L'ensemble support total en écriture est l'ensemble des variables qui sont écrites dans tous les chemins de la substitution.*

$$v \in writeall(S) \equiv v \neq \xi \wedge \forall p \cdot (p \in spaths(S) \Rightarrow v \in write^s(p))$$

Plus formellement, nous devrions définir $v \in writeall(S)$ par $spaths(S) \neq \varnothing \wedge \forall p \cdot (p \in spaths(S) \Rightarrow v \in write^s(p))$ pour éviter que $writeall(S)$ soit l'ensemble de toutes les variables dans le cas où $spaths(S) = \varnothing$. En effet, il pourrait sembler plus correct que $writeall(S) = \varnothing$ dans ce cas. Cependant, pour toute substitution S, $spaths(S)$ n'est en fait jamais vide. La condition $spaths(S) \neq \varnothing$ peut donc être retirée. En considérant implicite la condition $v \neq \xi$, nous écrirons également, pour simplifier les écritures, $v \in writeall(S) \equiv \forall p \cdot (p \in spaths(S) \Rightarrow v \in write^s(p))$ ou encore $writeall(S) = \{v | \forall p \cdot (p \in spaths(S) \Rightarrow v \in write^s(p))\}$.

Propriété 3 (Compositionnalité des ensembles supports) *Les trois ensembles supports peuvent être calculés ensemble de manière compositionnelle sur la structure de la substitution.*

La façon de calculer de manière compositionnelle les ensembles supports est résumée dans le tableau 9.1. Les preuves de la correction de ces définitions par rapport aux définitions 9, 10 et 11 sont données dans l'annexe B. Notez que, pour toute substitution S, nous avons $writeall(S) \subseteq write(S)$.

Notations

Dans le reste de ce document, pour toute substitution S, nous noterons par R_S l'ensemble support en lecture de S, par W_S l'ensemble support en écriture et par W_S^a l'ensemble support total en écriture, tels qu'ils sont définis ci-dessus.

TAB. 9.1 – Définitions compositionnelles des ensembles supports

Substitution	$write$	$writeall$	$read$
$x := E$	$\{x\}$	$\{x\}$	$free(E)$
$x :\in E$	$\{x\}$	$\{x\}$	$\varnothing$
$A \| B$	$write(A) \cup write(B)$	$writeall(A) \cup writeall(B)$	$read(A) \cup read(B)$
$A; B$	$write(A) \cup write(B)$	$writeall(A) \cup writeall(B)$	$read(A) \cup (read(B) - writeall(A))$
IF P THEN A ELSE B END	$write(A) \cup write(B)$	$writeall(A) \cap writeall(B)$	$free(P) \cup read(A) \cup read(B)$
IF P THEN A END	$write(A)$	$\varnothing$	$free(P) \cup read(A)$
$(O) \leftarrow ic(I)$	$local(O)$	$local(O)$	$local(I)$
$\mathcal{B}_v(I, S)$	$write(S) - v$	$writeall(S) - v$	$read(S) - v$

9.2.3 Remarques

Dans certains cas, nous pouvons avoir certaines variables dans l'ensemble support en lecture alors que cette variable n'est en fait jamais effectivement lue. Dans l'exemple ci-dessous, l'ensemble support en lecture est $\{x, n, u\}$.

IF $x = 1$ THEN $n := 2$ ELSE $u := 3$ END ; IF $x = 1$ THEN $a := n$ ELSE $a := u$ END

En fait, les valeurs *avant* de n et u ne sont jamais utilisées. Lorsque x vaut 1, n est d'abord positionné à 2 et ensuite a prend la valeur de n, c'est-à-dire 2. Le même raisonnement peut être fait pour la variable u lorsque x ne vaut pas 1. Cette situation vient du fait que nous nous basons uniquement sur la syntaxe pour calculer les ensembles supports. Pour calculer au plus juste les ensembles supports il serait nécessaire de se baser sur la sémantique mais cela n'est pas possible en général. Pour s'en convaincre, on pourrait par exemple remplacer la première condition $x = 1$ par un prédicat testant si x est un nombre pair strictement positif et la deuxième condition $x = 1$ par un prédicat stipulant que x est la somme de deux nombres premiers. La conjecture de Goldbach forte donnée par Euler nous dit que tout nombre pair strictement positif est la somme de deux nombres premiers, mais ceci n'est, à l'heure actuelle, pas prouvé.

Guide de modélisation 1 Un bon moyen d'éviter ce genre de problème est d'organiser le code de telle façon qu'il reflète la façon dont l'espace des états est partitionné. Ceci peut être fait en emboîtant des composition conditionnelle :

```
IF P THEN
   IF Q THEN
      Substitution for P ∧ Q
   ELSE
      Substitution for P ∧ ¬Q
   END
ELSE
   IF Q THEN
      Substitution for ¬P ∧ Q
   ELSE
      Substitution for ¬P ∧ ¬Q
   END
END
```

Le problème des variables dans l'ensemble support en lecture mais jamais effectivement lues est évité si P et Q sont des prédicats qui n'utilisent pas les mêmes variables (par exemple P est un prédicat sur une variable x, et Q un prédicat sur une variable y).

Guide de modélisation 2 Dans le cas où la substitution à appliquer est la même dans deux situations différentes, il peut être utile (afin d'éviter des répétitions inutiles d'une partie du circuit électronique modélisé) d'utiliser la composition conditionnelle uniquement pour le contrôle ou les branchements, et d'écrire la partie opérationnelle en dehors de la composition conditionnelle. Dans l'exemple ci-dessous, dans le premier cas nous calculons $x + y$, et dans le deuxième cas nous calculons $x + z$.

IF $\mathcal{P}$ THEN
  $i := y$
ELSE
  $i := z$
END
;
$o := x + i$

Dans le cas de l'ensemble support en écriture, le seul moyen de produire un problème semblable (c'est-à-dire une variable qui est dans l'ensemble support en écriture mais jamais effectivement écrite) est d'écrire une composition conditionnelle dont une des alternatives est impossible. Ce cas n'est pas non plus analysable statiquement en général.

Il est cependant possible que certaines variables soient écrites mais ne soient jamais utilisées. Dans un modèle BHDL, c'est notamment le cas des variables modélisant les sorties du circuit, qui doivent par ailleurs être écrites dans tous les chemins. Si la variable en question n'était pas une sortie, cela pourrait être considéré comme une erreur de modélisation.

9.2.4 Classification des variables en utilisant les ensembles supports

En utilisant les ensembles supports, les variables peuvent être classifiées comme suit. Nous donnons avec cette classification comment interpréter les variables du point de vue du circuit physique qui est modélisé.

- Les variables qui sont lues mais jamais écrites : $read(S) - write(S)$. Ces variables sont les entrées du circuit.
- Les variables qui sont écrites mais qui ne sont pas lues : $write(S) - read(S)$. Ces variables modélisent les fils internes du circuit physique. De telles variables peuvent aussi modéliser les sorties du circuit si elles sont également écrites dans tous les chemins : $writeall(S) - read(S)$. Si ce n'est pas le cas, elles modélisent nécessairement des fils internes.
  - $s \in writeall(S) - read(S) \Rightarrow s$ est un fil interne ou une sortie
  - $s \in (write(S) - writeall(S)) - read(S) \Rightarrow s$ est un fil interne.
- Les variables qui sont à la fois lues et écrites : $write(S) \cap read(S)$. Si on regarde la définition des ensembles supports, ceci signifie que ces variables sont écrites et qu'elles sont lues avant d'être écrites. Lorsqu'une variable est lue avant d'être écrite, cela veut dire que la valeur qui est utilisée lors de la lecture est la valeur écrite lors du cycle précédent : temporellement, la dernière écriture de la variable se situe nécessairement dans le cycle précédent. Ainsi, ces variables doivent être implantées par des registres.

Cette classification est faite automatiquement par le calcul des ensembles supports du modèle BHDL. Elle peut être utilisée pour vérifier que ce que le modélisateur a spécifié comme entrées/sorties/variables est correct (cf. section 9.3). La classification est aussi utilisée pour déterminer automatiquement, pendant la traduction vers des langages de description de circuits, quelles variables seront des fils et quelles variables seront des registres du circuit.

En plus de la classification donnée ci-dessus qui est obtenue par le calcul des ensembles supports de la substitution principale du modèle BHDL (clause OPERATIONS), le modélisateur peut imposer à une

variable de devenir un registre en l'initialisant de façon déterministe dans la clause INITIALISATION. Cela a essentiellement été ajouté aux règles de traductions afin de donner un sens à une initialisation sur une variable qui a priori devrait être un fil : un fil ne peut pas être initialisé ! On aurait tout aussi bien pu considérer cela comme une erreur de modélisation.

9.3 Bonne formation d'un modèle BHDL

Les conditions de bonne formation d'un modèle BHDL reprennent les conditions de bonne formation d'une machine B classique. Nous ajoutons cependant un certain nombre de conditions qui sont nécessaires pour prendre en compte le fait qu'un modèle BHDL modélise un circuit électronique.

Nous utilisons l'opérateur *type* permettant de donner le type d'un variable tel qu'il est donné dans l'invariant. Nous rappelons que dans un modèle BHDL, une variable ne doit être typée qu'une seule fois dans l'invariant. L'expression $type(v, inv)$ retourne le type de la variable v tel qu'il est donné dans l'invariant inv. Nous définissons aussi l'opérateur *typed* qui nous si une variable est typée ou non.

$$
\begin{array}{l}
type(s, s \in F) = F \\
type(s, P\&Q) = type(s, P) \text{ si } typed(s, P) \\
type(s, P\&Q) = type(s, Q) \text{ si } typed(s, Q) \\
typed(s, s \in F) = true \\
typed(x, s \in F) = false \text{ if } x \neq s \\
typed(s, P\&Q) = typed(s, P) \vee typed(s, Q)
\end{array}
$$

Les conditions de bonne formation qui sont ajoutées à celles déjà définies pour un modèle B classique sont listées ci-dessous.

- Une entrée ne peut pas être écrite. Cela n'aurait pas de sens, en BHDL nous ne considérons pas qu'un port puisse être à la fois une entrée et une sortie. Certains langages de modélisation de circuit le permettent mais cela est ensuite transformé pour donner un circuit synthétisable. Le langage BHDL se situe directement au niveau synthétisable.

$$i \cap write(C) = \varnothing$$

- Une sortie ne peut pas être lue. Compte tenu de la définition des ensembles supports, cela signifie qu'une variable modélisant une sortie ne peut pas être lue avant d'avoir été écrite. En effet, si une variable est écrite (dans tous les chemins), elle peut être lue ensuite sans être considérée comme lue par la substitution globale. La valeur qui est lue est alors la valeur qui a été écrite avant. Ce qui est lu n'est donc pas la sortie mais un fil qui est généré automatiquement (cf. traduction de la composition séquentielle, section 14.5).

$$o \cap read(C) = \varnothing$$

- Les variables modélisant les entrées et les sorties du circuit ne doivent pas être initialisées de façon déterministe. L'opérateur $write_d$ donne l'ensemble des variables qui sont écrites de façon déterministe par une substitution, *init* est la substitution de la clause INITIALISATION.

$$(i \cup o) \cap write_d(init) = \varnothing$$

- Toutes les variables doivent être initialisées (de façon non déterministe pour les entrées/sorties). Cette condition est identique à celle des machines B classiques mais nous la réécrivons en termes d'ensemble support.

$$i \cup o \cup v = write(init)$$

- Les sorties doivent être écrites par tous les chemins syntaxiques. Si une sortie n'était pas écrite dans un chemin, cela signifierait qu'elle ne serait pas spécifiée pour tous les cas possibles. Or une sortie d'un circuit a toujours une valeur, il faut donc que celle-ci soit spécifiée.

$$o \subseteq writeall(C)$$

- Lorsqu'une expression est affectée à une variable, le type de cette expression doit être un sous-ensemble du type de la variable. Ceci est nécessaire pour s'assurer par exemple qu'on n'affecte pas un nombre implanté sur n bits à une variable implantée sur m bits avec $m < n$. Pour chaque substitution simple $x := E$ ou $x :\in E$, la condition suivante doit être vérifiée :

$$type(E, inv) \subseteq type(x, inv)$$

Il faut remarquer la différence entre cette condition et les obligations de preuves générées automatiquement par la méthode B. En B, il faut prouver que la *valeur* de l'expression fait partie du type de la variable. C'est-à-dire qu'il faut prouver que, dans tous les cas de figure qui peuvent se présenter, l'expression est toujours évaluée à une valeur faisant partie du type de la variable. Cette condition est plus faible que celle proposée ici. La condition que nous imposons est statique, elle ne dépend pas de la valeur à laquelle l'expression sera finalement évaluée. Par exemple, si une variable y est déclarée comme étant un nombre sur 8 bits et une autre variable x est déclarée comme étant un nombre sur 5 bits, on ne pourra pas écrire ne substitution comme $y := 2; x := y$ car le type de y n'est pas inclus dans le type de x. En B cette substitution serait correcte car on peut prouver que la valeur de y est toujours 2 et que 2 fait bien partie des nombres représentables sur 5 bits. En BHDL on doit s'imposer une telle condition statique car nous modélisons des circuits et le modèle est destiné à être traduit dans un langage de description de circuit. Or, une substitution simple comme $x := y$ modélise une connexion de fils, et, dans un circuit électronique, les connexions sont établies de manière statique, lors de la fabrication. On ne peut pas connecter directement 5 bits sur 3 bits. Si c'est réellement ce qu'on veut faire, il faudra introduire un convertisseur qui réalisera cette connexion et spécifiera de quelle manière elle doit être faite exactement.
On peut éventuellement se permettre d'écrire $y := x$ car la plupart des langages de description de circuit sont capables de gérer les conversions implicites. Si on souhaite empêcher les conversions implicites, on peut ajouter la condition que pour toute substitution simple on doit aussi avoir $type(x, inv) \subseteq type(E, inv)$. On peut la combiner avec la condition précédente pour obtenir une seule égalité à prouver :

$$type(x, inv) = type(E, inv)$$

- La clause INVARIANT doit contenir le typage de toutes les variables.

$$\text{Pour toute variable } v,\ typed(v, inv)$$

- Pour tout bloc $\mathcal{B}_v(I, S)$, toutes les variables écrites par S doivent être écrites dans tous les chemins de S. Ceci est dû à la sémantique d'un bloc : un bloc modélise un sous composant, et un bloc doit toujours pouvoir être remplacé par une importation de composant. De ce fait, une variable écrite dans un bloc (outre les variables strictement internes au bloc) est considérée comme une sortie du sous composant qu'il modélise. Cette condition reprend la condition de bonne formation d'un modèle BHDL qui dit qu'une sortie doit être écrite dans tous les chemins.

$$\text{Pour tout bloc } \mathcal{B}_v(I, S),\ W_S - v = W_S^a - v$$

9.4 Conclusion

Ce chapitre a introduit la méthode B en présentant les substitutions généralisées et leur sémantique par prédicat *avant-après* ou par un modèle ensembliste. Ces deux styles de sémantique seront utilisés pour définir plus loin deux sémantiques de BHDL.

Nous avons défini des ensembles supports d'une substitution, généralisation des ensembles supports (*frames* en anglais) introduits par Dunne dans [40]. Pour les définir nous avons introduit la notion de chemin syntaxique, permettant d'identifier tous les cas de figure pouvant être introduits par des compositions conditionnelles.

Nous avons précisé que cette notion de chemin syntaxique est en fait trop faible pour obtenir une définition 'parfaite' des ensembles supports, mais il est en fait impossible de pouvoir calculer automatiquement les ensembles supports réels. Le problème que l'on peut rencontrer avec notre définition basée sur la syntaxe est de considérer des variables comme lues (ou écrites) alors qu'elles ne le sont jamais effectivement. Au pire, compte tenu de l'utilisation que nous faisons de ces ensembles supports, cela peut conduire à considérer certaines variables comme modélisant des registres alors que ceux-ci ne sont en fait pas nécessaires. Ceci n'a aucune influence sur la fonctionnalité du circuit, il comportera simplement des registres non utilisés. Nous avons proposé des guides de modélisation permettant d'éviter ce genre de cas.

Nous utilisons les ensembles supports pour classifier les variables et calculer automatiquement lesquelles modélisent des mémoires (des registres) et lesquelles modélisent des fils du circuit électronique qui est modélisé. Ceci est basé sur le principe qu'une variable qui est à la fois lue et écrite est une mémoire car la valeur lue est nécessairement la valeur calculée dans le cycle d'horloge précédent.

Nous utilisions également les ensembles supports pour définir des conditions de bonne formation supplémentaires à celle d'une machine B classique. Outre les conditions liées aux entrées et aux sorties la principale différence avec B est que pour chaque substitution simple (du type $x := E$), les types de la variable et de l'expression doivent correspondre et ceci doit être déterminé syntaxiquement. Contrairement à la méthode B classique qui génère des obligations de preuve demandant de prouver que l'évaluation de l'expression conduit dans tous les cas à une valeur compatible avec le type de la variable. On peut considérer qu'en BHDL il faut que la relation entre les types de la variable et de l'expression soit bien formée pour toutes les valeurs que le type de l'expression contient, y compris les valeurs que l'expression ne prendra jamais.

Chapitre 10

Définir des sémantiques de traces

L'objectif de ce chapitre est de présenter les éléments de base permettant de définir des sémantiques de trace pour le langage BHDL, nous définissons ce qu'est une trace et quelques opérateurs sur les traces qui seront utilisés dans les chapitres suivants.

Un circuit synchrone fonctionne par cycles d'horloge. À chaque cycle d'horloge, nous associons une valeur à chaque élément du circuit (registres et fils). À un cycle donné, la valeur d'un registre est la valeur qu'il délivre constamment pendant toute la durée du cycle ; la valeur d'un fil est la valeur qu'il porte lorsque le circuit est stabilisé (en général vers la fin du cycle). À chaque entrée, sortie et registre on associe la séquence qui modélise l'historique des valeurs portées pour chaque cycle, cette séquence est ce qu'on appelle une trace. La définition d'une sémantique en se basant sur la séquence des valeurs des variables se rapproche de ce qui est fait dans les logiques temporelles, par exemple dans la logique temporelle des actions, ou TLA, dont la sémantique est définie par Lamport dans [63].

Nous donnons d'abord quelques définitions nous permettant de définir une sémantique sur les traces que nous illustrerons par un petit exemple. Ensuite nous définissons des opérateurs nous permettant de manipuler plus facilement les traces.

10.1 Définitions

Nous définissons les concepts qui sont utilisés pour définir les sémantiques de BHDL qui sont basées sur les traces. Certaines de ces définitions viennent du livre [45] qui définit des sémantiques formelles pour VHDL.

Définition 12 (Trace) *Une trace est une séquence finie ou infinie de données qui modélise l'historique des valeurs qui sont portées par un élément. Si D est un ensemble de données, D^* dénote l'ensemble de toutes les séquences finies sur D et D^∞ est l'ensemble des séquences infinies sur D. L'ensemble $D^\omega = D^* \cup D^\infty$ est l'ensemble de toutes les séquences, finies ou infinies. Pour un élément donné x, on note sa trace par $x^\omega = \langle x_0, x_1, ... \rangle$.*

La notation x^ω signifie que x^ω est une trace qui peut être finie ou infinie, mais la longueur n'est pas spécifiée explicitement. Si nous voulons être plus précis sur la longueur d'une trace, on peut utiliser la notation x^l pour une de trace des valeurs de l'élément x et de longueur l, où l peut éventuellement être ∞ lorsque la trace est infinie.

Par exemple, supposons une exécution sur trois cycles et une entrée i qui vaut 1 au premier cycle, puis 3 au deuxième et enfin 7 au troisième cycle. La trace de i pour ces trois cycles est $i^\omega = \langle 1, 3, 7 \rangle$.

Pour un n-uplet T d'éléments de circuit qui peuvent porter des valeurs, en utilisant la définition ci-dessus T^ω est une trace de n-uplet de valeurs. Par exemple, $(i, o)^\omega = \langle (1, 2), (3, 1), (7, 6) \rangle$ est une trace possible pour la paire d'éléments (i, o). Nous définissons l'opérateur $unzip^n \in (D^n)^\omega \rightarrow (D^\omega)^n$ qui associe un n-uplet de traces à chaque trace de n-uplet. Par exemple, $unzip^2((i, o)^\omega = \langle (1, 2), (3, 1), (7, 6) \rangle) = (i^\omega = \langle 1, 3, 7 \rangle, o^\omega = \langle 2, 1, 6 \rangle)$. L'opérateur *unzip* se définit de la manière suivante :

$unzip^n(\langle\rangle) = \langle\rangle^n$ où $\langle\rangle^n$ est le n-uplet de séquences vides
$unzip^n(\langle(e_1, ..., e_n)|T\rangle) = (\langle e_1|T_1\rangle, ..., \langle e_n|T_n\rangle)$ où $\forall i \in 1..n \quad T_i = pos(i, unzip^n(T))$

Le constructeur $\langle e|s^\omega\rangle$ donne la trace qui a e comme élément initial suivi des éléments de la trace s^ω. L'opérateur $pos(i,t)$ donne l'élément qui a la position i dans le n-uplet t (projection).

$$\forall (e, s^\omega) \cdot (e \in E \wedge s^\omega = \langle s_0, s_1, ...\rangle \in E^\omega \Rightarrow \langle e|s^\omega\rangle = \langle e, s_0, s_1, ...\rangle)$$

$$pos(i, (t_1, ..., t_n)) = t_i$$

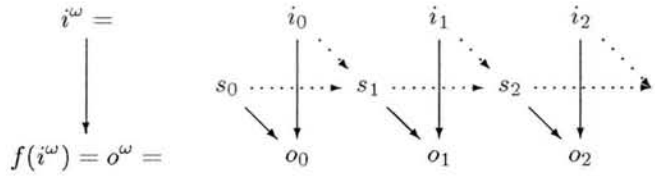
On peut prouver que $unzip^n$ définit une bijection entre $(D^n)^\omega$ et $(D^\omega)^n$. À partir de maintenant, on considèrera que T^ω, où T est un n-uplet de variables, se réfère de préférence à un n-uplet de traces (élément de $(D^\omega)^n$) des variables de T .

Définition 13 (Fonctions de traces) *Une fonction de traces associe des traces de sorties à des traces d'entrée. Une fonction de traces est dans un ensemble de fonctions* $I_1^\omega \times ... \times I_n^\omega \rightarrow O_1^\omega \times ... \times O_m^\omega$ *où* $I_1, ..., I_n$ *(resp.* $O_1, ..., O_m$*) sont les ensembles dans lesquels les entrées (resp. les sorties) prennent leurs valeurs.*

Pour un circuit modélisé par une machine de Mealy (cf. section 1.3), étant données les trois fonctions c (calcul des sorties), $next$ (calcul du nouvel état) et $reset$ (initialisation de l'état), la fonction f qui associe aux traces d'entrée i^ω les traces de sortie o^ω se calcule en appliquant les fonctions c et $next$ de façon répétitive. On peut définir cette fonction f par le prédicat suivant :

$$\begin{array}{l} \forall i^\omega = \langle i_0, i_1, ...\rangle \exists o^\omega = (o_0, o_1, ...) \quad f(i^\omega) = o^\omega \quad \wedge \\ \exists s^\omega = \langle s_0, s_1, ...\rangle \\ s_0 = reset \quad \wedge \quad \forall k \geq 0 \quad (o_k, s_{k+1}) = (c(i_k, s_k), next(i_k, s_k)) \end{array}$$

La trace s^ω représente la trace de l'état interne de la machine de Mealy. L'illustration ci-dessous montre de quelle manière les traces dépendent les unes des autres.



Définition 14 (Sémantique de traces prédicative) *La sémantique de traces prédicative d'un circuit est un prédicat qui définit l'*ensemble *des fonctions de traces qui représentent le comportement du circuit. Si l'ensemble de fonctions est vide, cela signifie qu'il n'y a aucune fonction satisfaisant le prédicat, le prédicat correspond à une spécification* inconsistante *du circuit. Si l'ensemble est un singleton, une seule fonction correspond à la spécification du circuit, cette dernière est donc consistante et* déterministe. *Si l'ensemble contient plus d'une fonction, la spécification est non déterministe ; c'est-à-dire que la spécification autorise plusieurs comportement différents pour le circuit.*

En supposant que les fonctions c, $next$ et $reset$ d'une machine de Mealy sont connues, le prédicat donné ci-dessus, comme illustration pour la définition d'une fonction de traces, constitue une sémantique de trace prédicative pour le circuit correspondant à la machine de Mealy définie par les fonctions c, $next$ et $reset$. Si ces trois fonctions sont bien définies (fonctions totales, ni partielles, ni relations), la sémantique est consistante et déterministe. Si on prend le problème dans l'autre sens, il est possible qu'une sémantique ne corresponde à aucune machine de Mealy si la sémantique n'est pas consistante. Une sémantique peut également définir plusieurs triplets de fonctions $(reset, c, next)$ si la sémantique n'est pas déterministe.

Notons que, formellement, une sémantique définit un *ensemble de fonctions*, chaque fonction étant déterministe (par définition de ce qu'est une fonction) ; pour une même entrée, la sortie est toujours la même. Cependant, la sémantique peut être non déterministe, ce qui résulte dans la définition d'un ensemble de fonctions qui représentent toutes les alternatives. Ceci pourrait également se représenter par

une relation au lieu d'un ensemble de fonctions. On peut voir cela sur l'exemple suivant, la sémantique est définie par le prédicat $\mathcal{P}$ ci-dessous.

$$\mathcal{P}(f) \equiv \forall i^\omega \exists o^\omega f(i^\omega) = o^\omega \wedge o_0 \in \{0,1\} \wedge \forall k \geq 1\ o_k = 1$$

Ce prédicat est non déterministe sur la première valeur de o. Dans l'ensemble de fonctions défini par $\mathcal{P}$, il y a une fonction qui positionne toujours o_0 à 0 et une fonction qui positionne toujours o_0 à 1. Cependant, il y a aussi des fonctions qui positionnent o_0 à 0 pour certaines entrées et à 1 pour d'autres entrées. On peut remarquer qu'on peut exprimer cela en donnant, à la place d'un ensemble de fonction, la relation $\bigcup_{i^\omega} \{i^\omega \mapsto \langle 0,1,1,...\rangle, i^\omega \mapsto \langle 1,1,1,...\rangle\}$. Ainsi, bien que la sémantique prédicative définisse un ensemble de fonctions, on peut aussi considérer qu'elle définit une relation.

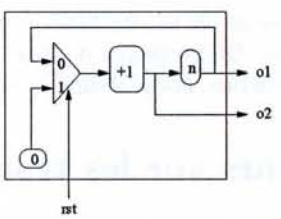
Définition 15 (Sémantique de traces relationnelle) *La sémantique de trace relationnelle d'un circuit est une relation qui associe les traces des sorties aux traces des entrées. Si la relation est une fonction (seulement une sortie pour chaque entrée), le circuit est déterministe.*

La différence entre la sémantique prédicative et la sémantique relationnelle est que la sémantique prédicative définit un prédicat qui définit lui-même un ensemble de fonctions (ou une relation) alors que la sémantique relationnelle définit directement une relation. La sémantique prédicative permet de garder une certaine structure dans le prédicat, elle peut être utilisée pour définir des traductions vers d'autres langages parce qu'elle est en fait une description logique du circuit. Par contre, la sémantique relationnelle perd la structure du circuit mais est plus simple à manipuler formellement, elle peut être utilisée pour faire des preuves. Dans les chapitres suivants, nous définissons ces deux types de sémantique pour le langage BHDL.

10.2 Illustration

Prenons l'exemple d'un compteur (une représentation graphique de la structure du circuit est donnée sur la figure 10.1). Ce compteur a une entrée rst qui réinitialise (réinitialisation synchrone) le compteur et deux sorties $o1$ et $o2$ qui portent la valeur du compteur. Un registre interne n est utilisé pour sauvegarder la valeur du compteur. La différence entre les deux sorties est que $o1$ est directement connecté au registre n alors que $o2$ est la sortie de la partie combinatoire qui calcule la valeur du registre.

FIG. 10.1 – Compteur avec signal de réinitialisation synchrone



La table sur la figure 10.2 montre les valeurs de ces éléments à chaque cycle (sur huit cycles). Le cycle numéroté 0 correspond à l'initialisation asynchrone initiale du système qui initialise les registres.

FIG. 10.2 – Exemple du compteur : valeur des éléments à chaque cycle

cycle	0	1	2	3	4	5	6	7
rst	0	0	0	0	1	0	0	0
n	0	1	2	3	4	1	2	3
$o1$	0	1	2	3	4	1	2	3
$o2$	1	2	3	4	1	2	3	4

Le comportement de ce circuit peut être modélisé par une fonction f qui associe à chaque trace d'entrée rst^ω les traces des sorties $o1^\omega$ et $o2^\omega$. Nous donnons ci-dessous une sémantique prédicative qui correspond au comportement illustré sur la figure 10.2. Ce prédicat reprend la structure du circuit représentée sur la figure 10.1.

$$
\begin{array}{l}
\forall rst^\omega = \langle rst_0, rst_1, ... \rangle \\
\exists (o1^\omega = \langle o1_0, o1_1, ... \rangle, o2^\omega = \langle o2_0, o2_1, ... \rangle) \\
f(rst^\omega) = (o1^\omega, o2^\omega) \wedge \\
\exists n^\omega = \langle n_0, n_1, ... \rangle \\
rst_0 \in \{0, 1\} \wedge \\
o1_0 \in \mathbb{N} \wedge \\
o2_0 \in \mathbb{N} \wedge \\
n_0 = 0 \wedge \\
\forall k \geq 0 \begin{pmatrix} rst_k = 0 \Rightarrow o2_k = n_k + 1 \wedge \\ rst_k = 1 \Rightarrow o2_k = 1 \wedge \\ o1_k = n_k \wedge \\ n_{k+1} = o2_k \end{pmatrix}
\end{array}
$$

Ce prédicat définit seulement une fonction, il est déterministe : si la trace d'entrée rst^ω est connue, les traces des sorties sont entièrement connues. En utilisant ce prédicat on peut calculer les traces des sorties et du registre interne à partir des traces des entrées. Ceci se fait en propageant les contraintes exprimées par le prédicat définissant la sémantique. Ainsi, si la sémantique prédicative n'est pas computationnelle en elle-même, elle produit un prédicat à partir duquel on peut calculer les sorties en fonction des entrées, il faut se doter pour cela d'un système de propagation de contraintes relativement simple puisque le prédicat est essentiellement constitué d'égalités et d'implications (toutes les alternatives sont prévues si le prédicat est déterministe).

Par exemple, on prend la trace d'entrée $rst^\omega = \langle rst_0 = 0, rst_1 = 0, rst_2 = 0, rst_3 = 1, rst_4 = 0, rst_5 = 0, rst_6 = 0 \rangle$. On calcule d'abord les valeurs pour le cycle 0.

- La sémantique spécifie que $n_0 = 0$ donc la valeur de n_0 est connue.
- La sémantique spécifie que si rst_0 est 0 alors $o2_0 = n_0 + 1$ sinon $o2_0 = 1$. Nous savons que rst_0 est 0 (valeur donnée par la trace d'entrée) donc $o2_0 = n_0 + 1$, ce qui conduit à $o2_0 = 1$ car nous savons que $n_0 = 0$.
- La sémantique spécifie que $o1_0 = n_0$, donc $o1_0 = 0$.
- Et finalement, la sémantique spécifie que $n_1 = o2_0$, donc $n_1 = 1$

Maintenant, nous connaissons la valeur de n_1, on peut donc recommencer le raisonnement précédent pour calculer les valeurs de $o2_1$, $o1_1$ et n_2. En propageant ainsi les contraintes exprimées par la sémantique, on peut calculer toutes les traces des sorties, étant donné la trace des entrées.

10.3 Quelques opérateurs sur les traces

Nous définissons ici quelques opérateurs qui nous seront utiles pour manipuler les traces. Ils seront principalement utilisés lorsque nous considérerons la sémantique relationnelle qui travaille directement sur les traces elles-mêmes. Le premier opérateur (d^ω, d pour domaine) nous permet de connaître le type (le domaine) d'une trace d'un ensemble de variables (dans un contexte où chacune des variables est typée). Le deuxième opérateur (*select*) extrait une trace d'un ensemble de variables à partir d'une trace d'un ensemble de variables plus grand. L'opérateur *merge* mélange deux traces en utilisant une troisième trace sur un booléen pour savoir, à chaque cycle, quelle trace est choisie : à un cycle t, si le booléen de la troisième trace est positif, c'est la valeur de la première trace qui est choisie, sinon c'est la valeur de la deuxième trace. Nous définissons également un opérateur de décalage sur les traces qui permet de retirer les premiers éléments d'une trace. Nous finissons en définissant un opérateur permettant de réduire la taille d'une trace. Il s'agit d'un opérateur complémentaire à l'opérateur de décalage (qui réduit également la longueur de la trace), il supprime un certain nombre d'éléments à la fin de la trace.

Domaine des traces d'un ensemble de variables

On fait l'hypothèse d'être dans un contexte dans lequel toute variable est typée, et le type d'une variable x est noté T_x. Dans le cas d'un modèle BHDL, le type des variables se trouve dans la clause INVARIANT du modèle. Une trace x^ω de la variable x est donc dans l'ensemble T_x^ω. Si nous considérons maintenant un vecteur de variables $(x_1, ..., x_n)$, une trace de ce vecteur est dans l'ensemble $(T_{x_1} \times ... \times T_{x_n})^\omega$. En fait nous ne considérerons par ce genre de trace sur des vecteurs, mais nous considérerons les vecteurs de traces qui sont dans l'ensemble $T_{x_1}^\omega \times ... \times T_{x_n}^\omega$. Pour éviter des formules trop complexes, nous utilisons l'opérateur d^ω qui associe à tout ensemble de variables le type des traces sur ces variables.

$$d^\omega(\{x_1, ..., x_n\}) = T_{x_1}^\omega \times ... \times T_{x_n}^\omega$$

Remarquez que nous ne faisons pas de différence entre le vecteur (ou la liste) $(x_1, ..., x_n)$ et l'ensemble $\{x_1, ..., x_n\}$. Nous considérons que tout ensemble de variables est ordonné, et que cet ordre est toujours le même (par exemple l'ordre alphabétique).

Nous définissons également l'opérateur d qui donne le type d'un vecteur de variables.

$$d(\{x_1, ..., x_n\}) = T_{x_1} \times ... \times T_{x_n}$$

Réduction du domaine d'une trace

On considère une trace donnée v^ω d'un ensemble de variables v. On veut extraire de v^ω une trace u^ω d'un sous ensemble u de v.

Si $v = \{v_1, ..., v_n\}$ et $v^\omega = (v_1^\omega, ..., v_n^\omega)$, la trace correspondant au sous ensemble de variables $u = \{v_{i_1}, ..., v_{i_p}\}$ est $u^\omega = (v_{i_1}^\omega, ..., v_{i_p}^\omega)$. Nous définissons l'opérateur *select* pour réaliser cette transformation. Dans la définition suivante, les indices i_1 à i_p sont une sélection des indices de 1 à n, l'ensemble de variables $\{v_{i_1}, ..., v_{i_p}\}$ est donc un sous ensemble de $\{v_1, ..., v_n\}$.

$$select_{\{v_1,...,v_n\} \to \{v_{i_1},...,v_{i_p}\}}((v_1^\omega, ..., v_n^\omega)) = (v_{i_1}^\omega, ..., v_{i_p}^\omega)$$

Par exemple, prenons les ensembles de variables $v = \{a, b, c\}$ et $u = \{b\}$ et une trace v^ω sur v : $v^\omega = (\langle 1, 4, 7, 10\rangle, \langle 2, 5, 8, 11\rangle, \langle 3, 6, 9, 12\rangle)$. Nous avons $select_{v \to u}(v^\omega) = \langle 2, 5, 8, 11\rangle$.

Mélange de deux traces par rapport à une troisième

Trois traces u, v et c de la même longueur (éventuellement non finies) sont données. La trace c est de type $Boolean^\omega$. On veut construire une nouvelle trace t qui est le mélange de u et v en suivant ce principe : pour toute position i dans les trois traces, si c_i est positif alors $t_i = u_i$ sinon $t_i = v_i$.

$$merge_c(u, v) = t \text{ où } \forall i \cdot (c_i \Rightarrow t_i = u_i \wedge \neg c_i \Rightarrow t_i = v_i)$$

Pour créer la trace qui permet de conduire le mélange à partir d'un prédicat, on utilise la notation x^ω / P où x^ω est une trace sur l'ensemble x des variables libres dans le prédicat P.

$$x^\omega / P = \langle p_1, ...\rangle \text{ où } \forall i \cdot (p_i = P(x_i))$$

Décalage d'une trace

Pour une trace $x^\omega = \langle x_0, x_1, ...\rangle$, la trace décalée une fois à gauche est notée par $x_{+1}^\omega = \langle x_1, x_2, ...\rangle$. Plus généralement, on peut définir $x_{+n}^\omega = \langle x_n, x_{1+n}, ...\rangle$. Pour éviter l'introduction de valeurs non définies, nous ne définissons pas de décalage à droite.

Le décalage à gauche peut réduire la longueur de la trace. Dans le cas de traces finies, un décalage à gauche de n éléments produit une trace dont la longueur est réduite de n. Dans le cas d'une trace infinie, la longueur de la trace résultant du décalage est infinie aussi.

$$length(x_{+n}^\omega) = length(x^\omega) - n$$

Réduction de la longueur d'une trace

Pour une trace x^k de longueur k, on note $x^{k\to l}$ la trace de longueur l qui est égale à x^k pour les l premiers éléments. Ceci est bien défini seulement dans le cas où $l \leq k$; nous considérons que $l \leq \infty$ pour tout entier l. Par exemple si on prend la trace de longueur 4 $i^4 = \langle 1, 3, 7, 9\rangle$ on peut la réduire à une trace de longueur 2 et on a $i^{4\to 2} = \langle 1, 3\rangle$.

Dans le cas où k est en fait ω, c'est-à-dire que la longueur de x^ω n'est pas spécifiée explicitement, $x^{\omega\to l}$ est considéré bien formé lorsqu'il est possible de s'assurer à partir du contexte que l n'est pas plus grand que la longueur de x^ω.

Par exemple, on verra dans la définition de la sémantique de traces relationnelle que nous considérons deux traces x^ω et y^ω de la même longueur. La trace x^ω est une trace d'un ensemble de variables R et y^ω est une trace d'un ensemble de variables W. Considérons l'expression suivante :

$$select_{R\to R\cap W}(x^\omega_{+1}) = select_{W\to R\cap W}(y^{\omega\to length(x^\omega_{+1})})$$

Cette égalité teste si la trace x^ω décalée une fois à gauche et la trace y^ω, prises sur une ensemble commun de variables $R \cap W$, sont égales. Il est nécessaire de réduire la taille de y^ω (dans le cas où elle n'est pas infinie) car la longueur de x^ω_{+1} est $length(x^\omega) - 1$, c'est-à-dire qu'il y a un élément de plus dans y^ω (qui a la même longueur que x^ω). Dans le cas où les deux traces sont infinies, la trace $y^{\omega\to length(x^\omega_{+1})}$ est égale à y^ω car $\infty - 1 = \infty$, donc écrire $y^{\omega\to length(x^\omega_{+1})}$ revient à écrire $y^{\infty\to\infty}$.

Chapitre 11

Sémantique prédicative

Ce chapitre définit une sémantique de traces pour BHDL en utilisant les prédicats *avant-après* (cf. section 11.2). La sémantique consiste à produire, à partir d'un modèle BHDL, un *prédicat* définissant les traces des variables de sortie en fonction des variables d'entrée. En fait, le prédicat défini par la sémantique définit un ensemble de fonctions (une seule si le modèle est déterministe), qui font cette association.

Nous définissons cette sémantique avec pour objectif de l'utiliser pour prouver le correction de la traduction d'un modèle BHDL vers un langage cible. Pour cela nous définirons une sémantique similaire pour le langage cible et nous montrerons que la sémantique du modèle initial et la sémantique de la traduction sont identiques..

Pour éviter des formules trop compliquées nous définissons d'abord un certains nombre d'opérateurs qui nous seront utiles : des opérateurs permettant de récupérer le nom du modèle ou le contenu d'une autre clause, des opérateurs permettant d'obtenir la liste des modèles qui sont importés directement ou indirectement (en donnant à chacun d'eux un nom unique), et des opérateurs collectant toutes les constantes définies dans les machines déclaratives vues (directement ou indirectement).

Nous utilisons la notation $[\![\cdot]\!]$ pour cette sémantique. Elle est définie en deux étapes. La sémantique est d'abord exprimée comme un prédicat *avant-après*, noté $(|\cdot|)$, de la substitution modélisant le circuit. Ce prédicat est défini comme en B, en ajoutant des définitions pour les deux substitutions qui ont été ajoutées au langage BHDL : substitution de réutilisation de composants existants et substitution de bloc sous-modèle. Le prédicat *avant-après* exprime la relation entre les valeurs des variables en début de cycle (avant l'application de la substitution), et les valeurs des variables à la fin du cycle (après application de la substitution). Pour une substitution d'importation, le principe consiste à réécrire le prédicat *avant-après* du composant importé et à ajouter les initialisations des variables importées aux initialisations des variables locales. Une fois le prédicat *avant-après* défini, celui-ci est élevé au rang de prédicat sur des traces en remplaçant les variables v par des variables représentant les traces v^ω des variables originales.

11.1 Quelques opérateurs utiles

Dans cette section, nous définissons quelques opérateurs qui nous seront utiles dans la définition de la sémantique. Les modèles BHDL sont supposés être dans un espace commun et identifiés de manière unique par leurs noms. Certains opérateurs sont définis pour extraire de l'information d'un modèle à partir de son nom. Nous définissons aussi des opérateurs qui fournissent la liste des autres modèles BHDL qui sont importés et des machines déclaratives qui sont vues.

11.1.1 Espace des noms

Tous les modèles BHDL sont dans un espace commun de modèles de composants CS et les noms des composants sont dans l'espace de nom CNS. L'opérateur $Comp$ associe à chaque nom le modèle correspondant.

$$Comp \in CNS \rightarrowtail CS$$

Nous définissons les opérateurs suivants pour extraire des informations d'un modèle de composant à partir de son nom :

- $Init$: donne la substitution de la clause INITIALISATION,
- Op : donne la substitution de la clause OPERATIONS,
- $Sees$: donne la liste des machines déclaratives vues spécifiées dans la clause SEES,
- F : l'ensemble des variables qui peuvent être écrites (c'est-à-dire les variables locales et les sorties),
- Var : opérateur donnant la liste des variables définies par la clause VARIABLES.

$$\text{Si } Comp(D) = \begin{cases} \text{MACHINE } D \\ \text{IMPORTS } imp \\ \text{SEES } param \\ \text{INPUTS } i \\ \text{OUTPUTS } o \\ \text{VARIABLES } v \\ \text{INVARIANT } inv \\ \text{INITIALISATION } init \\ \text{OPERATIONS } C \\ \text{END} \end{cases} \text{ alors } \begin{cases} Init(D) = init \\ Op(D) = C \\ Sees(D) = param \\ F(D) = o \cup v \\ Q(D) = imp \\ Var(D) = v \end{cases}$$

De la même façon, nous définissons, pour les machines déclaratives, les opérateurs $Sees$, $Const$ et $Prop$ qui donnent respectivement le contenu des clauses SEES, CONSTANTS et PROPERTIES d'une machine déclarative à partir de son nom.

11.1.2 Modèles importés

Un modèle BHDL peut importer d'autres modèles qui peuvent eux-mêmes importer encore d'autres modèles. Notons que, pour être acceptable, le graphe de dépendance doit ne contenir aucun cycle. Dans ce graphe, nous appelons *chemins d'importation* les chemins partant du nœud principal, correspondant au modèle principal, jusqu'aux nœuds représentant les modèles importés. Nous définissons l'opérateur *imported* qui donne l'ensemble des modèles importés, récursivement, en donnant un nom unique à chacun. Le nom d'un modèle C importé correspond au chemin d'importation dans le graphe de dépendance partant du modèle initial D jusqu'à C. Si le chemin est $D \rightarrow Imp_1 \rightarrow Imp_2 \rightarrow C$, le nom de C dans D sera $Imp_1.Imp_2.C$. L'opérateur *imported* est défini ci-dessous. Nous utilisons l'opérateur *limport* qui fournit, à partir de la clause IMPORTS, la liste des modèles importés sous forme de couples, le premier élément d'un couple est le nom de l'instance et le deuxième élément est le modèle lui-même.

$$imported(\text{MACHINE } d \text{ IMPORTS } imp \ ...) = \begin{cases} \text{let } li = limport(imp) \text{ in} \\ li \cup \bigcup_{(ic,D) \in li} \bigcup_{(n,F) \in imported(D)} \{(ic.n, F)\} \end{cases}$$

$$limport(\varnothing) = \varnothing$$

$$limport(ic : D, rest) = \{(ic, Comp(D))\} \cup limport(rest)$$

Lorsqu'un modèle est importé, il possède ses propres variables locales. La sémantique prédicative que nous définissons ici importe toutes les variables locales des modèles importés. Il est nécessaire de renommer toutes ces variables pour assurer l'unicité des noms. Le problème peut se poser par exemple lorsque deux modèles importés utilise un même nom de variable, ou encore lorsque qu'on importe plusieurs fois le même modèle (sous des noms d'instances différents). Ces variables sont renommées selon le chemin d'importation du modèle.

$$RenamedImportedVar(D) = \bigcup_{(n,C) \in imported(D)} \bigcup_{v \in Var(C)} \{n.v\}$$

11.1.3 Clause SEES

Nous aurons également besoin de collecter les définitions de constantes définies dans les machines déclaratives vues qui sont listées dans la clause SEES. Les machines vues définissent un environnement

de constantes pour le modèle BHDL. Les dépendances entre les machines déclaratives définissent un graphe de 'vision' qui doit ne contenir aucun cycle. L'environnement du modèle BHDL est constitué de l'ensemble des constantes qui sont définies dans les machines vues directement et des constantes définies dans les autres machines du graphe de vision.

L'opérateur *getallnames* fournit l'ensemble des constantes qui sont définies par les machines du graphe de vision et *getallspecs* donne les définitions de ces constantes. L'opérateur *getallpackages* est utilisé pour collecter toutes les machines déclaratives du graphe de vision. Les constantes n'ont pas besoin d'être renommées car nous avons imposé qu'un modèle BHDL ne peut pas voir, directement ou indirectement, deux machines qui définissent des constantes ayant des noms identiques. Les définitions de ces trois opérateurs sont données ci-dessous.

$$getallpackages(D) = Sees(D) \cup \bigcup_{d \in Sees(D)} getallpackages(d)$$
$$getallnames(D) = \bigcup_{d \in getallpackages(D)} Const(D)$$
$$getallspecs(D) = \bigwedge_{d \in getallpackages(D)} Prop(D)$$

11.1.4 Variables cachées par les blocs

Certaines variables sont cachées par l'utilisation des blocs sous-modèles. La sémantique prédicative des modèles contenant des blocs est définie en aplatissant le modèle : les variables des blocs sont considérées comme des variables globales et leur initialisation est faite avec les autres variables. Pour récupérer l'ensemble des variables cachées qui doivent être globalisées nous utilisons l'opérateur *HiddenVar*. Il est défini par induction sur la structure des substitutions. Nous définissons également l'opérateur *HiddenInit* qui donne la substitution permettant d'initialiser toutes ces variables.

substitution S	$HiddenVar(S)$	$HiddenInit(S)$
$x := E$	$\varnothing$	$skip$
$A; B$	$HiddenVar(A) \cup HiddenVar(B)$	$HiddenInit(A) \| HiddenInit(B)$
$A \| B$	$HiddenVar(A) \cup HiddenVar(B)$	$HiddenInit(A) \| HiddenInit(B)$
IF P THEN A ELSE B END	$HiddenVar(A) \cup HiddenVar(B)$	$HiddenInit(A) \| HiddenInit(B)$
$(O) \leftarrow ic(I)$	$\varnothing$	$skip$
$\mathcal{B}_v(init, S)$	v	$init$

11.2 Prédicats *avant-après* des substitutions généralisées

Le prédicat *avant-après* d'une substitution est un prédicat qui exprime la relation qui existe entre les valeurs des variables *avant* l'application de la substitution et les valeurs des variables *après* l'application de la substitution. Pour un bloc sous-modèle ou un composant importé, il n'est en fait pas possible d'écrire un prédicat relient les valeurs *avant* et les valeurs *après* des variables car cette relation n'est pas fixe dans le temps. Ceci est dû au fait qu'un bloc ou qu'un modèle importé peut cacher des mémoires. La sémantique prédicative que nous sommes en train de définir résout ce problème en aplatissant totalement le modèle, la description des modèles importés est entièrement importée et toutes les variables cachées sont globalisées. Les prédicats *avant-après* que nous définissons pour l'importation de modèles et les blocs réalise cet aplatissement en intégrant au prédicat global, les prédicats des sous-modèles.

Le prédicat *avant-après* d'une substitution est défini par induction sur la structure de la substitution. La définition du prédicat dépend de deux paramètres :

- L'ensemble F est l'ensemble des variables qui peuvent être écrites par le modèle. C'est un ensemble support en écriture potentiel, il doit au moins contenir l'ensemble support en écriture. Par simplification, nous utiliserons l'ensemble des variables du modèle et des sorties. Si on souhaite 'optimiser' le prédicat il faudra calculer au plus juste les ensembles supports en écriture des substitutions.

- L'ensemble Q est l'ensemble des modèles qui sont importés. Cet ensemble est obtenu en utilisant l'opérateur *imported* défini précédemment.

Nous utilisons la notation $(|S|)_{F;Q}$ pour le prédicat *avant-après* d'une substitution S avec les paramètres F et Q. Dans ce prédicat, la valeur *après* d'une variable v est noté v', la valeur *avant* est simplement notée v. Le paramètre Q contenant l'ensemble des modèles importés, il n'est utilisé que dans la substitution d'importation.

Substitutions simples

La définition du prédicat *avant-après* d'une substitution simple ne pose pas de difficulté. La valeur *après* de la variable modifiée est la valeur à laquelle l'expression se situant dans la partie droite de la substitution est évaluée. Nous ne devons pas oublier de spécifier que les valeurs des autres variables ne changent pas ($\bigwedge_{y \in F-\{s\}} y' = y$) : si l'ensemble F était calculé au plus juste, cette deuxième partie ne serait pas nécessaire car F contiendrait seulement la variable qui est modifiée.

- $(|s := E|)_{F;Q} \equiv s' = E \wedge \bigwedge_{y \in F-\{s\}} y' = y$
- La substitution multiple $x, y, \ldots := E, F, \ldots$ est considérée comme une abréviation pour $x := E \| y := F \| \ldots$ En fait, la définition du langage B considère plutôt que c'est le contraire (la composition parallèle est définie à partir de la substitution multiple), mais il nous paraît plus simple de définir notre sémantique de cette façon.
- $(|s :\in E|)_{F;Q} \equiv s' \in E \wedge \bigwedge_{y \in F-\{s\}} y' = y$
- Le cas où l'expression est une λ-expression (représentant un tableau) suit la même règle que le cas général. Nous développons cependant la règle dans ce cas pour montrer comment le prédicat obtenu peut se simplifier.

$$\left(\left|s := \bigcup_i \lambda k \cdot (k \in F_i | E_i) \cup \bigcup_j \lambda k \cdot (k = D_j | G_j)\right|\right)_{F;Q}$$
$$\equiv s' = \bigcup_i \lambda k \cdot (k \in F_i | E_i) \cup \bigcup_j \lambda k \cdot (k = D_j | G_j) \ \wedge \ \bigwedge_{y \in F-\{s\}} y' = y$$
$$\equiv \bigwedge_i (\forall k \in F_i \cdot s'(k) = E_i) \ \wedge \ \bigwedge_j (s'(D_j) = G_j) \ \wedge \ \bigwedge_{y \in F-\{s\}} y' = y$$

Puisque l'on doit avoir $type(s, inv) \supseteq type(\bigcup_i \lambda k \cdot (k \in F_i | E_i) \cup \bigcup_j \lambda k \cdot (k = D_j | G_j), inv)$ (condition de bonne formation des substitutions simples), on déduit qu'on doit avoir $\bigcup_i F_i \cup \bigcup_j \{D_j\} = dom(type(s, inv))$.

Composition séquentielle

Le prédicat *avant-après* d'une composition séquentielle $A; B$ consiste à dire que la valeur *avant* utilisée par la substitution B est la valeur *après* de A. À partir de la valeur *avant* F des variables, l'application de A produit les valeurs après F^* qui sont les valeurs *avant* de B. La substitution B, à partir de F^*, produit les valeurs *après* F'. Ceci est résumé sur le graphique ci-dessous.

$$F \xrightarrow{A} F^* \xrightarrow{B} F'$$

La notation F^d représente l'ensemble des variables de F auxquelles on ajoute la décoration d. C'est-à-dire que F' est l'ensemble des valeurs *après* des variables de F, F représentant l'ensemble des valeurs *avant* (puisqu'on utilise la même notation pour les noms des variables et les valeurs *avant*). L'ensemble F^* est l'ensemble des variables décorées par une étoile, cet ensemble représente l'ensemble des valeurs intermédiaires. Dans le prédicat, les valeurs intermédiaires sont introduites par un quantificateur existentiel, exprimant que pour qu'une valeur *après* v' résulte de l'application de $A; B$ à partir de la valeur *avant* v, il faut qu'il existe une valeur v^* qui résulte de l'application de A à partir de la valeur *avant* v et qui soit telle que la valeur *après* v' soir obtenue par l'application de B à partir de la valeur v^*.

$$(|A; B|)_{F;Q} \equiv \exists F^* \ [F' := F^*] \, (|A|)_{F;Q} \ \wedge \ [F := F^*] \, (|B|)_{F;Q}$$

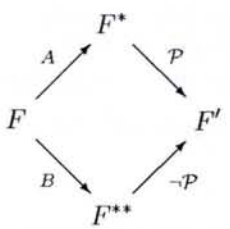
Composition parallèle

Le prédicat *avant-après* d'une composition parallèle $A \| B$ est la conjonction des prédicats *avant-après* de chacune des deux substitutions composées en parallèle. Cependant, le calcul des prédicats de chacune des deux substitution est fait, pour chacune, sur leur propre ensemble support en écriture. Ceci permet d'éviter la duplication de termes du style $y' = y$ pour les variables qui ne sont dans l'ensemble support ni de A ni de B. Ces termes sont ajoutés par le terme $\bigwedge_{y \in F-(write(A) \cup write(B))} y' = y$. Rappelons que les substitutions A et B ont nécessairement des ensembles supports en écriture disjoints, c'est une condition de bonne formation du langage B.

$$(|A \| B|)_{F;Q} \equiv (|A|)_{write(A);Q} \wedge (|B|)_{write(B);Q} \wedge \bigwedge_{y \in F-(write(A) \cup write(B))} y' = y$$

Composition conditionnelle

Le prédicat *avant-après* d'une composition conditionnelle est constitué des deux prédicats des substitutions qui sont composées (disons A pour la première, et B pour la deuxième) dans lesquels nous remplaçons les valeurs *après* F' par des valeurs intermédiaires : F^* pour la première substitution, et F^{**} pour la deuxième. Le choix parmi les deux possibilités est fait selon la valeur à laquelle est évaluée la condition $\mathcal{P}$. Si $\mathcal{P}$ est évaluée à *TRUE* alors $F' = F^*$, sinon $F' = F^{**}$. A partir des valeurs *avant* F, l'application de A produit les valeurs F^* et l'application de B produit les valeurs F^{**}. Ainsi, si $\mathcal{P}$ est évaluée à *TRUE*, la valeur *après* est F^*, sinon la valeur après est F^{**}. Ceci est résumé sur le graphique ci-dessous.



Le prédicat lui-même s'exprime par la conjonction de quatre prédicats. D'une part les deux prédicats *avant-après* des substitutions qui sont composées dans lesquels les valeurs *après* ont été remplacées par F^* ou F^{**} selon le cas. D'autre part deux prédicat qui expriment le choix qui est fait suivant l'évaluation de $\mathcal{P}$.

$$(|\text{IF } \mathcal{P} \text{ THEN } A \text{ ELSE } B \text{ END}|)_{F;Q} \equiv \exists F^*, F^{**} \begin{cases} \mathcal{P} \Rightarrow F' = F^* \wedge \\ \neg\mathcal{P} \Rightarrow F' = F^{**} \wedge \\ [F' := F^*]\,(|A|)_{F;Q} \wedge \\ [F' := F^{**}]\,(|B|)_{F;Q} \end{cases}$$

Notons que ce prédicat (équivalent à $\mathcal{P} \Rightarrow (|A|)_{F;Q} \wedge \neg\mathcal{P} \Rightarrow (|B|)_{F;Q}$) n'est pas exactement celui qui est généralement donné pour définir le prédicat *avant-après* d'une composition conditionnelle en B, mais il lui est équivalent. Le prédicat qui est classiquement donné est le suivant : $\mathcal{P} \wedge (|A|)_{F;Q} \vee \neg\mathcal{P} \wedge (|B|)_{F;Q}$.

Nous avons choisi cette formulation car elle correspond mieux à la sémantique d'un circuit. Dans un circuit, les deux branches d'une composition conditionnelle sont toujours 'calculées', elles sont physiquement présentes sur le circuit et le choix est finalement fait par un multiplexeur qui choisit entre les deux sorties de chaque branche, en fonction de l'évaluation de $\mathcal{P}$.

Importations de modèles

Nous utilisons abusivement le terme de prédicat *avant-après* pour l'importation de modèle. En fait, nous copions le prédicat *avant-après* de la clause OPERATIONS du modèle importé en remplaçant les

entrées et les sorties par les variables locales auxquelles les ports sont connectés, comme cela est spécifié dans les listes de connexions O et I de la substitution $(O) \leftarrow ic(I)$.

Nous utilisons le terme de prédicat *avant-après* par homogénéité alors qu'il n'est en fait pas possible, en général, de spécifier la sémantique d'une importation par un simple prédicat *avant-après* car le modèle importé peut contenir des variables modélisant de la mémoire (des registres). Pour pouvoir exprimer la sémantique d'une importation il faut considérer les traces des variables (cf. la sémantique relationnelle définie dans le chapitre suivant), ou éventuellement une logique temporelle.

Ici, nous définissons la sémantique par aplatissement du modèle BHDL : les variables des modèles importés sont importées et deviennent des variables du modèle important (avec renommage pour éviter les conflits de nom). Les initialisations de ces variables sont également importées et jointes aux initialisations des variables du modèle important. La description du modèle importé (le prédicat *avant-après* de sa clause OPERATIONS) est recopié (en remplaçant les entrées/sorties par les variables qui y sont connectées). C'est pourquoi les variables du modèle importé sont libres dans le prédicat ci-dessous, elles sont déclarées avec les autres variables dans le prédicat qui définit la sémantique du modèle complet (cf. section 11.3.2).

$$(|(O) \leftarrow ic(I)|)_{F;Q} \equiv_{(ic:Comp)\in Q} \begin{cases} \text{let} \\ \quad vc = Var(Comp) \\ \quad iv = RenamedImportedVar(Comp) \\ \quad av = vc \cup iv \\ \quad (po, lo) = PortLoc(O) \\ \quad (pi, li) = PortLoc(I) \\ \quad FC = F(Comp) \\ \quad QC = Q(Comp) \\ \text{in} \\ \begin{bmatrix} po, po', & lo, lo', \\ pi, pi', & := li, li', \\ av, av' & ic.av, ic.av' \end{bmatrix} (|Op(Comp)|)_{FC;QC} \\ \wedge \bigwedge_{y\in F-lo} y' = y \end{cases}$$

Blocs sous-modèles

Le cas d'un bloc suit le même principe que celui d'une importation mais est plus simple car il n'y a pas de variables à renommer. Les variables qui sont cachées par le bloc sont considérées comme globales (on suppose que deux blocs n'utilisent pas des variables du même nom) et la clause OPERATIONS est traitée comme si elle n'était pas à l'intérieur d'un bloc.

$$(|\text{BLOCK } v \text{ TYPES } inv \text{ INITIALISATION } init \text{ OPERATIONS } S \text{ END}|)_{F;Q} \equiv (|S|)_{F;Q}$$

Décorations

Dans l'expression des prédicats *avant-après* ci-dessus nous avons utilisé les notations F^* and F^{**} pour identifier des nouveaux noms de variables. Cependant, l'imbrication des quantifications existentielles $\exists$ sur ces variables ne suffit pas à prévenir des conflits de nom entre ces nouvelles variables. Nous avons besoin d'assurer l'unicité des noms, pour cela, nous pouvons utiliser des compteurs à la place des étoiles.

Par exemple, nous montrons comment il est possible de redéfinir le prédicat *avant-après* de la composition conditionnelle en utilisant des compteurs. La notation utilisant $\uparrow$ et $\downarrow$ peut être vue comme une fonction dans un langage de programmation ou comme un prédicat Prolog : F, Q et n sont les paramètres d'entrée et p est un paramètre de sortie. Lorsque nous voulons calculer le prédicat *avant-après* de la substitution, nous devons fournir F, Q et n, et nous récupérons p en résultat (en plus du prédicat lui-même qui est considéré comme un paramètre de sortie implicite).

$$(|\text{IF } \mathcal{P} \text{ THEN } A \text{ ELSE } B \text{ END}|) \downarrow F \downarrow Q \downarrow n \uparrow p$$
$$\equiv \exists F^n, F^{n+1} \begin{cases} \mathcal{P} \Rightarrow F' = F^n \wedge \\ \neg\mathcal{P} \Rightarrow F' = F^{n+1} \wedge \\ [F' := F^n]\,(|A|) \downarrow F \downarrow Q \downarrow n+2 \uparrow m \wedge \\ [F' := F^{n+1}]\,(|B|) \downarrow F \downarrow Q \downarrow m \uparrow p \end{cases}$$

11.3 Sémantique de traces d'un modèle BHDL

La sémantique de traces d'un modèle BHDL est définie en utilisant les prédicats *avant-après* définis précédemment. Cette sémantique de traces définit par un prédicat la relation existant entre les traces des sorties et les traces des entrées. Le principe général consiste à élever les prédicats *avant-après* au rang de prédicats sur des traces. Dans la suite de cette section, nous expliquons d'abord comment élever au rang de variables de traces les variables d'un modèle BHDL et nous verrons à quoi correspondent ces variables de traces. Ensuite, nous définissons formellement la sémantique d'un modèle BHDL avant d'illustrer le calcul de cette sémantique sur l'exemple du compteur 3-bits.

11.3.1 Élévation des variables au rang de variables de traces

La sémantique de traces d'un modèle est définie en utilisant les prédicats *avant-après* des substitutions qui définissent le modèle (clauses INITIALISATION et OPERATIONS du modèle, des modèles importés — directement ou indirectement — et des blocs sous-modèles). Comme définis précédemment, les prédicats *avant-après* sont des prédicats utilisant des variables sans décoration x pour désigner les valeurs *avant* des variables du modèle, et des variables décorées x' pour désigner les valeurs *après*. Pour définir la sémantique de traces, ces variables sont élevées au rang de variables de traces. Une variable de traces exprimant la trace de la variable correspondante le long de la succession des cycles d'horloge.

Le principe général de la sémantique de traces d'un modèle BHDL, permettant de définir les valeurs des traces est le suivant :

- la substitution de la clause INITIALISATION (en fait on ajoute la réunion de toutes les clauses INITIALISATION des modèles importés et des blocs sous-modèles) est appliquée une seule fois,
- une fois que la substitution de la clause INITIALISATION a été appliquée, on applique de façon répétitive la substitution de la clause OPERATIONS.

Nous illustrons la séquence d'application des substitutions sur la figure ci-dessous. Nous notons par S^i la substitution de la clause INITIALISATION et par S^{op} la substitution de la clause OPERATIONS. Au fur et à mesure de l'application des substitutions, les valeurs portées par les variables changent. Pour chacune des applications d'une substitution, les modifications peuvent être spécifiées par le prédicat *avant-après* de la substitution appliquée. Afin de différencier les applications des substitutions les unes des autres, nous ajoutons un indice aux substitutions et aux noms de variables. Cet indice correspond à l'index de l'application de la substitution dans la séquence d'applications : S_k est la substitution qui appliquée à l'index numéro k de la séquence de substitution. La numérotation commence à 0. L'initialisation, appliquée une seule fois en début de séquence, correspond à l'index 0, elle sera notée S_0^i. Les index des substitutions S^{op} commencent à 1. Les valeurs *avant-après* sont indexées de cette façon : avant l'application de S_k la valeur de la variable x est x_{k-1} et après l'application de S_k la valeur de x est x'_{k-1}. La valeur *avant* de S_{k+1} est la valeur après de S_k : nous avons la propriété $\forall k \cdot (k \geq 0 \Rightarrow x_{k+1} = x'_k)$. Ainsi, l'application de S_k produit la valeur x_k. La notation x'_i correspond à la valeur de x après l'application de l'initialisation S_0^i, et elle est égale à x_0. La valeur x'_i ne fait pas partie de $(x')^\omega$, cf. la remarque sur l'initialisation ci-dessous.

$$\underset{}{S_0^i} \quad \underset{x'_i = x_0}{\longrightarrow} \quad \underset{}{S_1^{op}} \quad \underset{x'_0 = x_1}{\longrightarrow} \quad \underset{}{S_2^{op}} \quad \underset{x'_1 = \ldots}{\longrightarrow} \quad \ldots$$

Deux variables de traces, $x^\omega = (x_0, x_1, \ldots)$ et $(x')^\omega = (x'_0, x'_1, \ldots)$, sont associée à chaque variable x. La variable de traces x^ω représente la trace des valeurs *avant* de x et $(x')^\omega$ la trace des valeurs *après* de x.

Initialisation On notera que chacune des deux traces commence *après* l'initialisation. En effet, l'initialisation correspond au *reset* d'une machine de Mealy, qui initialise les registres. Une fois les registres initialisés, les nouvelles valeurs de ces registres se propagent le long des fils du circuit et arrivent finalement jusqu'aux sorties. Les modifications correspondant à cette propagation dans le circuit est ce qui est modélisé par la clause OPERATIONS d'un modèle BHDL. Ainsi, l'*effet* de l'initialisation correspond à l'application de la clause INITIALISATION suivie d'une application de la

clause OPERATIONS. Concernant la trace des valeurs *avant*, deux choix étaient a priori possibles : soit les valeurs initiales sont les valeurs *avant* l'initialisation, soit les valeurs entre l'initialisation et la première application de la clause OPERATIONS. Nous avons choisi la deuxième solution car dans ce cas les valeurs sont bien spécifiées par le modèle (c'est le résultat de l'application de la clause INITIALISATION). Ceci correspond également à la façon dont nous modélisons les circuits en B événementiel : nous observons les modifications apportées par les événements, l'état de départ est celui spécifié par la clause INITIALISATION. On peut résumer l'effet de l'initialisation par la formule ci-dessous, où le signe + signifie "suivi de" et "1∗" signifie "appliquer 1 fois".

$$\text{Effet de l'initialisation} = \text{INITIALISATION} + 1 * \text{OPERATIONS}$$

C'est pourquoi la valeur *après* de l'initialisation, x'_i, ne fait pas partie de la trace des valeurs *après*. Elle correspond à un état instable du circuit, se situant entre l'initialisation des registres et le début de la propagation des nouvelles valeurs dans le circuit.

Une différence avec les affectations concurrentes de certains autres langages de description de circuits est que la composition séquentielle introduit des valeurs intermédiaires pour les variables. Dans la sémantique que nous définissons ici, les variables intermédiaires ne sont pas élevées au rang de traces ; elles sont considérées comme des variables locales quantifiées existentiellement (sous la quantification des traces). Dans certains cas, pour la simulation par exemple, il pourrait être utile d'avoir accès à ces valeurs intermédiaires. Dans ce cas, les variables intermédiaires pourraient également être élevées au rang de variables de traces.

11.3.2 Sémantique de trace

La sémantique de traces est un prédicat définissant une fonction f qui associe les traces des sorties et des variables aux traces des entrées du circuit. La sémantique (cf. formule marquée ⊛ ci-dessous) est composée de six parties : définitions des constantes, initialisation de l'ensemble des variables (parties 2 à 4), le prédicat *avant-après* de la clause OPERATIONS du modèle et un prédicat reliant les valeurs *après* d'un cycle aux valeurs *avant* du cycle suivant.

On remarquera que dans le cas des initialisations, on utilise les prédicats *avant-après* des clauses INITIALISATION, et on remplace les valeur *après*, de type v', par les valeurs des éléments initiaux (éléments d'indice 0) des traces, de type v_0.

Dans le prédicat *avant-après* de la clause OPERATIONS, les valeurs *avant* (par exemple x) sont remplacées par des variables indicées k (par exemple x_k) et les valeurs *après* (par exemple x') par des variables indicées également par k (par exemple x'_k). On ajoute le prédicat $\forall k \cdot (k \geq 0 \Rightarrow av_{k+1} = av'_k)$ pour lier les valeurs *avant* d'un cycle aux valeurs *après* du cycle précédent. Ainsi le prédicat *avant-après* est élevé au rang de prédicat sur des traces, il décrit le lien entre les valeurs des variables au cycle $k+1$ en fonction des valeurs des variables au cycle k. Les six parties définissant la sémantique dont les suivantes :

1. Les définitions des constantes définies dans les machines déclaratives vues.
2. L'initialisation des variables du modèle. Cela correspond au prédicat *avant-après* de la clause INITIALISATION.
3. L'initialisation des variables importées. Les variables représentant des entrées/sorties sont ignorées ($Init(C) \ominus p$) car elles ne font pas partie effectivement du modèle : ce sont des noms formels permettant d'identifier les ports des circuits, elles sont en fait liées aux variables du modèle importateur. Nous utilisons la notation $Init(C) \ominus p$ pour représenter la substitution $Init(C)$ dans laquelle on a enlevé toutes les substitutions concernant les variables qui sont dans l'ensemble p.
4. L'initialisation des variables des blocs sous-modèles.
5. Le cœur du modèle, le prédicat *avant-après* de la clause OPERATIONS. Rappelons que la définition du prédicat *avant-après* donnée précédemment réutilise elle-même les prédicats *avant-après* des modèles importés et des blocs sous-modèles. Le prédicat *avant-après* est élevé au rang de prédicat sur les traces.

6. Un prédicat reliant toutes les valeurs *après* aux valeurs *avant* du cycle suivant : pour toute variable v (y compris les variables importées et les variables provenant des bloc sous-modèles), la valeur *avant*, au cycle $k+1$, v_{k+1}, est égale à la valeur *après* du cycle k, v'_k.

$$
\left[\!\left[\begin{array}{l}\text{MACHINE } D\\ \text{IMPORTS } imp\\ \text{SEES } param\\ \text{INPUTS } i\\ \text{OUTPUTS } o\\ \text{VARIABLES } v\\ \text{INVARIANT } inv\\ \text{INITIALISATION } init\\ \text{OPERATIONS } C\\ \text{END}\end{array}\right]\!\right] (f) \equiv \left\{\begin{array}{l}\text{let}\\ \quad iv = RenamedImportedVar(D)\\ \quad hv = HiddenVar(C)\\ \quad av = v \cup iv \cup hv\\ \quad p = i \cup o\\ \quad ConstNames = getallnames(D)\\ \text{in}\\ \forall i^\omega = (i_0, i_1, ...)\\ \exists o^\omega = (o_0, o_1, ...), o'^\omega = (o'_0, o'_1, ...)\\ f(i^\omega) = (o^\omega, o'^\omega) \wedge\\ \exists av^\omega = (av_0, av_1, ...), av'^\omega = (av'_0, av'_1, ...)\\ \exists ConstNames\\ \circledast\end{array}\right.
$$

$$
\circledast \equiv \left\{\begin{array}{l} getallspecs(D) \wedge\\ [v', p' := v_0, p_0]\,(|init|)_{write(init)} \wedge\\ [hv' := hv_0]\,(|HiddenInit(C)|)_{write(HiddenInit(C))} \wedge\\ \bigwedge_{(n,C)\in imported(D)} \left(\begin{array}{l}\text{let}\\ \quad p = In(C) \cup Out(C)\\ \quad v = Var(C)\\ \quad imp = Imp(C)\\ \quad Sinit = Init(C) \ominus p\\ \text{in}\\ \quad [v' := n.v_0]\,(|Sinit|)_{v;imp}\end{array}\right) \wedge\\ \forall k \cdot (k \geq 0 \Rightarrow [p', p, av', av := p'_k, p_k, av'_k, av_k]\,(|C|)_{v\cup o;imp}) \wedge\\ \forall k \cdot (k \geq 0 \Rightarrow av_{k+1} = av'_k)\end{array}\right.
$$

Notez que p et p' dénotent respectivement les valeurs *avant* et les valeurs *après* des entrées et des sorties, bien que les entrées ne peuvent être écrites et les sorties ne peuvent être lues. La sémantique définie ici peut contenir certaines égalités de la forme $o' = o$, venu du fait que les sorties o sont dans l'ensemble support en écriture et que la définition du prédicat *avant-après* comporte, par exemple dans le cas de la substitution simple $s := E$, les égalités $\bigwedge_{y\in F-\{s\}} y' = y$. Ce genre d'égalité n'a aucune influence sur la sémantique. Ce cas n'apparaît pas pour les entrées, et avoir la substitution $[i' := i'_k]$ (inclue dans $[p' := p'_k]$) n'a aucun effet.

11.4 Exemple

La sémantique prédicative définie précédemment peut être calculée statiquement en fournissant le prédicat définissant la fonction f. Nous illustrons ce calcul en détaillant les différentes étapes du calcul sur un exemple. Nous reprenons l'exemple du compteur à rebours module 8 *counter3bitsmodulo* (figure 8.5 page 167). Rappelons que ce compteur importe le compteur *counter3bits* (figure 8.4 page 166). Comme il est nécessaire au calcul de la sémantique de *counter3bitsmodulo*, nous calculons d'abord le prédicat *avant-après* de *counter3bits*. Ce prédicat est ensuite utilisé pour calculer la sémantique de traces de *counter3bitsmodulo*.

11.4.1 Prédicats *avant-après* du modèle *counter3bits*

Il y a deux prédicats *avant-après* à calculer : le prédicat de la clause INITIALISATION et le prédicat de la clause OPERATIONS. Nous nous focalisons sur la clause OPERATIONS qui représente la difficulté

principale. Le modèle initial est donné sur la figure 8.4 page 166. Nous montrons ci-dessous comment la substitution est parenthésée.

$$\left(\begin{pmatrix}\text{IF } rst = TRUE \text{ THEN}\\ \quad n := seven3\\ \text{ELSE}\\ \quad \text{IF } n > zero3 \text{ THEN}\\ \quad\quad n := n - one3\\ \quad \text{END}\\ \text{END}\end{pmatrix} \parallel val := n\right) \parallel alm := bool(n = zero3)$$

Nous calculons d'abord le prédicat *avant-après* de la substitution IF imbriquée. Notez que, à cause de la composition parallèle, l'ensemble support en écriture à prendre en compte pour cette substitution est seulement $\{n\}$.

$$\left(\left|\begin{matrix}\text{IF } n > zero3 \text{ THEN}\\ \quad n := n - one3\\ \text{END}\end{matrix}\right|\right)_{\{n\}} \equiv \exists(n^3, n^4) \cdot \begin{pmatrix} n > zero3 \Rightarrow n' = n^3 \wedge\\ \neg(n > zero3) \Rightarrow n' = n^4 \wedge\\ n^3 = n - one3 \wedge\\ n^4 = n\end{pmatrix}$$

Le calcul du prédicat *avant-après* de la substitution IF globale est donné ci-dessous.

$$\left(\left|\begin{matrix}\text{IF } rst = TRUE \text{ THEN}\\ \quad n := seven3\\ \text{ELSE}\\ \quad \text{IF } n > zero3 \text{ THEN}\\ \quad\quad n := n - one3\\ \quad \text{END}\\ \text{END}\end{matrix}\right|\right)_{\{n\}} \equiv \exists(n^1, n^2) \cdot \begin{pmatrix} rst = TRUE \Rightarrow n' = n^1 \wedge\\ \neg(rst = TRUE) \Rightarrow n' = n^2 \wedge\\ n^1 = seven3 \wedge\\ [n' := n^2]\left(\left|\begin{matrix}\text{IF } n > zero3 \text{ THEN}\\ \quad n := n - one3\\ \text{END}\end{matrix}\right|\right)_{\{n\}}\end{pmatrix}$$

$$\equiv \exists(n^1, n^2) \cdot \begin{pmatrix} rst = TRUE \Rightarrow n' = n^1 \wedge\\ \neg(rst = TRUE) \Rightarrow n' = n^2 \wedge\\ n^1 = seven3 \wedge\\ \exists(n^3, n^4) \cdot \begin{pmatrix} n > zero3 \Rightarrow n^2 = n^3 \wedge\\ \neg(n > zero3) \Rightarrow n^2 = n^4 \wedge\\ n^3 = n - one3 \wedge\\ n^4 = n\end{pmatrix}\end{pmatrix}$$

On peut noter sur cet exemple la nécessité d'avoir une unicité des noms des nouvelles variables qui sont introduites par la quantification existentielle. Si nous avions seulement utilisé les décorations $*$ et $**$ pour les nouvelles variables, $[n' := n^2]$ serait $[n' := n^{**}]$ et il y aurait eu un conflit avec la variable n^{**} : $\exists(n^2) \cdot (\exists(n^4) \cdot (...n^2...))$ aurait été $\exists(n^{**}) \cdot (\exists(n^{**}) \cdot (...n^{**}...))$; la variable n^{**} serait liée à la deuxième quantification alors qu'elle devrait être liée à la première.

Le reste du calcul du prédicat *avant-après* la clause OPERATIONS ne contient pas de difficultés, nous obtenons :

$$\exists(n^1, n^2) \cdot \begin{pmatrix} rst = TRUE \Rightarrow n' = n^1 \wedge\\ \neg(rst = TRUE) \Rightarrow n' = n^2 \wedge\\ n^1 = seven3 \wedge\\ \exists(n^3, n^4) \cdot \begin{pmatrix} n > zero3 \Rightarrow n^2 = n^3 \wedge\\ \neg(n > zero3) \Rightarrow n^2 = n^4 \wedge\\ n^3 = n - one3 \wedge\\ n^4 = n\end{pmatrix}\end{pmatrix} \wedge \begin{matrix} val' = n \quad \wedge\\ alm' = bool(n = zero3)\end{matrix}$$

Le calcul du prédicat *avant-après* de la clause INITIALISATION est relativement simple, puisqu'il s'agit de substitutions simples composées en parallèle, nous obtenons :

$$rst' \in BOOL \wedge val' \in UINT3 \wedge alm' \in BOOL \wedge n' = seven3$$

11.4.2 Sémantique de traces du modèle *counter3bitsmodulo*

Nous pouvons maintenant calculer la sémantique de traces de compteur *counter3bitsmodulo*, le modèle BHDL est donné sur la figure 8.5 page 167. Nous commençons par les prédicats *avant-après* des clauses INITIALISATION et OPERATIONS, et nous les utilisons ensuite pour exprimer la sémantique de traces elle-même.

Prédicat *avant-après*

Nous calculons d'abord le prédicat *avant-après* de la clause OPERATIONS. Cette fois, la composition n'est pas parallèle mais pas séquentielle, l'ensemble support en écriture est $\{rstbis, alm, rval, val\}$. Cet ensemble est utilisé comme paramètre F pour tous les calculs des prédicats *avant-après* des substitutions. Nous rappelons la substitution de la clause OPERATIONS ci-dessous. Elle est composée de trois substitutions composées séquentiellement.

$$\begin{array}{l} rstbis := \mathrm{bool}(rval = one3 \vee rst = TRUE) \\ ; \\ (alm : alm,\ val : rval\ ) \leftarrow c3(rst : rstbis) \\ ; \\ val := rval \end{array}$$

Nous effectuons le calcul du prédicat *avant-après* en quatre étapes :

1. Le calcul du prédicat *avant-après* de $rstbis := \mathrm{bool}(rval = one3 \vee rst = TRUE)$.
2. Le calcul du prédicat *avant-après* de $(alm : alm,\ val : rval\ ) \leftarrow c3(rst : rstbis)$.
3. Le calcul du prédicat *avant-après* de la composition séquentielle de 1 et 2.
4. le calcul du prédicat *avant-après* de la composition séquentielle de 3 et $val := rval$. Le résultat est le prédicat *avant-après* de la clause OPERATIONS.

Ces quatre étapes sont détaillées ci-dessous.

1. Prédicat *avant-après* de $rstbis := \mathrm{bool}(rval = one3 \vee rst = TRUE)$
   $(|rstbis := \mathrm{bool}(rval = one3 \vee rst = TRUE)|)_{\{rstbis,alm,rval,val\}}$
   $$= rstbis' = bool(rval = one3 \vee rst = TRUE) \wedge alm' = alm \wedge rval' = rval \wedge val' = val$$
2. Prédicat *avant-après* de $(alm : alm,\ val : rval\ ) \leftarrow c3(rst : rstbis)$
   Le prédicat *avant-après* d'une importation de modèle est le prédicat *avant-après* du modèle importé dans lequel les noms des ports (entrées/sorties) sont remplacés par les noms des variables auxquelles ils sont connectés. Les variables locales sont également renommées en leur ajoutant en préfixe le nom du modèle duquel elles proviennent (c'est-à-dire le modèle qui est importé). Le nom $c3$ est un nom d'instance donné au modèle *counter3bits* (cf. clause IMPORTS du modèle, figure 8.5 page 167). En réutilisant le prédicat *avant-après* de *counter3bits* calculé dans la section précédente, nous obtenons :
   $(|(alm : alm,\ val : rval\ ) \leftarrow c3(rst : rstbis)|)_{\{rstbis,alm,rval,val\};\{c3:counter3bits\}}$

$$= \exists(n^1, n^2) \cdot \left(\begin{array}{l} rstbis = TRUE \Rightarrow ic3.n' = n^1 \wedge \\ \neg(rstbis = TRUE) \Rightarrow ic3.n' = n^2 \wedge \\ n^1 = seven3 \wedge \\ \exists(n^3, n^4) \cdot \left(\begin{array}{l} ic3.n > zero3 \Rightarrow n^2 = n^3 \wedge \\ \neg(ic3.n > zero3) \Rightarrow n^2 = n^4 \wedge \\ n^3 = ic3.n - one3 \wedge \\ n^4 = ic3.n \end{array} \right) \end{array} \right)$$

$$\wedge\ rval' = ic3.n \wedge alm' = bool(ic3.n = zero3)$$

$$\wedge\ rstbis' = rstbit \wedge val' = val$$

3. Prédicat *avant-après* de la composition séquentielle de 1 et 2

   La composition séquentielle des deux premières substitutions introduit des nouveaux noms de variables dans le prédicat. Nous utilisons des noms de variables décorés avec un 5 (pour éviter des conflits de noms), du type x^5. Dans le prédicat *avant-après* de la première substitution, les valeurs *après* x' sont remplacées par x^5, et dans le prédicat de la deuxième substitution , les valeurs *avant* sont remplacées par x^5.

$$\exists \begin{pmatrix} rstbis^5, \\ alm^5, \\ rval^5, \\ val^5 \end{pmatrix} \cdot \begin{pmatrix} rstbis^5 = bool(rval = one3 \vee rst = TRUE) \\ \wedge\ alm^5 = alm \ \wedge\ rval^5 = rval \ \wedge\ val^5 = val \\ \wedge\ \exists (n^1, n^2) \cdot \begin{pmatrix} rstbis^5 = TRUE \Rightarrow ic3.n' = n^1 \wedge \\ \neg(rstbis^5 = TRUE) \Rightarrow ic3.n' = n^2 \wedge \\ n^1 = seven3 \wedge \\ \exists \begin{pmatrix} n^3, \\ n^4 \end{pmatrix} \cdot \begin{pmatrix} ic3.n > zero3 \Rightarrow n^2 = n^3 \wedge \\ \neg(ic3.n > zero3) \Rightarrow n^2 = n^4 \wedge \\ n^3 = ic3.n - one3 \wedge \\ n^4 = ic3.n \end{pmatrix} \end{pmatrix} \\ \wedge\ rval' = ic3.n \ \wedge\ alm' = bool(ic3.n = zero3) \\ \wedge\ rstbis' = rstbit^5 \ \wedge\ val' = val^5) \end{pmatrix}$$

4. Prédicat *avant-après* de la composition compositionnelle de 3 et $val := rval$

   La substitution restante $val := rval$, composée séquentiellement avec les substitutions dont nous avons déjà calculé le prédicat *avant-après* ne pose pas de difficulté : on introduit des nouveaux noms de variables décorés avec 6 qui remplacent les valeurs *après* des substitutions précédentes et les valeurs *avant* de $val := rval$. Nous obtenons finalement le prédicat ci-dessous, qui est le prédicat *avant-après* de la clause OPERATIONS.

$$\exists \begin{pmatrix} rstbis^6, \\ alm^6, \\ rval^6, \\ val^6 \end{pmatrix} \cdot \begin{pmatrix} \exists \begin{pmatrix} rstbis^5, \\ alm^5, \\ rval^5, \\ val^5 \end{pmatrix} \cdot \begin{pmatrix} rstbis^5 = bool(rval = one3 \vee rst = TRUE) \\ \wedge\ alm^5 = alm \ \wedge\ rval^5 = rval \ \wedge\ val^5 = val \\ \wedge\ \exists \begin{pmatrix} n^1, \\ n^2 \end{pmatrix} \cdot \begin{pmatrix} rstbis^5 = TRUE \Rightarrow ic3.n' = n^1 \wedge \\ \neg(rstbis^5 = TRUE) \Rightarrow ic3.n' = n^2 \wedge \\ n^1 = seven3 \wedge \\ \exists \begin{pmatrix} n^3, \\ n^4 \end{pmatrix} \cdot \begin{pmatrix} ic3.n > zero3 \Rightarrow n^2 = n^3 \wedge \\ \neg(ic3.n > zero3) \Rightarrow n^2 = n^4 \wedge \\ n^3 = ic3.n - one3 \wedge \\ n^4 = ic3.n \end{pmatrix} \end{pmatrix} \\ \wedge\ rval^6 = ic3.n \ \wedge\ alm^6 = bool(ic3.n = zero3) \\ \wedge\ rstbis^6 = rstbis^5 \ \wedge\ val^6 = val^5) \end{pmatrix} \\ \wedge\ val' = rval^6 \ \wedge\ rstbis' = rstbis^6 \ \wedge\ alm' = alm^6 \ \wedge\ rval' = val^6 \end{pmatrix}$$

Le prédicat *avant-après* de la clause INITIALISATION s'obtient simplement par la conjonction des prédicats *avant-après* de toutes les substitutions simples.

$$rst' \in BOOL \ \wedge\ val' \in UINT3 \ \wedge\ alm' \in BOOL \ \wedge\ rval' = 0 \ \wedge\ rstbis' \in BOOL$$

Sémantique de trace

La sémantique de traces est obtenue principalement en élevant au rang de prédicat sur les traces les prédicats *avant-après* calculés précédemment. Nous reprenons les six points définis dans la section 11.3.2, page 200, permettant de définir cette sémantique de traces.

1. Récupération des définitions des constantes définies dans les machines déclaratives vues. En l'occurrence, les constantes *seven*3, *one*3, *zero*3 et les types $UINT3$ et $BOOL$.

$$seven3 = 7 \ \wedge\ one3 = 1 \ \wedge\ zero3 = 0 \ \wedge\ UINT3 = 0..7 \ \wedge\ BOOL = \{FALSE, TRUE\}$$

2. Élévation au rang de prédicat sur les traces *avant-après* de la clause INITIALISATION. Ceci est fait en remplaçant les valeurs *après* par les noms des variables indicés par 0 (éléments initiaux des traces des valeurs *avant*).

$$rst_0 \in BOOL \wedge val_0 \in UINT3 \wedge alm_0 \in BOOL \wedge rval_0 = 0 \wedge rstbis_0 \in BOOL$$

3. Importation et élévation au rang de prédicat sur les traces du prédicat *avant-après* de la clause INITIALISATION des modèles importés (ici *counter3bits*). Les initialisations des entrées et des sorties sont supprimées (on supprime totalement les substitutions simples concernant les ports de la clause INITIALISATION) puisque les ports sont remplacés par des variables du modèle importateur (qui sont déjà initialisées). Les noms des variables locales au modèle importé sont préfixés par le nom d'instance du modèle. Dans notre cas, il ne reste que l'initialisation de la variable n.

$$ic3.n_0 = seven3$$

4. Élévation du prédicat *avant-après* de la clause OPERATIONS que nous avons calculé précédemment au rang de prédicat sur les traces . Cette élévation consiste à remplacer les valeurs *après* (par exemple x') par les éléments des traces des valeurs au cycle k *après* (par exemple x'_k), et à remplacer les valeurs *avant* par les éléments des traces des valeurs *avant* au cycle k (par exemple x_k). On ajoute la quantification universelle sur la variable k représentant le cycle courant. Le prédicat sur les traces que nous obtenons est donné ci-dessous.

$$\forall k \cdot \left(\begin{array}{l} k \geq 0 \Rightarrow \exists \begin{pmatrix} rstbis^5, \\ alm^5, \\ rval^5, \\ val^5, \\ rstbis^6, \\ alm^6, \\ rval^6, \end{pmatrix} \cdot \left(\begin{array}{l} rstbis^5 = bool(rval = one3 \vee rst = TRUE) \\ \wedge\ alm^5 = alm_k \ \wedge\ rval^5 = rval_k \ \wedge\ val^5 = val_k \\ \\ \wedge\ \exists (n^1, n^2) \cdot \left(\begin{array}{l} rstbis^5 = TRUE \Rightarrow ic3.n'_k = n^1 \wedge \\ \neg(rstbis^5 = TRUE) \Rightarrow ic3.n'_k = n^2 \wedge \\ n^1 = seven3 \wedge \\ \exists (n^3, n^4) \cdot \left(\begin{array}{l} ic3.n_k > zero3 \Rightarrow n^2 = n^3 \wedge \\ \neg(ic3.n_k > zero3) \Rightarrow n^2 = n^4 \\ \wedge\ n^3 = ic3.n_k - one3 \\ \wedge\ n^4 = ic3.n_k \end{array} \right) \end{array} \right) \\ \\ \wedge\ rval^6 = ic3.n_k \ \wedge\ alm^6 = bool(ic3.n_k = zero3) \\ \wedge\ rstbis^6 = rstbit^5 \ \wedge\ val^6 = val^5) \\ \wedge\ val'_k = rval^6 \ \wedge\ rstbis'_k = rstbis^6 \\ \wedge\ alm'_k = alm^6 \ \wedge\ rval'_k = val^6 \end{array} \right) \end{array} \right)$$

5. Dans notre exemple nous n'avons pas de bloc sous-modèle
6. Prédicat établissant le lien entre les valeurs *après* d'un cycle aux valeurs *avant* du cycle suivant. Sans oublié la variable importée n, renommée $ic3.n$.

$$\forall k \cdot \left(k \geq 0 \Rightarrow \begin{pmatrix} val_{k+1} = val'_k \ \wedge\ rstbis_{k+1} = rstbis'_k \\ \wedge\ alm_{k+1} = alm'_k \ \wedge\ rval_{k+1} = rval'_k \\ \wedge\ ic3.n_{k+1} = ic3.n'_k \end{pmatrix} \right)$$

Le prédicat sur les traces définissant la sémantique du modèle du compteur à rebours modulo 8 est la conjonction des cinq prédicats listés ci-dessus. Pour être complet, il faut rajouter la définition de la fonction f qui dit être définie par la sémantique prédicative et la quantification sur toutes les variables : les entrées sont quantifiées universellement, les sorties, les constantes et les variables sont quantifiées existentiellement. En notant $\mathcal{P}$ le prédicat résultant de la conjonction de tous les prédicats calculés plus haut, $\mathcal{P} \equiv 1 \wedge 2 \wedge 3 \wedge 4 \wedge 6$, on obtient la sémantique prédicative suivante :

$$\forall rst^\omega \cdot \exists (val^\omega, alm^\omega) \cdot \left(f(rst^\omega) = (val^\omega, alm^\omega) \ \wedge\ \exists \begin{pmatrix} rstbis^\omega, rval^\omega, \\ ic3.n^\omega, seven3, \\ one3, zero3, \\ UINT3, BOOL \end{pmatrix} \cdot (\mathcal{P}) \right)$$

TAB. 11.1 – Règles de transformation de la sémantique

variable x	lecture de x	écriture de x
	$[x_n := \cdot]$	$[x'_n := \cdot]$
entrée	x_n	
sortie		x_n
fils		x_n
registre	x_n	x_{n+1}

11.5 Où sont les registres et les fils ?

La sémantique prédicative a été définie plus haut par élévation au rang de prédicat sur les traces des prédicats *avant-après* des substitutions utilisées par les modèles BHDL. Comme un modèle BHDL, ce prédicat est basé sur des variables, plus exactement sur les valeurs *avant* et *après* des variables pour chaque cycle. Dans un circuit électronique, les valeurs qui sont échangées entre les composants ne sont pas portées par des variables mais par des fils. Certains de ces fils étant des entrées/sorties de registres qui mémorisent des valeurs le temps d'un cycle. Selon les principes de modélisation que nous avons choisis (cf. chapitre 4), certaines variables sont utilisées pour modéliser des fils, et d'autres modélisent des registres. La sémantique prédicative telle que nous l'avons définie ne fait pas cette distinction explicitement. Il est nécessaire d'indiquer comment effectuer cette distinction pour avoir une interprétation correcte du modèle comme un circuit électronique. Un différence notable entre une variable représentant un fils et une variable représentant un registre est la façon dont leurs affectations sont interprétées :

- Affecter une valeur à une variable représentant un fil signifie que la valeur portée par le fils est modifiée,
- Affecter une valeur à une variable représentant un registre signifie que la valeur mémorisée par le registre est inchangée pendant le cycle courant et que la valeur assignée sera la valeur mémorisée par le registre pendant le cycle suivant (à moins qu'il n'y ait une autre affectation avant la fin du cycle courant). Cela dit, si, par la suite (donc pas dans une substitution parallèle), dans le modèle, on lit cette variable qui a été modifiée, on accède à la nouvelle valeur, non à l'ancienne.

Transformer la sémantique prédicative définie plus haut pour obtenir une sémantique avec des fils et des registres nécessite de séparer les variables qui représentent les fils et les variables qui représentent des registres. Dans cette nouvelle forme de la sémantique, nous ne cherchons plus à connaître les valeurs *avant* et les valeurs *après*, mais à donner pour chaque variable sa valeur à chaque cycle. Pour une variable x, x_n dénote la valeur de la variable au cycle n. Dans le cas des variables modélisant des entrées, des sorties et des fils nous faisons un choix : dans certains cas (pour les entrées) la valeur de la variable est la valeur *avant*, dans les autres (les sorties et les fils), la valeur de la variable est la valeur *après*. Ces choix correspondent aux valeurs qui sont observées dans le circuit. Pour les entrées et les sorties, le choix est simple puisqu'il n'existe pas de valeurs *après* pour les entrées, et pas de valeurs *avant* pour les sorties (elles peuvent exister formellement dans la sémantique mais n'ont pas de sens du point du vue du circuit physique). En ce qui concerne les fils, il faut rappeler que nous n'observons pas l'évolution de l'état des fils pendant la durée du cycle (les états intermédiaires sont modélisés par les variables intermédiaires qui sont introduites pendant le calcul des prédicats *avant-après*, en particulier dans le cas de la composition séquentielle). La valeur d'un fil en début de cycle n'a pas vraiment d'intérêt, puisque le résultat du calcul ne dépend pas des valeurs des fils en tout début de cycle (sinon ce serait des registres). Nous choisissons donc d'observer les valeurs *après* des fils, comme dans le cas des sorties. Ce choix correspond également aux valeurs qui sont affichées par un simulateur de circuit lorsque l'on fixe les délais de propagation à zéro : les valeurs des fils qui sont affichées sont les valeurs des sorties des composants.

Nous avons déjà expliqué comment effectuer la classification des variables en entrées/sorties et en fils/registres dans la section sur les ensembles supports (cf. section 9.2.4). La transformation elle-même peut être faite en appliquant quelques substitutions sur la sémantique prédicative. Ces substitutions sont résumées dans le tableau 11.1. Nous discutons les différents cas ci-dessous.

- La valeur d'une entrée x au cycle n est la valeur *avant* de x au cycle n. Ainsi, nous remplaçons x_n par x_n dans la sémantique (pas de modification). Notez que x'_n n'existe pas pour les entrées.
- La valeur d'une sortie x au cycle n est la valeur *après* de x au cycle n, c'est-à-dire x'_n. Nous remplaçons donc x'_n par x_n dans la sémantique. Remarquez que, dans le cas des sorties, x_n existe dans la sémantique mais uniquement dans des égalités du type $x_n = x^t$ où x^t est une variable intermédiaire qui n'est finalement jamais utilisée. En conséquence, x_n peut être remplacée par n'importe quelle variable, rester inchangée par exemple.
- Le cas d'un fil est proche de celui d'une sortie. La seule différence est qu'une variable représentant un fil peut être lue (ce qui n'est pas le cas d'une sortie). Cependant, la variable n'est jamais lue avant d'être écrite (sinon ce serait un registre, non un fil). La valeur *avant* d'une variable représentant un fil n'est donc jamais utilisée, elle peut être laissée telle quelle dans la sémantique.
- Une variable x représentant un registre est à la fois lue et écrite. La valeur *avant* x_n est lue et la valeur *après* x'_n est écrite. La valeur *avant* est celle qui est utilisée pendant la durée du cycle, c'est-à-dire la valeur mémorisée par le registre. La valeur *après* est la valeur que le registre mémorisera pendant le cycle suivant, donc x'_n est remplacé par x_{n+1}.

11.6 Résumé

Dans ce chapitre, nous avons défini une sémantique pour BHDL basée sur les prédicats. Cette sémantique n'est pas à proprement parler compositionnelle. Elle a cependant l'avantage de respecter l'architecture du modèle. C'est pourquoi cette sémantique sert de base à la traduction de modèles BHDL vers d'autres langages de modélisation de circuits. Elle pourrait être utilisée pour faire de la simulation en utilisant un résolveur de contraintes. Les règles données ici ont été implantées en Prolog, permettant ainsi de générer automatiquement le prédicat.

Chapitre 12

Sémantique relationnelle

La sémantique prédicative définie au chapitre précédent construit un prédicat qui définit la fonctionnalité du circuit. Elle montre la structure du circuit et est utile pour la simulation ou la traduction vers d'autres formalismes de description de circuits. Nous définissons maintenant une sémantique à usage plus formel. Elle définit formellement la relation qui existe entre les traces des entrées et les traces des sorties. Cette sémantique est compositionnelle et peut être utilisée pour faire des preuves inductives. Nous verrons qu'elle est particulièrement utile pour étudier la réutilisation de modèles, qui est elle-même une construction inductive du langage BHDL. La définition de la réutilisation de modèles est particulièrement simple dans cette sémantique (cf. section 12.3).

Cette sémantique a été définie pour trois raisons. La première raison était d'avoir, en plus d'une sémantique prédicative, une sémantique relationnelle, comme pour le langage B où deux sémantiques sont données : une en utilisant les prédicats et une autre en utilisant les relations. La deuxième raison était d'étudier la possibilité d'avoir une sémantique compositionnelle, souvent appréciée pour faire des raisonnements formels. La troisième raison est d'avoir une base formelle pour l'importation de composants en utilisant ACL2 (chapitre 15) : la preuve faite avec ACL2 prouve une bisimulation entre un modèle VHDL et un modèle BHDL, et la propriété 17 obtenue à partir de la sémantique relationnelle nous dit qu'on peut échanger deux composants ayant la même sémantique, c'est-à-dire étant bisimulaires.

L'idée principale de cette sémantique est de voir chaque substitution généralisée comme un sous-composant et nous lui donnons une sémantique de trace. Le principe est d'associer à chaque substitution une relation qui relie les traces des valeurs *avant* des variables lues par la substitution aux traces des valeurs *après*. Formellement, cela consiste à élever au rang de relation sur les traces la relation rel sur les substitutions, définie par la méthode B (cf. section 9.1.1). La sémantique du modèle BHDL est alors obtenue en composant les sémantiques des substitutions définissant le modèle. Outre l'utilisation des traces, une différence avec la relation rel de la méthode B est que celle-ci considère toujours toutes les variables du modèle alors que notre sémantique ne considère, pour chaque substitution, que les variables qui sont lues (elles sont considérées comme des entrées de la substitution) ou écrites (elles sont considérées comme des sorties).

En fait, nous définissons deux sémantiques : une sémantique amnésique, et une sémantique complète. La première sémantique ne prend pas en compte le fait que certaines variables modélisent des registres, dans cette sémantique, il n'existe pas de lien entre les cycles successifs. De telles variables, qui sont lues et écrites par la substitution, sont considérées comme des entrées *et* des sorties du composant représenté par la substitution. La sémantique amnésique correspond à la partie combinatoire du circuit dans laquelle les registres sont remplacés par des entrées et des sorties. La sémantique complète considère les variables modélisant des registres comme des mémoires qui sont cachées à l'environnement.

L'intérêt de la sémantique amnésique est qu'elle est compositionnelle sur l'ensemble du langage, ce qui n'est pas le cas de la sémantique complète lorsque l'on considère les substitutions d'importation de modèles existants ou les substitutions blocs sous-modèles. Ceci est dû au fait que la composition de deux substitutions modifie les ensembles supports, en particulier dans le cas de la composition séquentielle : certaines variables peuvent être considérées comme des mémoires dans une des substitutions mais pas par la substitution qui résulte de la composition. Par exemple, dans la substitution $x := x + 1$, la variable

x est lue et écrite, elle est considérée comme une mémoire. Mais si on compose cette substitution pour obtenir $x := 0; x := x + 1$, la variable x n'est plus considérée comme une mémoire car elle n'est pas lue par la substitution. Comme la sémantique amnésique ne cache pas les mémoires, elle permet de traiter simplement ce type de situation. En fait, nous verrons que dans cet exemple, la sémantique complète est également compositionnelle, mais que cette compositionnalité peut être perdue (pour la sémantique complète uniquement) lorsqu'on importe des modèles existants ou qu'on utilise les blocs sous-modèles.

La sémantique complète est définie à partir de la sémantique amnésique et nous discutons à la fin de ce chapitre les cas dans lesquels elle est compositionnelle ou non. Nous commençons par définir les deux sémantiques dans les cas où il n'y a pas de mémoires cachées (quand l'importation et les blocs sous-modèles ne sont pas utilisés), ce sera l'occasion de montrer que les deux sémantiques sont compositionnelles dans ce cas. Puis nous expliquons comment définir les sémantiques des importations et des blocs sous-modèles.

12.1 Sémantiques dans le cas de modèles sans importation ni bloc sous-modèle

Cette section définit la sémantique amnésique et la sémantique complète dans le cas des substitutions sans bloc sous-modèle et sans importation de modèle. La sémantique amnésique est d'abord définie en élevant la relation *rel* de la méthode B au rang de relation sur des traces. De plus, cette relation est restreinte aux variables qui sont effectivement utilisées par la substitution (quelles soient lues ou écrites). Ensuite, la sémantique complète est définie sur la base de la sémantique amnésique. Cela est fait en connectant, pour les variables qui modélisent des mémoires, les traces des entrées aux traces des sorties : la valeur fournie en sortie au cycle n est réutilisée comme entrée au cycle $n + 1$. Ceci restaure l'évolution du système contenant des mémoires qui sont perdues dans la sémantique amnésique.

12.1.1 Principes

Chaque substitution est comme un sous-circuit qui a des entrées et des sorties. Les entrées sont les variables lues par la substitution (ensemble support en lecture) et les sorties les variables qui sont écrites (ensemble support en écriture). La sémantique associe à toute substitution une relation qui lie les traces des entrées et les traces des sorties. Cette relation est un modèle de la fonctionnalité de la substitution.

Comme dans le cas des prédicats *avant-après*, nous séparons les valeurs *avant* et les valeurs *après* des variables. On peut distinguer deux types de variables dans une substitution :

- Les variables qui sont lues, elles constituent l'ensemble support en lecture ; la substitution utilise les valeurs *avant* de ces variables. Elles sont considérées comme les entrées de la substitution.
- Les variables qui sont écrites, elles constituent l'ensemble support en écriture ; la substitution fournit les valeurs *après* de ces variables. Elles sont considérées comme les sorties de la substitution.

Nous commençons d'abord par définir la sémantique amnésique, notée $(\!|\cdot|\!)$. Dans cette sémantique, le comportement du circuit lors d'un cycle ne dépend pas du comportement du circuit au cycle précédent. Les variables modélisant des mémoires (c'est-à-dire les variables qui sont à la fois lues et écrites), sont considérées comme des entrées et des sorties de la substitution. Nous définissons ensuite la sémantique complète, notée $\{\!|\cdot|\!\}$. Dans cette sémantique, lorsqu'une variable est à la fois lue et écrite, c'est-à-dire qu'elle est à la fois une entrée et une sortie, la valeur prise en entrée doit être la valeur fournie en sortie lors du cycle précédent. Il est préférable de définir d'abord la sémantique amnésique car chaque composition (parallèle, séquentielle ou conditionnelle) de substitutions modifie les ensembles supports et les liens entre les variables. Par exemple, dans la substitution $x := x + 1$ la variable x est lue et écrite : la valeur lue au cycle $n+1$ est la valeur écrite au cycle n. Cependant, dans la substitution $x := 2; x := x+1$, x est seulement écrite et le résultat, à chaque cycle, ne dépend pas des cycles précédents.

12.1.2 Sémantique amnésique

La sémantique amnésique est basée sur la relation entre les traces des entrées et les traces des sorties. Les entrées sont les variables qui sont lues par la substitution (ensemble support en lecture), et les sorties

sont les variables qui sont écrites par la substitution (ensemble support en écriture). Si une variable est à la fois lue et écrite par la substitution (c'est-à-dire que cette variable modélise un registre), cette variable est placée à la fois en entrée *et* en sortie. Cette sémantique est dite amnésique car elle ne contient pas de mémoires cachées : la relation entre les entrées et les sorties lors d'un cycle donné ne dépend pas des cycles précédents. Pour obtenir la relation qui tienne compte des mémoires, il faut connecter les entrées aux sorties du cycle précédent (cf. sémantique complète).

Définition 16 (Sémantique amnésique d'une substitution) *La sémantique amnésique d'une substitution S est définie comme suit. Les ensembles R_S, W_S et W_S^a sont, respectivement, les ensembles supports en lecture, en écriture et total en écriture de la substitution S.*

$$[\![S]\!] = \{R_S\} \times \{W_S\} \times \{W_S^a\} \times rel^{\omega}_{R_S \times W_S}(S)$$

Pour une substitution S qui ne contient *aucun* bloc sous-modèle ou importation de modèle, la relation $rel^{\omega}_{R_S \times W_S}$ est la relation rel de la méthode B élevée au rang de relation sur les traces et restreinte de façon à ce que son domaine soit l'ensemble support en lecture (R_S) et que son codomaine soit l'ensemble support en écriture (W_S) de la substitution. Nous donnons ci-dessous la définition de rel^{ω} qui est la relation rel élevée au rang de relation sur des traces (sans restriction sur son domaine et son codomaine), et la définition de $rel^{\omega}_{R_S \times W_S}$ en utilisant rel^{ω}.

Définition 17 (Relation rel élevée au rang de relation sur les traces) *Pour une substitution S, nous définissons la relation $rel^{\omega}(S)$, qui est la relation rel de la méthode B élevée au rang de relation sur les traces. On pourrait définir $(rel(S))^{\omega}$, l'ensemble des traces formées de couples $x \mapsto y \in rel(S)$, mais il est plus pratique de manipuler des couples de traces $x^{\omega} \mapsto y^{\omega}$, où x^{ω} est une trace de la variable x. Il y a une bijection entre $rel^{\omega}(S)$ et $(rel(S))^{\omega}$.*

$$rel^{\omega}(S) = \{x^{\omega} \mapsto y^{\omega} | \forall k \cdot (k \geq 0 \Rightarrow x_k \mapsto y_k \in rel(S))\}$$

Définition 18 (Relation sur les traces restreinte aux ensembles supports) *Nous utilisons la notation $rel^{\omega}_{R_S \times W_S}(S)$ pour désigner la relation sur les traces correspondant à la substitution S et dont le domaine est restreint à l'ensemble support en lecture R_S et le codomaine à l'ensemble support en écriture W_S. La relation $rel^{\omega}_{R_S \times W_S}(S)$ est la relation qui lie les traces des variables lues (entrées) aux traces des variables écrites (sorties). Dans les formules ci-dessous, l'ensemble V est l'ensemble de toutes les variables et l'opérateur $d^{\omega}(V)$ donne le domaine (le "type") des traces de V.*

Dans le cas où l'ensemble support en lecture de la substitution n'est pas vide, on pose la définition suivante :

$$\begin{aligned} rel^{\omega}_{R_S \times W_S}(S) = \{x^{\omega} \mapsto y^{\omega} | x^{\omega} \in d^{\omega}(R_S) \wedge \\ y^{\omega} \in d^{\omega}(W_S) \wedge \\ \exists (X^{\omega}, Y^{\omega}) \cdot (X^{\omega} \in d^{\omega}(V) \wedge \\ Y^{\omega} \in d^{\omega}(V) \wedge \\ X^{\omega} \mapsto Y^{\omega} \in rel^{\omega}(S) \wedge \\ x^{\omega} = select_{V \to R_S}(X^{\omega}) \wedge \\ y^{\omega} = select_{V \to W_S}(Y^{\omega}))\} \end{aligned}$$

Dans le cas où R_S serait l'ensemble vide (c'est le cas de la substitution de la clause INITIALISATION *par exemple), l'égalité $x^{\omega} = select_{V \to R_S}(X^{\omega})$ n'a pas de sens, la définition est la suivante :*

$$\begin{aligned} rel^{\omega}_{\varnothing \times W_S}(S) = \{y^{\omega} \mid y^{\omega} \in d^{\omega}(W_S) \wedge \\ \exists (X^{\omega}, Y^{\omega}) \cdot (X^{\omega} \in d^{\omega}(V) \wedge \\ Y^{\omega} \in d^{\omega}(V) \wedge \\ X^{\omega} \mapsto Y^{\omega} \in rel^{\omega}(S) \wedge \\ y^{\omega} = select_{V \to W_S}(Y^{\omega}))\} \end{aligned}$$

Si la relation $rel^{\omega}_{R_S \times W_S}(S)$ est restreinte par rapport à $rel^{\omega}(S)$, on peut tout de même retrouver $rel^{\omega}(S)$ à partir de $rel^{\omega}_{R_S \times W_S}(S)$ car les variables qui ne sont pas dans l'ensemble support en écriture ne sont pas modifiées, la relation pour ces variables est donc l'identité. Cette propriété est importante pour pouvoir prouver la compositionnalité de $rel^{\omega}_{R_S \times W_S}(S)$, et par suite la compositionnalité de la sémantique.

Propriété 4 (Définition de $rel^{\omega}(S)$ à partir $rel^{\omega}_{R_S \times W_S}(S)$) *La relation sur les traces non restreinte aux ensembles supports peut être retrouvée à partir de la relation restreinte. Le principe est que les variables qui ne sont pas dans l'ensemble support en écriture ne sont pas modifiées. Ainsi, la relation sur ces variables est l'identité ($select_{V \to V - W_S}(Y^{\omega}) = select_{V \to V - W_S}(X^{\omega})$).*

Dans le cas où l'ensemble support en lecture de la substitution n'est pas vide, on a la propriété suivante :

$$\begin{aligned} rel^{\omega}(S) = \{X^{\omega} \mapsto Y^{\omega} | X^{\omega} \in d^{\omega}(V) \wedge \\ Y^{\omega} \in d^{\omega}(V) \wedge \\ \exists (x^{\omega}, y^{\omega}) \cdot (x^{\omega} \mapsto y^{\omega} \in rel^{\omega}_{R_S \times W_S}(S) \wedge \\ select_{V \to V - W_S}(Y^{\omega}) = select_{V \to V - W_S}(X^{\omega}) \wedge \\ x^{\omega} = select_{V \to R_S}(X^{\omega}) \wedge \\ y^{\omega} = select_{V \to W_S}(Y^{\omega})\} \end{aligned}$$

Dans le cas où l'ensemble support en lecture de la substitution serait vide, on a la propriété suivante :

$$\begin{aligned} rel^{\omega}(S) = \{X^{\omega} \mapsto Y^{\omega} | X^{\omega} \in d^{\omega}(V) \wedge \\ Y^{\omega} \in d^{\omega}(V) \wedge \\ \exists y^{\omega} \cdot (y^{\omega} \in rel^{\omega}_{R_S \times W_S}(S) \wedge \\ select_{V \to V - W_S}(Y^{\omega}) = select_{V \to V - W_S}(X^{\omega}) \wedge \\ y^{\omega} = select_{V \to W_S}(Y^{\omega})\} \end{aligned}$$

Nous ne définissions pas la relation restreinte $rel_{R_S \times W_S}$ (relation sur les variables et non pas sur les traces) car il n'est pas possible de la définir pour l'importation de modèle ou les blocs sous-modèles. Pour ces deux substitutions, il est nécessaire d'utiliser des traces, donc la relation $rel^{\omega}_{R_S \times W_S}$, qui est définie pour ces deux substitutions dans les sections 12.3 et 12.2. Si nous voulions absolument définir une relation pour ces substitutions sans utiliser les traces, il serait nécessaire de récupérer l'ensemble des variables cachées par ces constructions et de les rendre globales, comme cela a été fait dans la sémantique prédicative.

Sémantiques amnésiques des substitutions simples

Comme exemples de sémantiques amnésiques, nous donnons ci-dessous les sémantiques des substitutions simples. La figure 12.1 donne la représentation graphique des circuits (implicites) représentés par les deux substitutions simples.

FIG. 12.1 – Substitutions simples déterministes et non déterministes

- Affectation déterministe, $s := E$

  Pour chaque valeur x des variables de E correspond une valeur pour s qui est $E(x)$. Le fait que s puisse être dans $free(E)$ (les variables libres de E), et serait ainsi une mémoire, n'a pas d'importance puisque qu'il s'agit d'une sémantique amnésique.

  $R_{s:=E} = free(E)$

  $W_{s:=E} = \{s\}$

  $W^a_{s:=E} = \{s\}$

  Dans le cas où E n'est pas une constante, $free(E)$ n'est pas vide et on obtient le résultat suivant : $[\![s := E]\!] = \{R_{s:=E}\} \times \{W_{s:=E}\} \times \{W^a_{s:=E}\} \times \{x^{\omega} \mapsto y^{\omega} | (x^{\omega} = \langle x_0, x_1, ... \rangle) \in d^{\omega}(R_{s:=E}) \wedge (y^{\omega} = \langle y_0, y_1, ... \rangle) \in d^{\omega}(W_{s:=E}) \wedge \forall k \cdot (k \geq 0 \Rightarrow y_k = E(x_k))\}$

  Dans le cas où E est une constante, on a : $[\![s := E]\!] = \{R_{s:=E}\} \times \{W_{s:=E}\} \times \{W^a_{s:=E}\} \times \{y^{\omega} | ((y^{\omega} = \langle y_0, y_1, ... \rangle) \in d^{\omega}(W_{s:=E}) \wedge \forall k \cdot (k \geq 0 \Rightarrow y_k = E)\}$

- Affectation non déterministe, $s :\in C$

  Il n'y a aucune entrée puisque C doit être une constante. La sortie s peut prendre toutes les valeurs de l'ensemble C. Dans ce cas l'ensemble support en lecture est vide car C doit être une constante.

$R_{s:\in C} = \varnothing$
$W_{s\in C} = \{s\}$
$W^a_{s:\in C} = \{s\}$
$[\![s :\in C]\!] = \{R_{s:\in C}\} \times \{W_{s:\in C}\} \times \{W^a_{s:\in C}\} \times \{y^\omega | (y^\omega = \langle y_0, y_1, ...\rangle) \in d^\omega(W_{s:\in C}) \wedge \ \forall k \cdot (k \geq 0 \Rightarrow y_k \in C)\}$

Notations

Pour toute sémantique de la forme $[\![S]\!] = \{R_S\} \times \{W_S\} \times \{W^a_S\} \times f$, nous utiliserons parfois les notations suivantes, elles permettent de retrouver les ensembles supports et la relation sur les traces à partir de la sémantique, et non directement à partir de la substitution.
$\rho([\![S]\!]) = R_S$,
$\phi([\![S]\!]) = W_S$,
$\psi([\![S]\!]) = W^a_S$, et,
$\delta([\![S]\!]) = f$
Nous utiliserons également parfois la notation $\mathcal{F}_S$ pour se référer directement aux trois ensembles supports (en lecture, en écriture et total en écriture) : $\mathcal{F}_S = (R_S, W_S, W^a_S)$

12.1.3 Sémantique complète

Pour obtenir la sémantique complète à partir de la sémantique amnésique, il faut simplement *retirer* toutes les séquences de cycles dans lesquels, pour toutes les variables qui sont à la fois lues et écrites (mémoires), les entrées à chaque cycle ne correspondent pas aux sorties du cycle précédent. La formule $select_{R\to R\cap W}(x^\omega_{+1}) = select_{W\to R\cap W}(y^{\omega\to length(x^\omega_{+1})})$ spécifie que les traces x^ω et y^ω doivent être égales si on décale x^ω d'un cycle. Le changement de taille $y^{\omega\to length(x^\omega_{+1})}$ de y^ω est nécessaire car le décalage d'un cycle de x^ω réduit sa taille d'un cycle.

Définition 19 (Sémantique complète) *La sémantique complète est définie à partir de la sémantique amnésique de la façon suivante :*
$[\![S]\!] = \{R\} \times \{W\} \times \{W^a\} \times f \ \wedge \ R \cap W \neq \varnothing$
$\Rightarrow$
$(\!|S|\!) = \{R\} \times \{W\} \times \{W^a\} \times \{x^\omega \mapsto y^\omega | (R, W, W^a, x^\omega \mapsto y^\omega) \in f \wedge$
$select_{R\to R\cap W}(x^\omega_{+1}) = select_{W\to R\cap W}(y^{\omega\to length(x^\omega_{+1})})\}$
Si l'intersection $R \cap W$ est vide (pas de mémoires), la sémantique complète est égale à la sémantique amnésique :
$[\![S]\!] = \{R\} \times \{W\} \times \{W^a\} \times f \ \wedge \ R \cap W = \varnothing$
$\Rightarrow$
$(\!|S|\!) = \{R\} \times \{W\} \times \{W^a\} \times f$

Propriété 5 *La sémantique complète est définie complètement par la sémantique amnésique. Cette propriété dérive directement de la définition de la sémantique complète.*
pour toutes substitutions S et T, $[\![S]\!] = [\![T]\!] \Rightarrow (\!|S|\!) = (\!|T|\!)$

12.1.4 Sémantique d'un modèle BHDL

La sémantique d'un modèle BHDL D est constituée de la sémantique complète de la substitution principale du modèle BHDL ($Operation(D)$) et la sémantique de la substitution de l'initialisation $Init(D)$ qui définit les valeurs initiales des traces. Les variables locales sont cachées et les entrées (resp. les sorties) sont les variables listées dans la clause INPUTS (resp. la clause OUTPUTS).

Pour la sémantique d'un modèle BHDL, il n'y a pas de distinction entre sémantique amnésique ou sémantique complète puisque toutes les mémoires sont cachées par le modèle : toutes les mémoires sont des variables locales, une entrée ou une sortie ne peut pas être à la fois lue et écrite.

On notera que l'ensemble support en écriture de l'initialisation $Init(D)$ est nécessairement égal à l'ensemble des variables du modèles puisque toute variable doit être initialisée en B. Par ailleurs, si

l'ensemble support en lecture de $Operation(D)$ est vide, l'ensemble des entrées est nécessairement vide également. Ainsi, ci dessous nous notons $R \cap W_i$ pour être explicite alors que nous avons toujours $R \cap W_i = R$. Nous avons alors trois cas à prendre en compte.

- Cas général : si l'ensemble des entrées du modèle n'est pas vide, que l'ensemble support en lecture de $Operation(D)$ n'est pas vide et que son intersection avec l'ensemble support en écriture de $Init(D)$ n'est pas vide :
$\langle\!|D|\!\rangle = |\!|D|\!| = \{Input(D)\} \times \{Output(D)\} \times \{Output(D)\} \times f_d$,
où $f_d = \{x^\omega \mapsto y^\omega | \exists (R, W, W^a, W_i, W_i^a, X^\omega, Y^\omega, Z^\omega) \cdot ($
$(R, W, W^a, (X^\omega = \langle X_0, X_1, ...\rangle) \mapsto Y^\omega) \in \langle\!|Operation(D)|\!\rangle \wedge$
$(\varnothing, W_i, W_i^a, Z^\omega = \langle Z_0, Z_1, ...\rangle) \in |\!|Init(D)|\!| \wedge$
$select_{W_i \to R \cap W_i}(Z_0) = select_{R \to R \cap W_i}(X_0) \wedge$
$x^\omega = select_{R \to Input(D)}(X^\omega) \wedge$
$y^\omega = select_{W \to Output(D)}(Y^\omega)$
$)\}$

- Si l'ensemble des entrées du modèle est vide mais l'ensemble support en lecture de $Operation(D)$ n'est pas vide (le circuit n'a pas d'entrée mais possède des registres internes) :
$\langle\!|D|\!\rangle = |\!|D|\!| = \{\varnothing\} \times \{Output(D)\} \times \{Output(D)\} \times f_d$,
où $f_d = \{y^\omega | \exists (R, W, W^a, W_i, W_i^a, X^\omega, Y^\omega, Z^\omega) \cdot ($
$(R, W, W^a, (X^\omega = \langle X_0, X_1, ...\rangle) \mapsto Y^\omega) \in \langle\!|Operation(D)|\!\rangle \wedge$
$(\varnothing, W_i, W_i^a, Z^\omega = \langle Z_0, Z_1, ...\rangle) \in |\!|Init(D)|\!| \wedge$
$select_{W_i \to R \cap W_i}(Z_0) = select_{R \to R \cap W_i}(X_0) \wedge$
$y^\omega = select_{W \to Output(D)}(Y^\omega)$
$)\}$

- Si l'ensemble des entrées est vide et que l'ensemble support en lecture de $Operation(D)$ est vide (le circuit n'a pas d'entrée et ne possède pas de registre interne : c'est un circuit qui délivre une sortie constante) : $\langle\!|D|\!\rangle = |\!|D|\!| = \{\varnothing\} \times \{Output(D)\} \times \{Output(D)\} \times f_d$,
où $f_d = \{y^\omega | \exists (W, W^a, Y^\omega, Z^\omega) \cdot ($
$(\varnothing, W, W^a, Y^\omega) \in \langle\!|Operation(D)|\!\rangle \wedge$
$y^\omega = select_{W \to Output(D)}(Y^\omega)$
$)\}$

On pourrait demander pourquoi nous définissons une sémantique amnésique sur le traces et pourquoi nous n'utilisons pas directement la relation entre les entrées et les sorties, sans utiliser les traces, pour définir la sémantique complète. La sémantique amnésique sur les traces est nécessaire pour avoir une sémantique compositionnelle lorsque nous ajoutons les blocs sous-modèles et la réutilisation de modèles dans le langage. Nous verrons que lorsqu'un modèle est importé, la sémantique est définie en utilisant la sémantique complète (qui ne peut être définie qu'avec des traces à cause des mémoires) et que le résultat de l'importation est une sémantique amnésique. Nous verrons également que lorsque nous ajoutons l'importation de modèle et les blocs sous-modèles au langage, la sémantique amnésique reste compositionnelle, et pas la sémantique complète.

12.1.5 Compositionnalité de la sémantique amnésique

Pour prouver la compositionnalité de la sémantique amnésique, nous prouvons d'abord la compositionnalité de la relation $rel^\omega_{R \times W}$ (lemme 1) qui est déduite de la compositionnalité de la relation rel. A partir de là, on prouve la compositionnalité de la sémantique amnésique (propriété 6) en utilisant la compositionnalité des ensembles supports.

Propriété 6 (Compositionnalité de la sémantique amnésique) *La sémantique amnésique du langage BHDL sans bloc sous-modèle ni importation de modèles est compositionnelle.*

Pour toute composition $\star$ (la notation $\star$ dénote une des compositions ; ou $\|$ ou $\mathcal{C}_\mathcal{P}$), nous prouvons (lemme 1) qu'il existe un opérateur $\circ_\star$ tel que :

$(\!|S \star T|\!) =$

$$\{R_{S\star T}\} \times \{W_{S\star T}\} \times \{W^a_{S\star T}\} \times (\mathcal{F}_S, rel^\omega_{R_S \times W_S}(S)) \circ_\star (\mathcal{F}_T, rel^\omega_{R_T \times W_T}(T))$$

Puisque tous les ensembles supports sont connus à partir des sémantiques amnésiques des deux substitutions composées, nous pouvons calculer directement les ensembles supports (grâce à la compositionnalité des ensembles supports) de la composition.

- $R_S = \rho((\!|S|\!))$, $W_S = \phi((\!|S|\!))$, $W^a_S = \psi((\!|S|\!))$,
- $R_T = \rho((\!|T|\!))$, $W_T = \phi((\!|T|\!))$, $W^a_T = \psi((\!|T|\!))$,
- $\mathcal{F}_S = (\rho((\!|S|\!)), \phi((\!|S|\!)), \psi((\!|S|\!)))$,
- $\mathcal{F}_T = (\rho((\!|T|\!)), \phi((\!|T|\!)), \psi((\!|T|\!)))$,
- $rel^\omega_{R_S \times W_S}(S) = \delta((\!|S|\!))$,
- $rel^\omega_{R_T \times W_T}(T) = \delta((\!|T|\!))$,

De la même façon on obtient, à partir des deux sémantiques, les deux relations $rel^\omega_{R\times W}$. Nous devons prouver que pour toute composition $\star$ il existe un opérateur $\diamond_\star$ tel que :

$$(\!|S \star T|\!) = (\!|S|\!) \diamond_\star (\!|T|\!)$$

Pour cela, nous prouvons la compositionnalité de $rel^\omega_{R\times W}$ (lemme 1) en prouvant d'abord la compositionnalité de rel, puis celle de rel^ω et enfin celle de $rel^\omega_{R\times W}$.

Lemme 1 (Compositionnalité de $rel^\omega_{R\times W}$) *Pour toute composition $\star$ ($\star$ est ; ou $\|$ ou $\mathcal{C}_\mathcal{P}$), il existe un opérateur $\circ_\star$ tel que*

$(\!|S \star T|\!) = \{R_{S\star T}\} \times \{W_{S\star T}\} \times \{W^a_{S\star T}\} \times (\mathcal{F}_S, rel^\omega_{R_S \times W_S}(S)) \circ_\star (\mathcal{F}_T, rel^\omega_{R_T \times W_T}(T))$

La compositionnalité de la sémantique amnésique provient de la compositionnalité de la relation rel. Nous écrivons ci-dessous, pour chaque composition $\star$, la décomposition de $rel(S \star T)$ en terme de $rel(S)$ et $rel(T)$. Ensuite, nous montrons comment cette compositionnalité se propage jusqu'à la relation rel^ω. Finalement, nous exprimons la compositionnalité de $rel^\omega_{R\times W}$ en (1) utilisant la définition de $rel^\omega_{R_{S\star T} \times W_{S\star T}}(S \star T)$ à partir de $rel^\omega(S \star T)$, (2) en décomposant $rel^\omega(S \star T)$ en termes de $rel^\omega(S)$ et $rel^\omega(T)$, et (3) en utilisant la définition de rel^ω en termes de $rel^\omega_{R\times S}$ (propriété 4) pour retrouver $rel^\omega_{R_S \times W_S}(S)$ et $rel^\omega_{R_T \times W_T}(T)$.

Nous ne montrons ici les preuves que dans le cas général. Les cas où des ensembles supports en lecture peuvent être vides sont des cas particuliers dont la preuve se fait de façon semblable.

Compositionnalité de *rel*

Nous ne rappelons pas ici la définition de rel (cf. section 9.1.1). Nous montrons comment cette relation est compositionnelle pour les trois compositions suivantes : composition séquentielle, composition parallèle et composition conditionnelle. Ces résultats sont déjà connus, nous ne donnons pas les preuves formelles ici (preuves basées sur la manipulation des ensembles), mais nous récrivons les décompositions en utilisant nos notations. On notera qu'en BHDL toutes les substitutions sont "faisables", ainsi on arrive à certaines simplifications en utilisant cette hypothèse (s'exprimant par l'égalité $\overline{pre(S)} = \varnothing$). Bien que nous ne manipulions pas ici des traces mais de simples valeurs, nous utilisons certains opérateurs définis sur les traces : nous considérons les simples valeurs comme des traces de longueur 1.

▷ Composition séquentielle

L'expression ci-dessous définissant la relation rel pour la composition séquentielle d'une substitution S et d'une substitution T ($S;T$) s'interprète de cette façon : la relation de la substitution résultant de la composition lie x et y si S accepte x en entrée (x est dans le domaine de rel(S)) pour produire

une valeur u ($x \mapsto u \in rel(S)$) qui est acceptée par T qui produit y à partir de u ($u \mapsto y \in rel(T)$). Cette interprétation est faite pour le cas où les deux substitutions sont déterministes. Dans le cas non déterministe, il faut ajouter à l'interprétation des "il existe" : $x \mapsto y$ est dans $rel(S;T)$ s'il existe une valeur u telle que $x \mapsto u \in rel(S)$ et $u \mapsto y \in rel(T)$. À noter qu'en BHDL, le cas non déterministe ne peut intervenir que dans la clause INITIALISATION qui ne contient que des compositions parallèles.

$$rel(S;T) = \{x \mapsto y | \exists u \cdot (x \mapsto u \in rel(S) \wedge u \mapsto y \in rel(T))\} \quad (\text{hypothèse} : \overline{pre(S)} = \varnothing)$$

▷ **Composition parallèle**

Rappelons que dans une composition parallèle, les ensembles supports des deux substitutions sont nécessairement disjoints (condition de bonne formation d'une substitution B). Ainsi, les deux relations définissant les substitutions ont des sorties différentes, elles peuvent éventuellement avoir des entrées communes. La relation résultant de la composition parallèle est alors constituée de l'union des deux relations, en faisant attention aux entrées communes.

$$\begin{aligned} rel(S\|T) = \{X \mapsto Y | \exists (Y_S, Y_T) \cdot (& \\ & X \mapsto Y_S \in rel(S) \wedge X \mapsto Y_T \in rel(T)) \wedge \\ & select_{V \to W_S}(Y) = select_{V \to W_S}(Y_S) \wedge \\ & select_{V \to W_T}(Y) = select_{V \to W_T}(Y_T) \wedge \\ & select_{V \to V-(W_S \cup W_T)}(Y) = select_{V \to V-(W_S \cup W_T)}(X) \\ &)\} \end{aligned}$$

▷ **Composition conditionnelle**

La définition de rel dans la méthode B conduit à la définition suivante de la relation d'une composition conditionnelle.

$$rel(A \; \mathcal{C}_P \; B) = ((set(P) \times id) \cap rel(A)) \cup ((\overline{set(P)} \times id) \cap rel(B))$$

Pour des manipulations plus simples dans le reste de ce document, nous utilisons nos propres notations, nous réécrivons $rel(A \; \mathcal{C}_P \; B)$ de la façon suivante :

$$\begin{aligned} rel(A \; \mathcal{C}_P \; B) = \{X \mapsto Y | \exists (Y_A, Y_B) \cdot (& \\ & X \mapsto Y_A \in rel(A) \wedge X \mapsto Y_B \in rel(B) \wedge Y = merge_{X/P}(Y_A, Y_B))\} \end{aligned}$$

La relation résultant de la composition conditionnelle est un mélange des deux relations. Lorsque la condition est évaluée à *vraie*, c'est le résultat de la première substitution qui est choisi, sinon c'est le résultat de la seconde substitution.

Compositionnalité de rel^ω

Nous montrons ci-dessous comment la relation rel^ω est compositionnelle pour la composition séquentielle, la composition parallèle et la composition conditionnelle. Ces égalités proviennent directement de l'application de la définition à partir de la relation rel et des propriétés de compositionnalité de la relation rel définies ci-dessus. Puisque nous avions déjà utilisé les opérateurs sur les traces pour la compositionnalité de rel, nous obtenons des expressions très proches de celles déjà obtenues. Leur interprétation reste la même.

▷ **Composition séquentielle**

$$rel^\omega(S;T) = \{x^\omega \mapsto y^\omega | \exists u^\omega \cdot (u^\omega \in d^\omega(V) \wedge x^\omega \mapsto u^\omega \in rel^\omega(S) \wedge u^\omega \mapsto y^\omega \in rel^\omega(T))\}$$

▷ **Composition parallèle**

$$\begin{aligned} rel^\omega(S\|T) = \{X^\omega \mapsto Y^\omega | \exists (Y_S^\omega, Y_T^\omega) \cdot (& \\ & X^\omega \mapsto Y_S^\omega \in rel^\omega(S) \\ & X^\omega \mapsto Y_T^\omega \in rel^\omega(T) \\ & select_{V \to W_S}(Y^\omega) = select_{V \to W_S}(Y_S^\omega) \wedge \\ & select_{V \to W_T}(Y^\omega) = select_{V \to W_T}(Y_T^\omega) \wedge \\ & select_{V \to V-(W_S \cup W_T)}(Y^\omega) = select_{V \to V-(W_S \cup W_T)}(X^\omega) \\ &)\} \end{aligned}$$

▷ **Composition conditionnelle**

$$rel^{\omega}(A\ \mathcal{C}_P\ B) = \{X^{\omega} \mapsto Y^{\omega} | \exists (Y_A^{\omega}, Y_B^{\omega}) \cdot (X^{\omega} \mapsto Y_A^{\omega} \in rel^{\omega}(A) \wedge X^{\omega} \mapsto Y_B^{\omega} \in rel^{\omega}(B) \wedge Y^{\omega} = merge_{X/P}(Y_A^{\omega}, Y_B^{\omega}))\}$$

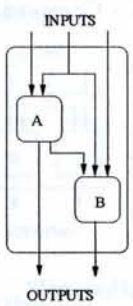
Compositionnalité de $rel^{\omega}_{R \times W}$

La compositionnalité de $rel^{\omega}_{R \times W}$ provient de la compositionnalité de rel^{ω} et de celle des ensembles supports. Pour prouver la compositionnalité de $rel^{\omega}_{R \times W}$ nous développons sa définition dans les trois cas de la composition séquentielle, composition parallèle et composition conditionnelle. Le principe de la preuve est ensuite le même dans les trois cas : le développement de la définition fait apparaître la relation rel^{ω} qui est remplacée par sa définition en termes de $rel^{\omega}_{R \times W}$ (propriété 4). De cette façon nous obtenons une définition de $rel^{\omega}_{R_{S \star T} \times W_{S \star T}}(S \star T)$ en fonction de $rel^{\omega}_{R_S \times W_S}(S)$ et $rel^{\omega}_{R_T \times W_T}(T)$. Ces deux dernières relations permettent de connaître les ensembles supports des deux substitutions qui sont composées, les ensembles supports ($R_{S \star T}$, $W_{S \star T}$, $W^a_{S \star T}$,) de la composition peuvent donc être calculés (compositionnalité des ensembles supports). Nous ne redonnons pas ici comment effectuer le calcul des ensembles supports (cf. tableau 9.1 page 182). À partir de ce résultat, nous pouvons prouver la compositionnalité de la sémantique amnésique (propriété 6).

▷ **Composition séquentielle**

La figure 12.2 montre une représentation graphique du circuit qui est modélisé par la composition séquentielle, les deux substitutions composées peuvent avoir des entrées communes, certaines entrées sont spécifiques et la deuxième substitution peut prendre en entrée des sorties de la première substitution. Une sortie provient d'une des deux substitutions mais jamais des deux (si elles ont des sorties communes, la sorties de la première substitution est écrasée par la deuxième substitution).

FIG. 12.2 – Composition séquentielle



Nous développons la définition de la relation $rel^{\omega}_{R_{S;T} \times W_{S;T}}(S;T)$ en fonction de $rel^{\omega}(S)$ et $rel^{\omega}(T)$ (définition 18).

$$\begin{aligned} rel^{\omega}_{R_{S;T} \times W_{S;T}}(S;T) = \{x^{\omega} \mapsto y^{\omega} | x^{\omega} \in d^{\omega}(R_{S;T}) \wedge y^{\omega} \in d^{\omega}(W_{S;T}) \wedge \exists (X^{\omega}, Y^{\omega}, U^{\omega}) \cdot (\\ X^{\omega} \in d^{\omega}(V) \wedge Y^{\omega} \in d^{\omega}(V) \wedge U^{\omega} \in d^{\omega}(V) \wedge \\ X^{\omega} \mapsto U^{\omega} \in rel^{\omega}(S) \wedge \\ U^{\omega} \mapsto Y^{\omega} \in rel^{\omega}(T) \wedge \\ x^{\omega} = select_{V \to R_{S;T}}(X^{\omega}) \wedge \\ y^{\omega} = select_{V \to W_{S;T}}(Y^{\omega}) \\ )\} \end{aligned}$$

En développant la définition rel^{ω} en fonction de $rel^{\omega}_{R_{S;T} \times W_{S;T}}$ (propriété 4), nous obtenons l'égalité suivante :

$$\begin{aligned} rel^{\omega}_{R_{S;T} \times W_{S;T}}(S;T) = \\ \{x^{\omega} \mapsto y^{\omega} | x^{\omega} \in d^{\omega}(R_{S;T}) \wedge y^{\omega} \in d^{\omega}(W_{S;T}) \wedge \exists (X^{\omega}, Y^{\omega}, U^{\omega}, x_S^{\omega}, x_S^{\omega}, x_T^{\omega}, x_T^{\omega}) \cdot (\\ X^{\omega} \in d^{\omega}(V) \wedge Y^{\omega} \in d^{\omega}(V) \wedge \end{aligned}$$

$$
\begin{aligned}
&select_{V\to V-W_S}(U^\omega) = select_{V\to V-W_S}(X^\omega)\\
&x_S^\omega \mapsto y_S^\omega \in \boldsymbol{rel^\omega_{R_S\times W_S}(S)}\\
&x_S^\omega = select_{V\to R_S}(X^\omega)\\
&y_S^\omega = select_{V\to W_S}(U^\omega)\\
&select_{V\to V-W_T}(Y^\omega) = select_{V\to V-W_T}(U^\omega)\\
&x_T^\omega \mapsto y_T^\omega \in \boldsymbol{rel^\omega_{R_T\times W_T}(T)}\\
&x_T^\omega = select_{V\to R_T}(U^\omega)\\
&y_T^\omega = select_{V\to W_T}(Y^\omega)\\
&x^\omega = select_{V\to R_{S;T}}(X^\omega)\wedge\\
&y^\omega = select_{V\to W_{S;T}}(Y^\omega)\\
&)\}
\end{aligned}
$$

où les ensembles supports $R_{S;T}$ et $W_{S;T}$ sont définis à partir des ensembles supports R_S, W_S, R_T, W_T et W_S^a. On note par $\circ_;$ l'opérateur défini par l'égalité ci-dessus qui construit $rel^\omega_{R_{S;T}\times W_{S;T}}(S;T)$ à partir de $rel^\omega_{R_S\times W_S(S)}$ et $rel^\omega_{R_T\times W_T}(T)$. Par simplicité, nous notons par $\mathcal{F}_S$ les trois ensembles supports R_S, W_S et W_S^a (de même $\mathcal{F}_T$ les ensembles supports de T). Nous pouvons réécrire l'égalité précédente de cette façon :

$$rel^\omega_{R_{S;T}\times W_{S;T}}(S;T) = \circ_;(F_S, rel^\omega_{R_S\times W_S}(S), F_T, rel^\omega_{R_T\times W_T}(T))$$

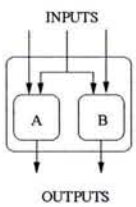
Par définition, nous avons $\{\!|S;T|\!\} = \{R_{S;T}\}\times\{W_{S;T}\}\times\{W^a_{S;T}\}\times rel^\omega_{R_{S;T}\times W_{S;T}}(S;T)$, qui peut se réécrire ainsi :

$$\{\!|S;T|\!\} = \{R_{S;T}\}\times\{W_{S;T}\}\times\{W^a_{S;T}\}\times(\mathcal{F}_S, rel^\omega_{R_S\times W_S}(S))\circ_;(\mathcal{F}_T, rel^\omega_{R_T\times W_T}(T))$$

▷ **Composition parallèle**

La figure 12.3 montre une représentation graphique du circuit qui est modélisé par la composition parallèle. Les deux substitutions composées en parallèle ont nécessairement des sorties distinctes mais peuvent avoir des entrées communes. Il n'y a pas de communication entre les deux substitutions.

FIG. 12.3 – Composition parallèle



Nous développons la définition de la relation $rel^\omega_{R_{S\|T}\times W_{S;T}}(S;T)$ en fonction de $rel^\omega(S)$ et $rel^\omega(T)$ (définition 18).

$$
\begin{aligned}
rel^\omega_{R_{S\|T}\times W_{S\|T}}(S\|T) = \{x^\omega \mapsto y^\omega | \exists(X^\omega, Y^\omega, Y^\omega_S, Y^\omega_T)\cdot(\\
&X^\omega \mapsto Y^\omega_S \in rel^\omega(S)\wedge\\
&X^\omega \mapsto Y^\omega_T \in rel^\omega(T)\wedge\\
&select_{V\to W_S}(Y^\omega) = select_{V\to W_S}(Y^\omega_S)\wedge\\
&select_{V\to W_T}(Y^\omega) = select_{V\to W_T}(Y^\omega_T)\wedge\\
&select_{V\to V-(W_S\cup W_T)}(Y^\omega) = select_{V\to V-(W_S\cup W_T)}(X^\omega)\wedge\\
&x^\omega = select_{V\to W_{S\|T}}(X^\omega)\wedge\\
&y^\omega = select_{V\to W_{S\|T}}(Y^\omega)\\
&)\}
\end{aligned}
$$

En développant la définition rel^ω en fonction de $rel^\omega_{R_{S;T}\times W_{S;T}}$ (propriété 4), nous obtenons l'égalité suivante :

$$
\begin{aligned}
rel^\omega_{R_{S\|T}\times W_{S\|T}}(S\|T) = \{x^\omega \mapsto y^\omega | \exists(X^\omega, Y^\omega, Y^\omega_S, Y^\omega_T, x^\omega_S, y^\omega_S, x^\omega_T, y^\omega_T)\cdot(\\
&\boldsymbol{x^\omega_S \mapsto y^\omega_S \in rel^\omega_{R_S\times W_S}(S)}\wedge\\
&select_{V\to V-W_S}(Y^\omega_S) = select_{V\to V-W_S}(X^\omega)\wedge
\end{aligned}
$$

$$
\begin{array}{l}
x_S^\omega = select_{V\to R_S}(X^\omega)\wedge \\
y_S^\omega = select_{V\to W_S}(Y_S^\omega)\wedge \\
x_T^\omega \mapsto y_T^\omega \in \boldsymbol{rel^\omega_{R_T\times W_T}(T)}\wedge \\
select_{V\to V-W_T}(Y_T^\omega) = select_{V\to V-W_T}(X^\omega)\wedge \\
x_T^\omega = select_{V\to R_T}(X^\omega)\wedge \\
y_T^\omega = select_{V\to W_T}(Y_T^\omega\wedge \\
select_{V\to W_S}(Y^\omega) = select_{V\to W_S}(Y_S^\omega)\wedge \\
select_{V\to W_T}(Y^\omega) = select_{V\to W_T}(Y_T^\omega)\wedge \\
select_{V\to V-(W_S\cup W_T)}(Y^\omega) = select_{V\to V-(W_S\cup W_T)}(X^\omega)\wedge \\
x^\omega = select_{V\to W_{S\|T}}(X^\omega)\wedge \\
y^\omega = select_{V\to W_{S\|T}}(Y^\omega) \\
)\}
\end{array}
$$

où les ensembles supports $R_{S\|T}$ et $W_{S\|T}$ sont définis à partir des ensembles supports R_S, W_S, R_T, W_T et W_S^a. On note par $\circ_\|$ l'opérateur défini par l'égalité ci-dessus qui construit $rel^\omega_{R_{S\|T}\times W_{S\|T}}(S\|T)$ à partir de $rel^\omega_{R_S\times W_S(S)}$ et $rel^\omega_{R_T\times W_T}(T)$. Par simplicité, nous notons par $\mathcal{F}_S$ les trois ensembles supports R_S, W_S et W_S^a (de même $\mathcal{F}_T$ les ensembles supports de T). Nous pouvons réécrire l'égalité précédente de cette façon :

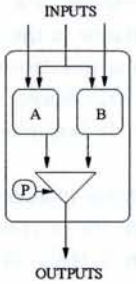
$$rel^\omega_{R_{S\|T}\times W_{S\|T}}(S\|T) = \circ_\|(F_S, rel^\omega_{R_S\times W_S}(S), F_T, rel^\omega_{R_T\times W_T}(T))$$

Par définition, nous avons $(\!|S\|T|\!) = \{R_{S\|T}\}\times\{W_{S\|T}\}\times\{W^a_{S\|T}\}\times rel^\omega_{R_{S\|T}\times W_{S\|T}}(S\|T)$, qui peut se récrire ainsi :

$$(\!|S\|T|\!) = \{R_{S\|T}\}\times\{W_{S\|T}\}\times\{W^a_{S\|T}\}\times(\mathcal{F}_S, rel^\omega_{R_S\times W_S}(S))\circ_\|(\mathcal{F}_T, rel^\omega_{R_T\times W_T}(T))$$

▷ **Composition conditionnelle**

FIG. 12.4 – Composition conditionnelle



Nous développons la définition de la relation $rel^\omega_{R_A\,c_P\,B\times W_{R_A\,c_P\,B}}(A\,C_P\,B)$ en fonction de $rel^\omega(A)$ et $rel^\omega(B)$ (définition 18).

$rel^\omega(A\ C_P\ B) = \{X^\omega \mapsto Y^\omega | \exists(Y_A^\omega, Y_B^\omega)\cdot(X^\omega \mapsto Y_A^\omega \in rel^\omega(A)\wedge X^\omega \mapsto Y_B^\omega \in rel^\omega(B)\wedge Y^\omega = merge_{X/P}(Y_A^\omega, Y_B^\omega))\}$

$$
\begin{array}{l}
rel^\omega_{R_A\,c_P\,B\times W_A\,c_P\,B}(A\,C_P\,B) = \{x^\omega \mapsto y^\omega | \exists(Y_A^\omega, Y_B^\omega)\cdot(\\
\qquad X^\omega \mapsto Y_A \in rel^\omega(A)\wedge \\
\qquad X^\omega \mapsto Y_B \in rel^\omega(B)\wedge \\
\qquad Y^\omega = merge_{X/P}(Y_A^\omega, Y_B^\omega))\wedge \\
\qquad x^\omega = select_{V\to R_A\,c_P\,B}(X^\omega)\wedge \\
\qquad y^\omega = select_{V\to R_A\,c_P\,B}(Y^\omega) \\
\qquad)\}
\end{array}
$$

En développant la définition rel^ω en fonction de $rel^\omega_{R_A\,c_P\,B\times W_A\,c_P\,B}$ (propriété 4), nous obtenons l'égalité suivante :

$$
\begin{array}{l}
rel^\omega_{R_A\,c_P\,B\times W_A\,c_P\,B}(A\,C_P\,B) = \{x^\omega \mapsto y^\omega | \exists(X^\omega, Y_A^\omega, Y_B^\omega, x_A^\omega, y_A^\omega, x_B^\omega, y_B^\omega)\cdot(\\
\qquad x_A^\omega \mapsto y_A^\omega \in \boldsymbol{rel^\omega_{R_A\times W_A}(A)}\wedge \\
\qquad select_{V\to V-W_A}(Y_A^\omega) = select_{V\to V-W_A}(X^\omega)\wedge
\end{array}
$$

$$
\begin{array}{l}
x_A^\omega = select_{V\to R_A}(x^\omega)\wedge \\
y_A^\omega = select_{V\to W_A}(y^\omega)\wedge \\
x_B^\omega \mapsto y_B^\omega \in \boldsymbol{rel^\omega_{R_B\times W_B}(B)}\wedge \\
select_{V\to V-W_B}(Y_B^\omega) = select_{V\to V-W_B}(X^\omega)\wedge \\
x_B^\omega = select_{V\to R_B}(x^\omega)\wedge \\
y_B^\omega = select_{V\to W_B}(y^\omega)\wedge \\
Y^\omega = merge_{X/P}(Y_A^\omega, Y_B^\omega)\wedge \\
x^\omega = select_{V\to R_A\, c_P\, B}(X^\omega)\wedge \\
y^\omega = select_{V\to R_A\, c_P\, B}(Y^\omega) \\
)\}
\end{array}
$$

où les ensembles supports $R_{A\,C_P\,B}$ et $W_{A\,C_P\,B}$ sont définis à partir des ensembles supports R_A, W_A, R_B, W_B et W_A^a. Notons $\circ_{\mathcal{C}_P}$ l'opérateur défini par l'égalité ci-dessus qui construit $rel^\omega_{R_{A\,C_P\,B}\times W_{A\,C_P\,B}}(A\,C_P\,B)$ à partir de $rel^\omega_{R_A\times W_A(A)}$ et $rel^\omega_{R_B\times W_B}(B)$. Par simplicité, nous notons par $\mathcal{F}_A$ les trois ensembles supports R_A, W_A et W_S^A (de même $\mathcal{F}_B$ les ensembles supports de B). Nous pouvons réécrire l'égalité précédente de cette façon :

$$rel^\omega_{R_{A\,C_P\,B}\times W_{A\,C_P\,B}}(A\,C_P\,B) = (F_S, rel^\omega_{R_A\times W_A}(A)) \circ_{\mathcal{C}_P} (F_B, rel^\omega_{R_B\times W_B}(B))$$

Par définition, nous avons

$$\{\!|A\,C_P\,B|\!\} = \{R_{A\,C_P\,B}\} \times \{W_{A\,C_P\,B}\} \times \{W^a_{A\,C_P\,B}\} \times rel^\omega_{R_{A\,C_P\,B}\times W_{A\,C_P\,B}}(A\,C_P\,B)$$

qui peut se récrire ainsi :

$$\{\!|A\,C_P\,B|\!\} = \{R_{A\,C_P\,B}\} \times \{W_{A\,C_P\,B}\} \times \{W^a_{A\,C_P\,B}\} \times (\mathcal{F}_A, rel^\omega_{R_A\times W_A}(A)) \circ_{\mathcal{C}_P} (\mathcal{F}_B, rel^\omega_{R_B\times W_B}(B))$$

12.1.6 Compositionnalité de la sémantique complète

Nous prouvons la compositionnalité de la sémantique complète (rappelons que nous sommes dans le cas du langage BHDL sans importation ni bloc sous-modèle) en définissant un opérateur qui permet d'obtenir la sémantique amnésique à partir de la sémantique complète. Cet opérateur est l'inverse de l'opérateur qui définit la sémantique complète à partir de la sémantique amnésique (cf. définition 19). Nous expliquons ci-dessous comment retrouver la sémantique amnésique à partir de la sémantique complète. Le principe est de retrouver la relation $rel_{R\times W}$, à partir de laquelle on peut recalculer la sémantique amnésique en utilisant sa définition.

Pour retrouver la relation $rel_{R\times W}$, nous nous intéressons aux éléments initiaux des traces. L'élément initial d'une trace est le tout premier élément de la trace, l'élément d'indice 0. Soit $x^\omega \mapsto y^\omega$ une trace de la sémantique complète d'une substitution S ($(R_S, W_S, W_S^a, x^\omega \mapsto y^\omega) \in \{\!|S|\!\}$) : le couple $x_0 \mapsto y_0$ est l'*élément initial* de $x^\omega \mapsto y^\omega$.

La sémantique complète est un ensemble de traces. Nous prouvons que l'ensemble des éléments initiaux de ces traces est exactement la relation $rel_{R_S\times W_S}$. On peut d'abord comprendre cela intuitivement, nous donnons la preuve plus formelle dans la suite de cette section.

La sémantique amnésique d'une substitution S est la relation $rel_{R_S\times W_S}$ qui a été élevée au rang de relation sur des traces. C'est-à-dire que pour chaque trace $x^\omega \mapsto y^\omega$ et pour chaque rang k l'élément $x_k \mapsto y_k$ est un élément de $rel_{R_S\times W_S}$, et, finalement, la relation $rel_{R_S\times W_S}$ est exactement l'ensemble de ces éléments. En fait, la relation $rel_{R_S\times W_S}$ peut être entièrement retrouvée à partir du sous ensemble des éléments initiaux. Par ailleurs, la sémantique complète est la sémantique amnésique qui est restreinte de façon à ce que, pour toutes les variables qui sont à la fois des entrées et des sorties, les entrées au cycle $n+1$ doivent être égales aux sorties du cycle n. Ces restrictions n'affectent pas les éléments initiaux, ils sont donc les mêmes dans la sémantique complète et dans la sémantique amnésique. Ainsi, on peut retrouver la relation $rel_{R_S\times W_S}$ à partir des éléments initiaux de la sémantique complète.

Ce raisonnement est formalisé dans le reste de cette section. Nous définissons d'abord formellement l'ensemble des éléments initiaux. Nous montrons ensuite que la restriction qui définit la sémantique complète à partir de la sémantique amnésique n'affecte pas les éléments initiaux, propriété à partir de laquelle nous prouvons que les éléments initiaux des deux sémantiques sont les mêmes. Nous montrons ensuite comment la relation $rel_{R\times W}$ correspond exactement à l'ensemble des éléments initiaux. Nous

montrons finalement comment la sémantique amnésique est retrouvée à partir des éléments initiaux et enfin nous prouvons la compositionnalité de la sémantique complète.

Définition 20 (Éléments initiaux) *Pour toute relation r sur des traces, nous définissons $i(r)$, la relation entre les éléments initiaux de r. C'est-à-dire que pour tout élément $x^\omega \mapsto y^\omega$ de r, $i(r)$ contient $x_0 \mapsto y_0$.*

$$i(r) = \{a \mapsto b | \exists (x^\omega, y^\omega) \cdot (x^\omega \mapsto y^\omega \in r \wedge x_0 = a \wedge y_0 = b)\}$$

Propriété 7 (Éléments initiaux de l'intersection de deux relations) *Les éléments initiaux de l'intersection de deux relations est l'intersection des deux ensembles d'éléments initiaux. Pour toutes relations r et s sur les traces, nous avons :*

$$i(r \cap s) = i(r) \cap i(s)$$

La preuve se fait en développant les définitions :
$i(r \cap s) = \{a \mapsto b | \exists (x^\omega, y^\omega) \cdot (x^\omega \mapsto y^\omega \in r \cap s \wedge x_0 = a \wedge y_0 = b)\}$
$= \{a \mapsto b | \exists (x^\omega, y^\omega) \cdot (x^\omega \mapsto y^\omega \in r \wedge x^\omega \mapsto y^\omega \in s \wedge x_0 = a \wedge y_0 = b)\}$
$= \{a \mapsto b | \exists (x^\omega, y^\omega) \cdot (x^\omega \mapsto y^\omega \in r \wedge x_0 = a \wedge y_0 = b)\} \cap \{a \mapsto b | \exists (x^\omega, y^\omega) \cdot (x^\omega \mapsto y^\omega \in s \wedge x_0 = a \wedge y_0 = b)\}$
$= i(r) \cap i(s)$

Lemme 2 *Pour toute entrée a et toute sortie b, on peut trouver une trace d'entrée et une trace de sortie qui ont respectivement les valeurs a et b comme éléments initiaux et telles que les entrées et les sorties correspondant aux variables modélisant des mémoires (à la fois entrées et sorties) sont connectées (l'entrée au cycle $n+1$ est la sortie du cycle n). Formellement :*

$$\begin{aligned}\forall (a,b) \cdot (a \in d^\omega(R) \wedge b \in d^\omega(W) \Rightarrow \exists (x^\omega, y^\omega) \cdot (& \\ & x^\omega \in d^\omega(R) \wedge y^\omega \in d^\omega(W) \wedge \\ & select_{R \to R \cap W}(x^\omega_{+1}) = select_{W \to R \cap W}(y^{\omega \to length(x^\omega_{+1})}) \wedge \\ & x_0 = a \wedge y_0 = b)\end{aligned}$$

La preuve se fait en construisant les traces x^ω et y^ω à partir de a et b. D'abord, nous éclatons a et b en deux parties afin de séparer les variables qui sont à la fois des entrées *et* des sorties ($R \cap W$) et les variables qui sont seulement des entrées ($R - R \cap W$) ou seulement des sorties ($W - R \cap W$) : $a = (a_R, a_P)$ et $b = (b_W, b_P)$. Formellement, a_R, a_P, b_W et b_P sont définis de cette façon :
$a_R = select_{R \to R - R \cap W}(a)$ (variables de a qui sont seulement des entrées)
$a_P = select_{R \to R \cap W}(a)$ (variables de a qui sont à la fois des entrées et des sorties)
$b_W = select_{W \to W - R \cap W}(b)$ (variables de b qui sont seulement des sorties)
$b_P = select_{W \to R \cap W}(b)$ (variables de b qui sont à la fois des entrées et des sorties)
On peut construire, par exemple, x^ω et y^ω de cette façon :
$x^\omega = \langle a, (a_R, b_P), (a_R, b_P), (a_R, b_P), ... \rangle$, et
$y^\omega = \langle b, b, b, b, ... \rangle$.
Nous avons $x^\omega_{+1} = \langle (a_R, b_P), (a_R, b_P), (a_R, b_P), ... \rangle$. Ainsi,
$select_{R \to R \cap W}(x^\omega_{+1}) = \langle b_P, b_P, ... \rangle$, et,
$select_{W \to R \cap W}(y^{\omega \to length(x^\omega_{+1})}) = \langle b_P, b_P, ... \rangle$.

Propriété 8 (La restriction n'affecte pas les éléments initiaux) *La sémantique complète est définie à partir de la sémantique amnésique en la restreignant de sorte que pour toute trace $x^\omega \mapsto y^\omega$, pour les variables qui sont à la fois des entrées et des sorties ($R \cap W$), la valeur de sortie lors d'un cycle est prise comme entrée au cycle suivant ($select_{R \to R \cap W}(x^\omega_{+1}) = select_{W \to R \cap W}(y^{\omega \to length(x^\omega_{+1})})$). Nous montrons que cette restriction n'affecte pas les éléments initiaux, c'est-à-dire que les éléments initiaux de la relation sur les traces définie pas le prédicat de la restriction est l'ensemble contenant toutes les possibilités ($d(R) \times d(W)$) :*

$$\begin{aligned} i\left(\{x^\omega \mapsto y^\omega | x^\omega \in d^\omega(R) \wedge y^\omega \in d^\omega(W) \wedge select_{R \to R \cap W}(x^\omega_{+1}) = select_{W \to R \cap W}(y^{\omega \to length(x^\omega_{+1})})\}\right) & \\ = d(R) \times d(W) & \end{aligned}$$

La preuve se fait appliquant la définition de $i(r)$ sur la relation définie par le prédicat de restriction et en utilisant le lemme 2 énoncé plus haut.

$i(\{x^\omega \mapsto y^\omega | x^\omega \in d^\omega(R) \wedge y^\omega \in d^\omega(W) \wedge select_{R\to R\cap W}(x^\omega_{+1}) = select_{W\to R\cap W}(y^{\omega\to length(x^\omega_{+1}))})\})$

$= \{a \mapsto b | a \in d(R) \wedge b \in d(W) \wedge \exists(x^\omega, y^\omega) \cdot (x^\omega \in d^\omega(R) \wedge y^\omega \in d^\omega(W) \wedge select_{R\to R\cap W}(x^\omega_{+1}) = select_{W\to R\cap W}(y^{\omega\to length(x^\omega_{+1})}) \wedge x_0 = a \wedge y_0 = b)\}$

$=_{lemme2} d(R) \times d(W)$

Propriété 9 (Égalité des éléments initiaux des deux sémantiques) *La sémantique complète a les mêmes éléments initiaux que la sémantique amnésique.*

$$i\left(\begin{array}{l} rel^\omega_{R_S\times W_S}(S) \cap \\ \quad \{x^\omega \mapsto y^\omega | x^\omega \in d^\omega(R) \wedge y^\omega \in d^\omega(W) \wedge select_{R\to R\cap W}(x^\omega_{+1}) = select_{W\to R\cap W}(y^\omega)\} \end{array}\right) = i(rel^\omega_{R_S\times W_S}(S))$$

La preuve découle directement de la propriété 8 :

$$\begin{array}{l} i\left(\begin{array}{l} rel^\omega_{R_S\times W_S}(S) \cap \\ \quad \{x^\omega \mapsto y^\omega | x^\omega \in d^\omega(R) \wedge y^\omega \in d^\omega(W) \wedge select_{R\to R\cap W}(x^\omega_{+1}) = select_{W\to R\cap W}(y^\omega)\} \end{array}\right) \\ =_{prop7} i(rel^\omega_{R_S\times W_S}(S)) \cap \\ \qquad\qquad i(\{x^\omega \mapsto y^\omega | x^\omega \in d^\omega(R) \wedge y^\omega \in d^\omega(W) \wedge select_{R\to R\cap W}(x^\omega_{+1}) = select_{W\to R\cap W}(y^\omega)\}) \\ =_{prop8} i(rel^\omega_{R_S\times W_S(S)}) \cap d(R_S) \times d(W_S) \\ = i(rel^\omega_{R_S\times W_S}(S)) \end{array}$$

Lemme 3 *On peut construire une trace formée uniquement d'éléments d'une relation $rel_{R_S\times W_S}$ à partir d'un élément de cette relation.*

$$\exists(x^\omega, y^\omega) \cdot (x^\omega \in d^\omega(R_S) \wedge y^\omega \in d^\omega(W_S) \wedge \forall k \cdot (k \geq 0 \Rightarrow x_k \mapsto y_k \in rel_{R_S\times W_S}(S)) \wedge x_0 = a \wedge y_0 = b) \Leftrightarrow a \mapsto b \in rel_{R_S\times W_S}(S)$$

Cette propriété est relativement évidente mais nous en avons besoin par la suite, la preuve se fait en montrant chaque implication :

- $\Rightarrow$

  Nous avons $\forall k \cdot (k \geq 0 \Rightarrow x_k \mapsto y_k \in rel_{R_S\times W_S}(S)) \Rightarrow x_0 \mapsto y_0 \in rel_{R_S\times W_S}(S))$

  On en déduit $\forall k \cdot (k \geq 0 \Rightarrow x_k \mapsto y_k \in rel_{R_S\times W_S}(S)) \wedge x_0 = a \wedge y_0 = b) \Rightarrow a \mapsto b \in rel_{R_S\times W_S}(S))$

- $\Leftarrow$

  Nous avons $a \mapsto b \in rel_{R_S\times W_S}(S)$

  Par définition de $rel_{R_S\times W_S}$ nous avons $a \in d(R_S) \wedge b \in d(W_S)$

  Nous construisons les deux traces $a^\omega = \langle a, a, a, ...\rangle$ et $b^\omega = \langle b, b, b, ...\rangle$ qui ont la propriété $a^\omega \in d^\omega(R_S) \wedge b^\omega \in d^\omega(W_S) \wedge \forall k \cdot (k \geq 0 \Rightarrow a_k \mapsto b_k \in rel_{R_S\times W_S}(S)) \wedge a_0 = a \wedge b_0 = b$

Propriété 10 (Les éléments initiaux forment la relation $rel_{R\times W}$) *L'ensemble des éléments initiaux de la relation sur les traces de la sémantique amnésique d'une substitution S est exactement la relation $rel_{R_S\times W_S}(S)$.*

$$i(rel^\omega_{R_S\times W_S}(S)) = rel_{R_S\times W_S}(S)$$

$i(rel^\omega_{R_S\times W_S}(S))$

$= i(\{x^\omega \mapsto y^\omega | x^\omega \in d^\omega(R_S) \wedge y^\omega \in d(W_S) \wedge \forall k \cdot (k \geq 0 \Rightarrow x_k \mapsto y_k \in rel_{R_S\times W_S}(S))\})$

$= \{a \mapsto b | \exists(x^\omega, y^\omega) \cdot (x^\omega \in d(R_S) \wedge y^\omega \in d^\omega(W_S) \wedge \forall k \cdot (k \geq 0 \Rightarrow x_k \mapsto y_k \in rel_{R_S\times W_S}(S)) \wedge x_0 = a \wedge y_0 = b)\}$

$=_{lemme3} rel_{R_S\times W_S}(S)$

Définition 21 (Restauration de la sémantique à partir des éléments initiaux) *A partir de l'ensemble des éléments initiaux de la sémantique complète on peut construire une nouvelle sémantique qui est la relation sur des traces résultant de la combinaison de tous les éléments initiaux.*

$Rev(\{R\} \times \{W\} \times \{W^a\} \times f)$

$= \{R\} \times \{W\} \times \{W^a\} \times \{x^\omega \mapsto y^\omega | \forall k \cdot (k \geq 0 \Rightarrow x_k \mapsto y_k \in i(f))\}$

Propriété 11 (La sémantique restaurée est la sémantique amnésique) *La sémantique obtenue par la combinaison des éléments initiaux de la sémantique complète est exactement la sémantique amnésique.*

$$(\!|S|\!) = Rev((\!|S|\!))$$

$Rev((\!|S|\!)) = Rev(\{R_S\} \times \{W_S\} \times \{W_S^a\} \times (rel^\omega_{R_S \times W_S} \cap \{x^\omega \mapsto y^\omega | x^\omega \in d^\omega(R_S) \wedge y^\omega \in d^\omega(W_S) \wedge select_{R_S \to R_S \cap W_S}(x^\omega_{+1}) = select_{W_S \to R_S \cap W_S}(y^\omega)\}))$
$= \{R_S\} \times \{W_S\} \times \{W_S^a\} \times \{x^\omega \mapsto y^\omega | \forall k \cdot (k \geq 0 \Rightarrow x_k \mapsto y_k \in i(rel^\omega_{R_S \times W_S} \cap \{x^\omega \mapsto y^\omega | x^\omega \in d^\omega(R_S) \wedge y^\omega \in d^\omega(W_S) \wedge select_{R_S \to R_S \cap W_S}(x^\omega_{+1}) = select_{W_S \to R_S \cap W_S}(y^\omega)\}))\}$
$=_{property 9} \{R_S\} \times \{W_S\} \times \{W_S^a\} \times \{x^\omega \mapsto y^\omega | \forall k \cdot (k \geq 0 \Rightarrow x_k \mapsto y_k \in i(rel^\omega_{R_S \times W_S})\}$
$=_{property 10} \{R_S\} \times \{W_S\} \times \{W_S^a\} \times \{x^\omega \mapsto y^\omega | \forall k \cdot (k \geq 0 \Rightarrow x_k \mapsto y_k \in rel_{R_S \times W_S}\}$
$= \{R_S\} \times \{W_S\} \times \{W_S^a\} \times rel^\omega_{R_S \times W_S}$
$= (\!|S|\!)$

Propriété 12 (Compositionnalité de la sémantique complète) *La sémantique complète du langage BHDL sans importation ni bloc sous-modèle est compositionnelle.*

Cette propriété est déduite de la compositionnalité de la sémantique amnésique et de la propriété 11. Pour tout opérateur de composition $\star$, la sémantique amnésique est associée à un opérateur $\diamond_\star$ qui donne la sémantique amnésique de la composition à partir des sémantiques amnésiques des deux substitutions composées : $(\!|S \star T|\!) = (\!|S|\!) \diamond_\star (\!|T|\!)$.

La sémantique complète est définie à partir de la sémantique amnésique, nous réduisons l'expression de la définition en disant qu'il existe un opérateur f qui construit la sémantique complète à partir de la sémantique amnésique.

$$\begin{aligned}(\!|S \star T|\!) &= f((\!|S \star T|\!)) \\ &=_{property\ 6} f((\!|S|\!) \diamond_\star (\!|T|\!)) \\ &=_{property\ 11} f(Rev((\!|S|\!)) \diamond_\star Rev((\!|T|\!))) \\ &=_{def} (\!|S|\!) \diamond_{\star f} (\!|T|\!)\end{aligned}$$

Ainsi, la sémantique complète associe à chaque opérateur de composition $\star$ un opérateur $\diamond_{\star f}$ qui donne la sémantique complète de la composition à partir des sémantiques complètes des deux substitutions composées.

12.2 Ajout des blocs sous-modèles

La sémantique amnésique d'un bloc sous-modèle est définie en utilisant la sémantique amnésique de la substitution de la clause INITIALISATION, et la sémantique complète de la substitution principale. Notons qu'en fait la sémantique amnésique et la sémantique complète de la clause INITIALISATION sont les mêmes puisque les substitutions qui induisent des mémoires ne sont pas admises dans cette clause.

Définition 22 (Sémantique amnésique d'un bloc sous-modèle)

$(\!|\mathcal{B}_v(I,S)|\!) = \{R_S - v\} \times \{W_S - v\} \times \{W_S^a - v\} \times rel^\omega_{R_{\mathcal{B}_v(I,S)} \times W_{\mathcal{B}(v,I,S)}}(\mathcal{B}_v(I,S))$

$rel^\omega_{R_{\mathcal{B}_v(I,S)} \times W_{\mathcal{B}_v(I,S)}}(\mathcal{B}_v(I,S))$
$= \{x^\omega \mapsto y^\omega | x^\omega \in d^\omega(R_S - v) \wedge y^\omega \in d^\omega(W_S - v) \wedge$
$\exists (x^\omega_S, y^\omega_S) \cdot (x^\omega_S \in d^\omega(R_S) \wedge y^\omega_S \in d^\omega(W_S) \wedge z^\omega \in d^\omega(v) \wedge$
$(R_S, W_S, W_S^a, x^\omega_S \mapsto y^\omega_S) \in (\!|S|\!) \wedge$
$(\varnothing, v, v, (x_S)_0) \in (\!|I|\!) \wedge$
$select_{v \to R_S \cap v}(z_0) = select_{R_S \to R_S \cap v}((x_S)_0) \wedge$
$x^\omega = select_{R_S \to R_S - v}(x^\omega_S) \wedge$
$y^\omega = select_{W_S \to W_S - v}(y^\omega_S)$
$)\}$

La sémantique complète $(\!|S|\!)$ est définie à partir de la sémantique amnésique $(\!|S|\!)$, cette définition définit donc un opérateur $\diamond_{\mathcal{B}(I,S)}$ tel que $(\!|\mathcal{B}(I,S)|\!) = (\!|I|\!) \diamond_{\mathcal{B}(I,S)} (\!|S|\!)$.

Définition 23 (Composition avec les autres substitutions) *Nous définissons la sémantique amnésique de la composition d'un bloc sous-modèle avec une autre substitution en utilisant l'opérateur déjà défini pour la sémantique amnésique. Pour toute composition $\star$ ($\star$ est ; ou $\|$ ou $\mathcal{C}_P$), nous définissons, pour toutes substitutions S et T (qui peuvent contenir des blocs sous-modèles) :*

$$(\!|S \star T|\!) = (\!|S|\!) \diamond_\star (\!|T|\!)$$

Propriété 13 *La sémantique amnésique d'un bloc sous-modèle est complètement définie par la donnée de la sémantique amnésique de l'initialisation I et la sémantique amnésique de la substitution principale S : pour toutes substitutions S, I_S, T et I_T,*

$(\!|I_S|\!) = (\!|I_T|\!) \wedge (\!|S|\!) = (\!|T|\!) \Rightarrow (\!|\mathcal{B}_v(I_S, S)|\!) = (\!|\mathcal{B}_v(I_T, T)|\!)$

Cette propriété est la conséquence directe des définitions : la sémantique amnésique $(\!|\mathcal{B}_v(I,S)|\!)$ est définie à partir de la sémantique amnésique de l'initialisation $(\!|I|\!)$ et de la sémantique complète de la substitution principale $(\!|S|\!)$, qui est elle même définie à partir de la sémantique amnésique $(\!|S|\!)$.

Propriété 14 (Compositionnalité de la sémantique amnésique) *La sémantique amnésique de BHDL avec les blocs sous-modèles est compositionnelle.*

Intuitivement, cette propriété est la conséquence du fait que (1) la sémantique amnésique d'un bloc sous-modèle $\mathcal{B}_v(I,S)$ est complètement définie à partir les sémantiques amnésiques de I et S, (2) la sémantique de BHDL sans bloc sous-modèle est compositionnelle, et (3) la composition d'un bloc sous-modèle avec une autre substitution est définie de façon compositionnelle.

Formellement, nous faisons une preuve par récurrence sur le nombre d'opérateurs de composition dans la substitution. Un opérateur de composition peut être une composition séquentielle ; ou une composition parallèle $\|$ ou une composition conditionnelle $\mathcal{C}_P$ ou encore un bloc sous-modèle $\mathcal{B}_v$.

- Base de la récurrence (nombre d'opérateur de composition = 0) : il s'agit d'une substitution simple $x := E$ or $x :\in E$. La compositionnalité est évidente.

- Nous supposons maintenant la compositionnalité de la sémantique amnésique pour l'ensemble des substitutions construites avec au plus n opérateurs de composition (; ou $\|$ ou $\mathcal{C}_P$ ou $\mathcal{B}_v$). Nous prouvons la compositionnalité de la sémantique amnésique pour l'ensemble des substitutions construites avec $n+1$ opérateurs de composition.

  Notons par S une substitution construite à partir de $n+1$ opérateurs de compositions. L'opérateur de composition principal de S est noté $\star$, de sorte que $S \equiv A \star B$ ($\star$ est ; ou $\mathcal{C}_P$ ou $\|$ ou $\mathcal{B}_v$).

  Par construction, les substitutions A et B sont construites en utilisant au plus n opérateurs de compositions. Ainsi, par hypothèse, leurs sémantiques amnésiques peuvent être calculées de manière compositionnelle.

  En utilisant la définition 23, on peut dire que $(\!|A \star B|\!) = (\!|A|\!) \diamond_\star (\!|B|\!)$, avec $(\!|A|\!)$ et $(\!|B|\!)$ compositionnelles. On en conclut que $(\!|A \star B|\!)$ peut se calculer de manière compositionnelle.

Définition 24 (Sémantique complète d'une substitution pouvant contenir des blocs sous-modèles) *La sémantique complète d'un bloc sous-modèle est définie à partir de la sémantique amnésique, exactement de la même façon que dans le langage de base sans les blocs sous-modèles. Pour toute substitution S (S pouvant contenir des blocs sous-modèles),*

$$\begin{aligned}(\!|S|\!) = {} & \{R_S\} \times \{W_S\} \times \{W_S^a\} \times \\ & (rel^\omega_{R_S \times W_S} \cap \\ & \quad \{x^\omega \mapsto y^\omega | x^\omega \in d^\omega(R_S) \ \wedge \ y^\omega \in d^\omega(W_S) \ \wedge \\ & \qquad select_{R_S \to R_S \cap W_S}(x^\omega_{+1}) = select_{W_S \to R_S \cap W_S}(y^{\omega \to length(x^\omega_{+1})})\})\end{aligned}$$

Propriété 15 (NON compositionnalité de la sémantique complète) *La sémantique complète du langage BHDL avec les blocs sous-modèles* N'EST PAS *compositionnelle.*

Nous donnons un contre-exemple pour illustrer la non compositionnalité de la sémantique complète lorsqu'on ajoute les blocs sous-modèles au langage. Considérons les deux blocs sous-modèles $\mathcal{B}_1$ et $\mathcal{B}_2$ ci-dessous.

$$\mathcal{B}_1 \equiv \mathcal{B}_{\{mem\}}(mem := 0, \mathcal{C}_{mem \neq x}(x := 1, x := 0); mem := x)$$
$$\mathcal{B}_2 \equiv \mathcal{B}_{\{mem\}}(mem := 0, \mathcal{C}_{x=1}(mem := 1, mem := mem) \| \mathcal{C}_{mem=1}(x := 0, x := x))$$

On peut calculer que les sémantiques complètes de $\mathcal{B}_1$ et $\mathcal{B}_2$ sont les mêmes :

$$\langle\!|\mathcal{B}_1|\!\rangle = \langle\!|\mathcal{B}_2|\!\rangle = \{\{x\}\} \times \{\{x\}\} \times \{\{x\}\} \times \{\langle 0,0,...\rangle \mapsto \langle 0,0,...\rangle, \langle 1,1,0,0,0,...\rangle \mapsto \langle 1,0,0,...\rangle\}$$

Si la sémantique complète était compositionnelle, il serait impossible de distinguer $\mathcal{B}_1$ et $\mathcal{B}_2$, quel que soit leur utilisation. Or, si on compose $\mathcal{B}_1$ et $\mathcal{B}_2$ avec la même substitution $x := 1$, les sémantiques complètes des deux compositions ne sont plus identiques : $\langle\!|x := 1; \mathcal{B}_1|\!\rangle \neq \langle\!|x := 1; \mathcal{B}_2|\!\rangle$.

$$\langle\!|x := 1; \mathcal{B}_1|\!\rangle = \{\varnothing\} \times \{\{x\}\} \times \{\{x\}\} \times \{\langle 1,0,1,0,1,0,...\rangle\}$$
$$\langle\!|x := 1; \mathcal{B}_2|\!\rangle = \{\varnothing\} \times \{\{x\}\} \times \{\{x\}\} \times \{\langle 1,0,0,0,...\rangle\}$$

12.3 Ajout de l'importation de modèles

La sémantique de l'importation de modèles est définie par la réutilisation de la sémantique du modèle importé. Les ensembles supports sont modifiés car les entrées et les sorties du modèle importé sont remplacées par les variables auxquelles elles sont connectées. Le point essentiel est que la réutilisation de la sémantique du modèle importé (qui est une sémantique complète) produit une sémantique amnésique, et non pas une sémantique complète.

Définition 25 *La sémantique amnésique de la substitution d'importation est définie en utilisant la sémantique du modèle importé. Pour un modèle ic tel que* $\langle\!|ic|\!\rangle = \{I\} \times \{O\} \times r_{ic}$*, nous définissons la sémantique amnésique de l'importation de cette façon :*

$|\!|(CO) \leftarrow ic(CI)|\!| = \{local(CI)\} \times \{local(CO)\} \times \{local(CO)\} \times r_{ic}$

Il faut remarquer que ceci définit une sémantique amnésique, pas une sémantique complète. Ceci est dû au fait que les ensembles de variables $local(CI)$ et $local(CO)$ ne sont pas nécessairement disjoints. Par exemple, supposons que le modèle importé soit équivalent à la substitution $o := i + 1$ et que nous connections à la fois la sortie o et l'entrée i sur une variable x. Le résultat est équivalent à la substitution $x := x+1$. La sémantique complète de $x := x+1$ est $\{\langle 0,1,2...\rangle \mapsto \langle 1,2,3...\rangle, \langle 1,2,3...\rangle \mapsto \langle 2,3,4...\rangle, ...\} = \{x^\omega \mapsto y^\omega | \forall k \cdot (k \geq 0 \Rightarrow y_k = x_k + 1 \wedge x_{k+1} = y_k)\}$, mais la sémantique complète de $o := i + 1$ contient, entre autres, $\langle 0,6,4...\rangle \mapsto \langle 1,7,5...\rangle$, ce qui fait partie de la sémantique amnésique de $x := x + 1$, mais pas de sa sémantique complète.

Définition 26 (Composition avec les autres substitutions) *De la même façon que pour les blocs sous-modèles, la sémantique amnésique de la composition de deux substitutions pouvant contenir des substitutions d'importation est définie en utilisant l'opérateur* $\diamond$ *déjà défini.*

Pour toute composition $\star$ *(*$\star$ *est ; ou* $\|$ *ou* $\mathcal{C}_P$ *ou* $\mathcal{B}_v$*), nous définissons, pour toutes substitutions* S *et* T *(qui peuvent contenir des substitutions d'importation) :*

$$|\!|S \star T|\!| = |\!|S|\!| \diamond_\star |\!|T|\!|$$

Propriété 16 (Compositionnalité de la sémantique amnésique) *La sémantique amnésique du langage BHDL contenant le substitutions d'importation est compositionnelle.*

Nous ne refaisons pas la preuve déjà faite pour la propriété 14 dans le cas de l'ajout des blocs sous-modèles. Dans cette preuve, il suffit de considérer la substitution d'importation comme une substitution de base puisqu'elle ne contient aucune autre substitution.

Ce qui est plus intéressant est de voir ce que la compositionnalité signifie dans ce cas. La propriété 16 dit en particulier que si deux substitutions d'importation ont la même sémantique amnésique alors elles peuvent être échangées. De plus, nous savons que la sémantique amnésique d'une substitution d'importation est définie à partir de la liste des connexions (CO et CI) et de la sémantique du modèle importé. Ainsi, si deux modèles ont la même sémantique, on peut les échanger et la sémantique amnésique de leur importation est la même (à condition que les connexions soient inchangées).

Propriété 17 (Échange de modèles importés) *Pour tous modèles ic_1 et ic_2, et étant données les listes de connexions CO et CI,*

$$\{\!|ic_1|\!\} = \{\!|ic_2|\!\} \Rightarrow \{\!|(CO) \leftarrow ic_1(CI)|\!\} = \{\!|(CO) \leftarrow ic_2(CI)|\!\}$$

Définition 27 (sémantique complète d'une substitution pouvant contenir des substitutions d'importation) *La sémantique complète d'une substitution d'importation est définie de la même manière que toutes les autres sémantiques complètes, par restriction de la sémantique amnésique. On définit donc de manière unique la sémantique complète pour toute substitution du langage BHDL (pouvant contenir des substitutions d"importation) : pour toute substitution S,*

$$\begin{aligned}&\{\!|S|\!\} = \{R_S\} \times \{W_S\} \times \{W_S^a\} \times \\ &\quad (rel^{\omega}_{R_S \times W_S} \cap \\ &\qquad \{x^\omega \mapsto y^\omega | x^\omega \in d^\omega(R_S) \ \wedge \ y^\omega \in d^\omega(W_S) \wedge \\ &\qquad\qquad select_{R_S \to R_S \cap W_S}(x^\omega_{+1}) = select_{W_S \to R_S \cap W_S}(y^{\omega \to length(x^\omega_{+1})})\})\end{aligned}$$

Propriété 18 (NON compositionnalité de la sémantique complète) *La sémantique complète du langage BHDL avec les substitutions d'importation* N'EST PAS *compositionnelle.*

Nous prouvons cette propriété en donnant un contre-exemple. Nous adaptons l'exemple déjà donné dans le cas des blocs sous-modèles. Nous considérons d'abord les deux modèles ic_1 et ic_2 ci-dessous.

```
MACHINE ic1
INPUT i
OUTPUT o
VARIABLES mem
INITIALISATION
  mem := 0
OPERATIONS
  IF mem ≠ i THEN
    o := 1
  ELSE
    o := 0
  END
  ;
  mem := o
END
```

```
MACHINE ic2
INPUT i
OUTPUT o
VARIABLES mem
INITIALISATION
  mem := 0
OPERATIONS
  IF i = 1 THEN          IF mem = 1 THEN
    mem := 1               o := 0
  ELSE              ||   ELSE
    mem := mem             o := i
  END                    END
END
```

Considérons maintenant les deux substitutions d'importation suivantes : $(o : x) \leftarrow ic_1(i : x)$ et $(o : x) \leftarrow ic_2(i : x)$. Elles ont la même sémantique complète :

$$\begin{aligned}&\{\!|(o : x) \leftarrow ic_1(i : x)|\!\} = \{\!|(o : x) \leftarrow ic_2(i : x)|\!\} \\ &\qquad = \{x\} \times \{x\} \times \{x\} \times \{\langle 0, 0, ...\rangle \mapsto \langle 0, 0, ...\rangle, \langle 1, 1, 0, 0, 0, ...\rangle \mapsto \langle 1, 0, 0, ...\rangle\}\end{aligned}$$

Pourtant, si nous composons ces substitutions avec $x := 1$, les sémantiques complètes ne sont pas les mêmes : $\{\!|x := 1; (o : x) \leftarrow ic_1(i : x)|\!\} \neq \{\!|x := 1; (o : x) \leftarrow ic_2(i : x)|\!\}$.

$$\{\!|x := 1; (o : x) \leftarrow ic_1(i : x)|\!\} = \{\varnothing\} \times \{x\} \times \{x\} \times \{\langle 1, 0, 1, 0, 1, 0, ...\rangle\}$$

$$\{\!|x := 1; (o : x) \leftarrow ic_2(i : x)|\!\} = \{\varnothing\} \times \{x\} \times \{x\} \times \{\langle 1, 0, 0, 0, ...\rangle\}$$

12.4 Discussions

Les sections précédentes donnent les preuves que la sémantique amnésique du langage BHDL est compositionnelle et des contre-exemples pour prouver que la sémantique complète n'est pas compositionnelle. Nous discutons maintenant ces cas pour approcher ces propriétés de manière plus intuitive. C'est également l'occasion de discuter pourquoi il est nécessaire d'utiliser des traces pour définir la sémantique du langage BHDL.

12.4.1 Sémantique amnésique / sémantique complète

Considérons d'abord le langage BHDL dépourvu des blocs sous-modèles et des substitutions d'importation. La sémantique amnésique de ce langage (qui ne contient que des substitutions du langage B) est complètement définie par la relation *rel* de la méthode B. La compositionnalité de la sémantique amnésique est directement héritée de la compositionnalité de la relation *rel*. La sémantique complète est ensuite définie en ajoutant une contrainte à la sémantique amnésique. Cependant, la compositionnalité de la sémantique ne vient pas directement de la compositionnalité de la sémantique amnésique, elle provient de la possibilité de reconstruire la sémantique amnésique à partir de la sémantique complète (propriété 11). Ceci signifie que, pour ce langage sans bloc sous-modèle et sans substitution d'importation, connaître la sémantique amnésique est équivalent à connaître le sémantique complète. Deux substitutions ont la même sémantique complète si et seulement si elles ont la même sémantique amnésique. La reconstruction de la sémantique amnésique à partir de la sémantique complète est possible car les éléments initiaux des deux sémantiques sont en fait les mêmes et qu'ils constituent la relation *rel* (à partir de laquelle on construit la sémantique amnésique). C'est pourquoi la sémantique de ce langage peut être définie par la relation *rel*, sans utiliser les *traces*.

Considérons maintenant le langage contenant les blocs sous-modèles (le raisonnement est le même pour les substitutions d'importation). La différence avec le cas précédent est qu'il est, en général, impossible de reconstruire la sémantique amnésique à partir de la sémantique complète. Pour ce langage, il n'y a pas de propriété équivalente à la propriété 11. Si cela était le cas, la sémantique complète serait compositionnelle, et nous avons montré sur des contre-exemples que ce n'est pas le cas. L'impossibilité, malgré l'égalité des éléments initiaux des deux sémantiques, de reconstruire la sémantique amnésique à partir de la sémantique complète vient du fait que la sémantique amnésique d'un bloc sous-modèle ne peut pas se définir à partir d'un relation *rel*. Encore une fois si cela était possible la sémantique complète serait compositionnelle et nous avons vu que ce n'est pas le cas. La non compositionnalité de la sémantique complète nous permet donc de justifier de la nécessité d'utiliser une sémantique basée sur des traces, en utilisant la relation rel^{ω} ; la relation *rel* n'existe pas pour les blocs sous-modèles (et sur les substitutions d'importation). Avec les blocs sous-modèles, deux substitutions peuvent avoir la même sémantique complète mais des sémantiques amnésiques différentes.

Illustrons ce raisonnement sur les contre-exemples donnés dans les sections précédentes. Si nous considérons seulement les traces finies de longueur 1 (c'est-à-dire que nous ne considérons pas les traces), on peut voir qu'il est impossible de distinguer les blocs sous-modèles $\mathcal{B}_1$ et $\mathcal{B}_2$. Dans les deux cas, la sémantique amnésique est $\{0 \mapsto 0, 1 \mapsto 1\}$. Cependant, les deux blocs $\mathcal{B}_1$ et $\mathcal{B}_2$ sont différents pour les traces finies de longueur 2. Par exemple, pour la trace d'entrée $\langle 1,0 \rangle$, la trace de sortie de $\mathcal{B}_1$ est $\langle 1,1 \rangle$ alors que la trace de sortie de $\mathcal{B}_2$ est $\langle 1,0 \rangle$.

Ceci nous permet d'affirmer que les traces sont nécessaires pour définir la sémantique d'un langage qui contient des blocs sous-modèles ou des importations de modèles. Nous pouvons maintenant nous intéresser à la façon dont ceci s'interprète en termes de circuits électroniques. Nous allons voir que ceci est typiquement dû aux mémoires qui sont à l'intérieur des blocs et des modèles importés. Intéressons nous de nouveau aux contre-exemples $\mathcal{B}_1$ et $\mathcal{B}_2$.

La trace d'entrée $\langle 1,0 \rangle$ permet de différencier $\mathcal{B}_1$ et $\mathcal{B}_2$ dans la sémantique amnésique. Cependant la trace d'entrée $\langle 1,0 \rangle$ n'est pas dans la sémantique complète! Ceci est dû au fait que pour l'entrée 1 au premier cycle la sortie est nécessairement 1 (dans les deux cas). Puisque, dans notre contre-exemple, la variable de sortie est la même que la variable d'entrée, il s'agit d'une mémoire et la valeur d'entrée de cette variable aux second cycle doit être la même que la valeur de sortie au premier cycle, c'est-à-dire 1.

En fait, le contre-exemple est construit tel que la contrainte supplémentaire qui permet de définir la sémantique complète à partir de la sémantique amnésique ($x^{\omega}_{+1} = y^{\omega \to length(x^{\omega}_{+1})}$) enlève toutes les traces qui différencient $\mathcal{B}_1$ et $\mathcal{B}_2$. Le résultat est que les blocs sous-modèles $\mathcal{B}_1$ et $\mathcal{B}_2$ ont la même sémantique complète mais ils n'ont pas la même sémantique amnésique.

12.4.2 Non compositionnalité de la sémantique complète

Cette section explique intuitivement pourquoi la sémantique complète n'est pas compositionnelle. Nous nous basons ici sur les blocs sous-modèles mais le raisonnement est le même pour les substitutions

d'importation. Ce qui rend la sémantique complète non compositionnelle est la conjonction de deux facteurs :

- un bloc sous-modèle peut contenir des *mémoires cachées* (variables locales au bloc qui sont à la fois lues et écrites), dans le contre-exemple *mem* est une mémoire cachée, et
- il peut y avoir des mémoires non cachées dans un bloc sous-modèle, dans le contre-exemple x est une mémoire non cachée.

La non compositionnalité de la sémantique complète vient du fait que, par composition avec une autre substitution, x peut devenir une variable ne modélisant plus une mémoire (écrite mais plus lue dans notre exemple). Intuitivement, on peut dire que la sémantique complète crée un lien entre les mémoires cachées et les mémoires non cachées, et que ce lien ne peut pas être rompu.

Jusqu'à maintenant nous nous sommes concentrés sur le bloc sous-modèle en lui-même. Le problème reste le même si la mémoire non cachée n'est pas directement utilisée dans le bloc sous-modèle lui-même mais lié à une variable utilisée par le bloc. Par exemple, dans la substitution $r := x; \mathcal{B}_v(v := 0, o := v + r \| v := v + 1); x := o$, la variable x est une mémoire non cachée et v est une mémoire cachée. La sémantique établit que la valeur de r et la valeur de x en début de cycle sont les mêmes et que la valeur de o et la valeur de x sont les mêmes en fin de cycle. La transition de la sémantique amnésique à la sémantique complète force la valeur de x au début de cycle à être la valeur de x à la fin du cycle précédent. Indirectement, cela force la valeur de r à être la même que la valeur de o au cycle précédent. Finalement, du point de vue de la substitution entière, le bloc sous-modèle se comporte comme s'il contenait la substitution $x := v + x$ à la place de $o := v + r$.

Notons que dans ce cas, l'absence de compositionnalité de la sémantique complète n'est plus concentrée sur le bloc sous-modèle lui-même mais sur la substitution entière qui utilise le bloc. Si nous considérons le bloc seul, $\mathcal{B}_v(v := 0, o := v + r \| v := v + 1)$ ne contient pas de mémoire non cachée. Ainsi, ce bloc peut être remplacé par toute substitution qui a la même sémantique complète que ce bloc. Cependant, si nous considérons la substitution entière $r := x; \mathcal{B}_v(v := 0, o := v + r \| v := v + 1); x := o$, elle ne peut pas être remplacée, a priori, par toute substitution ayant la même sémantique complète ! La substitution entière contient une mémoire cachée (qui vient du bloc sous-modèle) et une mémoire non cachée (la variable x est à la fois lue et écrite) qui sont liées.

Dans certains cas, une substitution peut avoir une mémoire cachée et une mémoire non cachée, mais qui ne sont pas "liées". Un exemple trivial est $x := x + 1; \mathcal{B}_v(v := 0, v := v + 1)$ où les ensembles supports de $x := x + 1$ et $\mathcal{B}_v(v := 0, v := v + 1)$ sont disjoints.

12.4.3 Compositionnalité et modularité

Savoir si les mémoires cachées et les mémoires non cachées sont liées ou non n'est pas si évident et il est difficile en général de savoir si deux substitutions qui ont la même sémantique complète peuvent être permutées ou non. En fait, nous ne voulons pas véritablement nous préoccuper de ce genre de question. Les modifications de substitutions (sans bloc sous-modèle ni substitution d'importation) peuvent être faites dans une processus de raffinement B traditionnel.

Les blocs sous-modèles et les substitutions d'importation sont ajoutés au langage BHDL pour permettre la modularité et la réutilisation de modèles. Heureusement, on peut remarquer que la sémantique complète peut être considérée comme compositionnelle du point de vue de la modularité. C'est-à-dire qu'un bloc sous-modèle peut être remplacé par un autre dont les substitutions intérieures ont la même sémantique complète et une importation de modèle peut être remplacée par l'importation d'un autre composant qui a la même sémantique. Ces deux principes sont basés sur les propriétés suivantes :

1. La sémantique amnésique est compositionnelle (propriété 16)
2. Pour les blocs sous-modèles et les substitutions d'importation, la réutilisation des sémantiques complètes produit une sémantique amnésique
   a) pour les blocs sous-modèles : (propriété 13) la sémantique amnésique d'un bloc sous-modèle est complètement définie par la sémantique complète de la substitution intérieure au bloc.

b) pour les substitutions d'importation : (propriété 17) la sémantique amnésique d'une substitution d'importation est complètement définie par la sémantique complète du modèle importé (et par la façon dont les entrées et les sorties sont connectées).

Ainsi, un bloc sous modèle peut être remplacé par tout bloc dont la substitution a la même sémantique complète, car le résultat est un bloc qui a la même sémantique amnésique. De la même façon, un modèle peut être remplacé par un autre qui a la même sémantique complète car l'importation de ces modèles (avec les mêmes connexions pour les entrées et les sorties) résulte en une même sémantique amnésique.

12.5 Résumé

Dans ce chapitre, nous avons défini deux sémantiques relationnelles. La première, la sémantique amnésique a l'avantage d'être toujours compositionnelle alors que la deuxième, la sémantique complète, ne l'est plus lorsqu'on introduit des blocs sous-modèles ou des substitutions d'importation. Sur le langage de base de BHDL (sans les blocs et sans l'importation), ces deux sémantiques sont compositionnelles. On résume brièvement quelques propriétés intéressantes ci-dessous :

- La sémantique amnésique est compositionnelle sur l'ensemble du langage BHDL
- La sémantique complète n'est pas compositionnelle sur l'ensemble du langage, mais
  - Le sémantique complète est compositionnelle sur le langage BHDL sans bloc sous-modèle ni substitution d'importation
  - La sémantique amnésique et la sémantique complète sont égale lorsqu'il n'y a aucune mémoire (ou quand toutes les mémoires sont complètement cachées)
  - Si les substitutions de deux blocs sous-modèles ont la même sémantique complète alors les deux blocs ont la même sémantique amnésique.
  - Si deux modèles ont la même sémantique, leur importation (avec les mêmes connexions pour les entrées et les sorties) ont la même sémantique amnésique. Les deux modèles peuvent donc être échangés.
- La sémantique complète n'est pas compositionnelle car des variables modélisant des mémoires non cachées peuvent être "liées" à des variables modélisant des mémoires cachées.
- Les sémantiques amnésique et complète d'un modèle sont égales car toutes les mémoires du modèle sont cachées.

S'il n'y a pas de mémoire non cachée, il n'y a pas de problème de compositionnalité : les sémantiques amnésique et complète sont égales.

Les propriétés sur les blocs sous-modèles et les substitutions d'importation sont équivalentes. En fait un bloc peut être remplacé (raffiné) par une substitution d'importation, et vice-versa.

Chapitre 13

Importation de composants

Nous avons proposé d'introduire deux nouveaux constructeurs au langage BHDL. A l'origine, un modèle BHDL provient d'un système B événementiel qui ne contient ni bloc sous-modèle ni substitution d'importation. Nous expliquons dans cette section comment introduire les blocs sous-modèles et les substitutions d'importation dans un modèle existant.

En fait, le rôle d'un bloc sous-modèle est de préparer le modèle pour l'introduction d'une substitution d'importation. Le processus d'intégration d'une substitution d'importation doit se faire classiquement de cette façon :

1. enfermer dans un bloc sous-modèle la partie du modèle qui doit être remplacée par une substitution d'importation,
2. remplacer ce bloc par une substitution d'importation qui a la même sémantique amnésique que le bloc sous-modèle.

Le processus d'introduire les blocs sous-modèles puis les substitutions d'importation peut être vu comme un processus de raffinement sur la structure du circuit, il ne doit pas modifier la sémantique du modèle.

13.1 Introduire des blocs sous-modèles dans le modèle

L'objectif d'un bloc sous-modèle est de cacher certaines variables au reste du modèle. Lorsqu'un bloc est introduit, ces variables ne font plus partie à part entière du modèle, elles sont supposées faire partie de la spécification du sous-composant, qui est modélisé par le modèle qui sera importé. Le bloc sous-modèle est considéré comme la spécification d'un sous-composant. L'introduction d'un bloc sous-modèle se fait en appliquant le processus suivant :

1. Identifier quelle partie du modèle doit être enfermée dans un bloc sous-modèle.
2. Identifier, à l'intérieur de la substitution qui a été enfermée dans le bloc, quelles variables sont utilisées uniquement par cette substitution et pas dans le reste du modèle. Si une variable est également utilisée dans le reste du modèle (à l'extérieur du bloc) elle ne peut pas être cachée et déclarée à l'intérieur du bloc.
3. Retirer ces variables de la clause VARIABLES du modèle et les déclarer à l'intérieur du bloc sous-modèle. Les variables qui sont également utilisées à l'extérieur du bloc ne sont pas touchées. On peut éventuellement choisir de laisser certaines variables globales bien qu'elles ne soient utilisées qu'à l'intérieur du bloc.
4. Déplacer les déclarations de type et les initialisations de ces variables du modèle global vers le bloc sous-modèle.
5. Vérifier que le bloc sous-modèle est bien formé, en particulier que toute variable globale écrite est écrite dans tous les chemins (l'ensemble support en écriture hors variables locales doit être égal à l'ensemble support total en écriture hors variables locales).

6. Vérifier que le bloc sous-modèle ainsi obtenu correspond au modèle que l'on souhaite importer. En particulier, vérifier que les entrées (resp. les sorties) correspondent aux variables qui sont lues (resp. écrites) par le bloc sous-modèle et qui ne sont pas déclarées locales au bloc. Il n'y a pas nécessairement une correspondance 1-1 entre les variables du bloc et le modèle qu'on souhaite importer : certaines variables du modèle contenant le bloc peuvent servir à la fois d'entrée et de sortie. Par ailleurs il peut y avoir renommage des variables. Tous ces points sont éclaircis en précisant la liste des connexions entre les variables du modèle contenant le bloc et les entrées et sorties du modèle importé.

Une fois que le bloc a été introduit, on peut le remplacer par tout bloc qui a la même sémantique amnésique, ce qui inclut, comme cela a été expliqué au chapitre précédent, tout bloc dont la substitution a la même sémantique complète que la substitution du bloc original.

Une fois qu'on a obtenu un bloc correspondant au modèle qu'on souhaite importé, ce bloc peut être remplacé par une substitution d'importation. La section suivante justifie comment ce remplacement peut être effectué.

13.2 Remplacement d'un bloc sous-modèle par une substitution d'importation

Pour remplacer un bloc sous-modèle par une substitution d'importation, il est nécessaire que le bloc et la substitution aient la même sémantique amnésique. Pour un bloc donné, nous montrons dans cette section quel modèle peut être importé à la place du bloc et nous prouvons que la sémantique amnésique est préservée.

Propriété 19 *Un bloc sous-modèle $\mathcal{B}_v(I,S)$ et une substitution d'importation d'un modèle ic défini ci-dessous ont la même sémantique amnésique. L'ensemble de variables R_S est l'ensemble support en lecture de S, et W_S son ensemble support en écriture.*

MACHINE ic
INPUTS $(R_S - W_S - v) \cup (R_S \cap W_S - v)_{in}$
OUTPUTS $(W_S - R_S - v) \cup (R_S \cap W_S - v)_{out}$
VARIABLES $v \cup (R_S \cap W_S - v)$
INITIALISATION
  $I \| \mathit{non\text{-}determ\text{-}init}((R_S \cap W_S - v)_{in} \cup (R_S \cap W_S - v)_{out})$
OPERATIONS
  $(R_S \cap W_S - v) := (R_S \cap W_S - v)_{in}$;
  S ;
  $(R_S \cap W_S - v)_{out} := (W_S \cap R_S - v)$
END

Ce modèle doit être importé par la substitution suivante, les notations F_R (resp. F_W) correspond à l'ensemble des variables non locales au bloc $\mathcal{B}_v(I,S)$ qui sont uniquement lues (resp. uniquement écrites) par S :

$$(F_R : F_R, W_Q : R_Q) \leftarrow ic(F_W : F_W, R_T : W_T)$$

Pour simplifier les expressions, nous notons par Q la première substitution du modèle ic et par T la troisième substitution. La substitution ajoutée à l'initialisation est notée J.

$$J \equiv \mathit{non\text{-}determ\text{-}init}((R_S \cap W_S - v)_{in} \cup (R_S \cap W_S - v)_{out})$$
$$Q \equiv (R_S \cap W_S - v) := (R_S \cap W_S - v)_{in}$$
$$T \equiv (R_S \cap W_S - v)_{out} := (W_S \cap R_S - v)$$

Le modèle ic est importé en utilisant la substitution d'importation ci-dessous. Elle consiste à connecter les entrées qui n'ont pas été renommées directement aux variables originales qui existent dans le modèle

importateur. Les entrées renommées sont connectées aux variables originales. La même chose est faite pour les sorties. Pour simplifier les expressions, nous notons l'ensemble de variables $R_S - W_S - v$ par F_R, $W_S - R_S - v$ par F_W, $R_S \cap W_S - v$ par W_Q ou R_T suivant les cas ($W_Q = R_T$), $(R_S \cap W_S - v)_{in}$ par R_Q et $(R_S \cap W_S - v)_{out}$ par W_T. Les notations proviennent de la correspondance avec les ensembles supports des substitutions Q et T.

$$(F_R : F_R, W_Q : R_Q) \leftarrow ic(F_W : F_W, R_T : W_T)$$

Nous voulons prouver que la sémantique amnésique de cette substitution d'importation est la même que la sémantique amnésique du bloc sous-modèle $\mathcal{B}_v(I, S)$. Nous commençons par calculer l'expression de la sémantique du modèle ic et la sémantique amnésique de son importation. L'expression ci-dessous est l'expression de la sémantique du modèle ic telle que nous la donne la définition.

$$\begin{aligned}
\langle\!|ic|\!\rangle = \{\!|D|\!\} = \{&R_S - W_S - v) \cup (R_S \cap W_S - v)_{in}\} \times \\
&\{W_S - R_S - v) \cup (R_S \cap W_S - v)_{out}\} \times \\
&\{W_S - R_S - v) \cup (R_S \cap W_S - v)_{out}\} \times f_{ic},
\end{aligned}$$

où $f_{ic} = \{x^\omega \mapsto y^\omega | \exists (R, W, W^a, W_{I\|J}, W^a_{I\|J}, X^\omega, Y^\omega, Z^\omega) \cdot ($

$$\begin{aligned}
&(R, W, W^a, (X^\omega = \langle X_0, X_1, ...\rangle) \mapsto Y^\omega) \in \langle\!|Q; S; T|\!\rangle \wedge \\
&(\varnothing, W_{I\|J}, W^a_{I\|J}, Z^\omega = \langle Z_0, Z_1, ...\rangle) \in \{\!|I\|J|\!\} \wedge \\
&select_{W_{I\|J} \to R \cap W_{I\|J}}(Z_0) = select_{R \to R \cap W_{I\|J}}(X_0) \wedge \\
&x^\omega = select_{R \to F_R \cup R_Q}(X^\omega) \wedge \\
&y^\omega = select_{W \to F_W \cup W_T}(Y^\omega) \\
&)\}
\end{aligned}$$

Les ensembles supports de I et $I\|J$ sont différents mais nous prouvons l'équivalence ci-dessous pour toute trace $X^\omega = \langle X_0, X_1, ...\rangle$ et R étant l'ensemble support en lecture de $Q; S; T$. Notons que $W_{I\|J}$ est l'ensemble de toutes les variables (y compris les entrées et les sorties) du composant ic, et W_i est l'ensemble des variables de $\mathcal{B}_v(I, S)$; ces deux ensembles sont des sur-ensembles de R.

$$\exists (W_{I\|J}, W^a_{I\|J}, X^\omega, Z^\omega) \cdot \begin{pmatrix} (\varnothing, W_{I\|J}, W^a_{I\|J}, Z^\omega = \langle Z_0, Z_1, ...\rangle) \in \{\!|I\|J|\!\} \wedge \\ select_{W_{I\|J} \to R}(Z_0) = X_0 \end{pmatrix}$$

$$\Leftrightarrow$$

$$\exists (W_i, W^a_i, X^\omega, Z^\omega) \cdot ((\varnothing, W_i, W^a_i, Z^\omega = \langle Z_0, Z_1, ...\rangle) \in \{\!|I|\!\} \wedge select_{W_i \to R}(Z_0) = X_0)$$

Les sémantiques $\{\!|I\|J|\!\}$ et $\{\!|I|\!\}$ définissent des ensembles supports et des traces sur des variables (relations qui ont $\varnothing$ comme domaine). L'équivalence vient du fait que nous ne sommes pas intéressés par les ensembles supports (seulement leur existence est requise, ce qui est le cas), et les deux traces sont égales sur l'ensemble de variables R. Cette égalité est due à ce que la seule différence entre les initialisations $I\|J$ et I, est que $I\|J$ ajoute des nouvelles variables, et, en particulier, ne modifie pas les variables de R. Nous ne donnons pas de preuve formelle de cette équivalence ici, la preuve consiste simplement à développer les deux définitions.

Sur le même principe, nous montrons qu'on peut remplacer $\langle\!|Q; S; T|\!\rangle$ par $\langle\!|S|\!\rangle$. Les ensembles supports de $Q; S; T$ et S sont différents mais nous prouvons que $rel^\omega_{R_{Q;S;T} \times W_{Q;S;T}}(Q; S; T) = rel^\omega_{R_S \times W_S}(S)$. Nous donnons la preuve de cette assertion ci-dessous. On commence par calculer $rel^\omega_{R_{Q;S;T} \times W_{Q;S;T}}(Q; S; T)$. Nous écrivons ci-dessous les définitions de $rel^\omega_{R_{Q;S;T} \times W_{Q;S;T}}(Q; S; T)$ et $rel^\omega_{R_{Q;S} \times W_{Q;S}}(Q; S)$ en utilisant la compositionnalité de $rel^\omega_{R \times W}$, .

$$\begin{aligned}
&rel^\omega_{R_{Q;S;T} \times W_{Q;S;T}}(Q; S; T) = \\
&\quad \{x^\omega \mapsto y^\omega | x^\omega \in d(R_{Q;S;T}) \wedge y^\omega \in d^\omega(W_{Q;S;T}) \wedge \exists (X^\omega, Y^\omega, U^\omega, x^\omega_{Q;S}, x^\omega_{Q;S}, x^\omega_T, x^\omega_T) \cdot (\\
&\qquad X^\omega \in d^\omega(V) \wedge Y^\omega \in d^\omega(V) \wedge \\
&\qquad select_{V \to V - W_{Q;S}}(U^\omega) = select_{V \to V - W_{Q;S}}(X^\omega) \\
&\qquad x^\omega_{Q;S} \mapsto y^\omega_{Q;S} \in \boldsymbol{rel^\omega_{R_{Q;S} \times W_{Q;S}}(Q; S)} \\
&\qquad x^\omega_{Q;S} = select_{V \to R_{Q;S}}(X^\omega) \\
&\qquad y^\omega_{Q;S} = select_{V \to W_{Q;S}}(U^\omega) \\
&\qquad select_{V \to V - W_T}(Y^\omega) = select_{V \to V - W_T}(U^\omega) \\
&\qquad x^\omega_T \mapsto y^\omega_T \in \boldsymbol{rel^\omega_{R_T \times W_T}(T)}
\end{aligned}$$

$$
\begin{array}{l}
x^\omega_T = select_{V\to R_T}(U^\omega)\\
y^\omega_T = select_{V\to W_T}(Y^\omega)\\
x^\omega = select_{V\to R_{Q;S;T}}(X^\omega)\wedge\\
y^\omega = select_{V\to W_{Q;S;T}}(Y^\omega)\\
)\}
\end{array}
$$

$$
\begin{array}{l}
rel^\omega_{R_{Q;S}\times W_{Q;S}}(Q;S) =\\
\quad \{x^\omega_{Q;S} \mapsto y^\omega_{Q;S} | x^\omega_{Q;S} \in d^\omega(R_{Q;S}) \wedge y^\omega_{Q;S} \in d^\omega(W_{Q;S}) \wedge \exists (I^\omega, O^\omega, K^\omega, x^\omega_Q, x^\omega_Q, x^\omega_S, x^\omega_S) \cdot (\\
\quad\quad I^\omega \in d^\omega(V) \wedge O^\omega \in d^\omega(V) \wedge\\
\quad\quad select_{V\to V-W_Q}(K^\omega) = select_{V\to V-W_Q}(I^\omega)\\
\quad\quad x^\omega_Q \mapsto y^\omega_Q \in \boldsymbol{rel^\omega_{R_Q\times W_Q}(Q)}\\
\quad\quad x^\omega_Q = select_{V\to R_Q}(I^\omega)\\
\quad\quad y^\omega_Q = select_{V\to W_Q}(K^\omega)\\
\quad\quad select_{V\to V-W_S}(O^\omega) = select_{V\to V-W_S}(K^\omega)\\
\quad\quad x^\omega_S \mapsto y^\omega_S \in \boldsymbol{rel^\omega_{R_S\times W_S}(S)}\\
\quad\quad x^\omega_S = select_{V\to R_S}(K^\omega)\\
\quad\quad y^\omega_S = select_{V\to W_S}(O^\omega)\\
\quad\quad x^\omega_{Q;S} = select_{V\to R_{Q;S}}(I^\omega)\wedge\\
\quad\quad y^\omega_{Q;S} = select_{V\to W_{Q;S}}(O^\omega)\\
\quad)\}
\end{array}
$$

Nous développons la définition de $rel^\omega_{R_{Q;S}\times W_{Q;S}}(Q;S)$ à l'intérieur de la définition, donnée ci-dessus, de $rel^\omega_{R_{Q;S;T}\times W_{Q;S;T}}(Q;S;T)$ pour obtenir la définition ci-dessous. Certaines lignes sont numérotées pour pouvoir s'y référer plus facilement dans la preuve qui suit.

$$
\begin{array}{ll}
rel^\omega_{R_{Q;S;T}\times W_{Q;S;T}}(Q;S;T) = & \\
\quad \{x^\omega \mapsto y^\omega | x^\omega \in d(R_{Q;S;T}) \wedge y^\omega \in d^\omega(W_{Q;S;T}) \wedge & \\
\quad\quad \exists (X^\omega, Y^\omega, U^\omega, x^\omega_{Q;S}, x^\omega_{Q;S}, x^\omega_T, x^\omega_T, I^\omega, O^\omega, K^\omega, x^\omega_Q, x^\omega_Q, x^\omega_S, x^\omega_S) \cdot (& \\
\quad\quad X^\omega \in d^\omega(V) \wedge Y^\omega \in d^\omega(V) \wedge & \\
\quad\quad select_{V\to V-W_{Q;S}}(U^\omega) = select_{V\to V-W_{Q;S}}(X^\omega) & \\
\quad\quad x^\omega_{Q;S} \in d^\omega(R_{Q;S}) \wedge y^\omega_{Q;S} \in d^\omega(W_{Q;S}) \wedge & \\
\quad\quad I^\omega \in d^\omega(V) \wedge O^\omega \in d^\omega(V) \wedge & \\
(1) \quad\quad select_{V\to V-W_Q}(K^\omega) = select_{V\to V-W_Q}(I^\omega) & \\
(2) \quad\quad x^\omega_Q \mapsto y^\omega_Q \in \boldsymbol{rel^\omega_{R_Q\times W_Q}(Q)} & \\
(3) \quad\quad x^\omega_Q = select_{V\to R_Q}(I^\omega) & \\
(4) \quad\quad y^\omega_Q = select_{V\to W_Q}(K^\omega) & \\
\quad\quad select_{V\to V-W_S}(O^\omega) = select_{V\to V-W_S}(K^\omega) & \\
\quad\quad x^\omega_S \mapsto y^\omega_S \in \boldsymbol{rel^\omega_{R_S\times W_S}(S)} & \\
(5) \quad\quad x^\omega_S = select_{V\to R_S}(K^\omega) & \\
\quad\quad y^\omega_S = select_{V\to W_S}(O^\omega) & \\
(6) \quad\quad x^\omega_{Q;S} = select_{V\to R_{Q;S}}(I^\omega)\wedge & \\
\quad\quad y^\omega_{Q;S} = select_{V\to W_{Q;S}}(O^\omega) & \\
(7) \quad\quad x^\omega_{Q;S} = select_{V\to R_{Q;S}}(X^\omega) & \\
\quad\quad y^\omega_{Q;S} = select_{V\to W_{Q;S}}(U^\omega) & \\
\quad\quad select_{V\to V-W_T}(Y^\omega) = select_{V\to V-W_T}(U^\omega) & \\
\quad\quad x^\omega_T \mapsto y^\omega_T \in \boldsymbol{rel^\omega_{R_T\times W_T}(T)} & \\
\quad\quad x^\omega_T = select_{V\to R_T}(U^\omega) & \\
\quad\quad y^\omega_T = select_{V\to W_T}(Y^\omega) & \\
(8) \quad\quad x^\omega = select_{V\to R_{Q;S;T}}(X^\omega)\wedge & \\
\quad\quad y^\omega = select_{V\to W_{Q;S;T}}(Y^\omega) & \\
\quad\quad)\} &
\end{array}
$$

Nous voulons exhiber la relation entre les deux couples $x^\omega \mapsto y^\omega$ et $x^\omega_S \mapsto y^\omega_S$. Nous prouvons qu'ils sont égaux. Ceci s'exprime par les deux propriétés a) et b) ci-dessous. Par conséquence, ceci prouve que la relation de $Q;S;T$ est la même que S (il y a juste un renommage des variables entre les deux et la relation ne prend pas en compte les noms des variables).

a) $x^\omega = x^\omega_S$

Le preuve se fait principalement en manipulant les ensembles ainsi que les égalités qui ont été numérotées ci-dessus. Pour réduire la taille de la preuve nous omettons le calcul des ensembles supports, les définitions peuvent être trouvées dans la section 9.2.2 page 182.

A partir des définitions de x^ω et x^ω_S par les égalités (5) and (8), on déduit que nous avons en fait à prouver que $select_{V\to R_{Q;S;T}}(X^\omega) = select_{V\to R_S}(K^\omega)$. Les égalités (6) et (7) nous conduisent à $select_{V\to R_{Q;S;T}}(X^\omega) = select_{V\to R_{Q;S;T}}(I^\omega)$ car $R_{Q;S;T} \subseteq R_{Q;S}$. De plus, on remarque que $R_{Q;S;T} \cap W_T = \varnothing$, donc, en utilisant l'égalité (1), nous trouvons finalement que nous avons à prouver l'égalité ci-dessous.

$$(*) select_{V\to R_{Q;S;T}}(K^\omega) = select_{V\to R_S}(K^\omega)$$

Nous calculons d'abord $R_{Q;S;T}$. Après quelques simplifications, nous obtenons $R_{Q;S;T} = R_Q \cup (R_S - W_Q)$.

En utilisant la définition de Q ($Q \equiv (R_S \cap W_S - v) := (R_S \cap W_S - v)_{in}$), nous savons que $rel^\omega_{R_Q \times W_Q}(Q)$ est l'identité, donc, à partir de la ligne (2) nous déduisons que $x^\omega_Q = y^\omega_Q$; et à partir des égalités numérotées (3) and (4), nous avons $select_{V\to R_Q}(I^\omega) = select_{V\to W_Q}(K^\omega)$. De plus, par l'égalité (1), et en remarquant que $R_Q \cap W_Q = \varnothing$, nous avons $select_{V\to R_Q}(K^\omega) = select_{V\to R_Q}(I^\omega)$. Finalement, nous arrivons à $select_{V\to R_Q}(K^\omega) = select_{V\to W_Q}(K^\omega)$.

Puisque $R_Q \cap (R_S - W_Q) = \varnothing$ et $W_Q \cap (R_S - W_Q) = \varnothing$, on peut décomposer $select_{V\to R_{Q;S;T}}(K^\omega)$ en $select_{V\to R_Q}(K^\omega)$ et $select_{V\to R_S - W_Q}(K^\omega)$; et (*) se réécrit en

$$(**) select_{V\to W_Q \cup (R_S - W_Q)}(K^\omega) = select_{V\to R_S}(K^\omega)$$

Pour finir, puisque $W_Q \cup (R_S - W_Q) = R_S$, l'égalité (**) devient évidente et nous avons prouvé l'égalité $x^\omega = x^\omega_S$.

b) $y^\omega = y^\omega_S$

La preuve se fait exactement de la même façon que celle de la propriété a), nous ne la donnons pas ici.

Ceci termine la preuve de la propriété suivante : $rel^\omega_{R_{Q;S;T} \times W_{Q;S;T}}(Q;S;T) = rel^\omega_{R_S \times W_S}(S)$. Nous avons finalement prouvé que la sémantique de ic peut se définir par l'expression que nous donnons ci-dessous. Notons que, du fait de la modification des ensembles supports (renommage de certaines variables), nous devons changer $R \to F_R \cup R_Q$ (resp. $F_W \cup W_T$) en $R \to F_R \cup W_Q$ (resp. $R \to F_W \cup R_T$) dans l'expression de l'opérateur *select* utilisé dans la définition de x^ω (resp. y^ω). De plus, nous utilisons les deux égalités suivantes, qui s'obtiennent en calculant les ensembles supports : $F_R \cup W_Q = R_S - v$ et $F_W \cup R_T = W_S - v$.

$\langle\!|ic|\!\rangle = |\!|D|\!| = \{R_S - W_S - v) \cup (R_S \cap W_S - v)_{in}\} \times \{W_S - R_S - v) \cup (R_S \cap W_S - v)_{out}\} \times \{W_S - R_S - v) \cup (R_S \cap W_S - v)_{out}\} \times f_{ic}$,

où $f_{ic} = \{x^\omega \mapsto y^\omega | \exists (R_S, W_S, W^a_S, W_i, W^a_i, X^\omega, Y^\omega, Z^\omega) \cdot ($
$(R_S, W_S, W^a_S, (X^\omega = \langle X_0, X_1, ...\rangle) \mapsto Y^\omega) \in \langle\!|S|\!\rangle \wedge$
$(\varnothing, W_i, W^a_i, Z^\omega = \langle Z_0, Z_1, ...\rangle) \in |\!|I|\!| \wedge$
$select_{W_i \to R_S \cap W_i}(Z_0) = select_{R_S \to R_S \cap W_i}(X_0) \wedge$
$x^\omega = select_{R_S \to R_S - v}(X^\omega) \wedge$
$y^\omega = select_{W_S \to W_S - v}(Y^\omega)$
$)\}$

On peut maintenant calculer la sémantique amnésique de l'importation de ce modèle. Remarquons que le modèle ic a le même nombre d'entrées que de variables connectées à elles par la substitution d'importation (il n'y a pas de variable qui soit connectée à plusieurs entrées). Ainsi, la sémantique amnésique de la substitution d'importation consiste simplement à modifier les ensembles supports ; la relation elle-même est inchangée. Il n'est pas nécessaire de recalculer les ensembles supports de la substitution d'importation : par construction l'ensemble support en lecture est $R_S - v$ et l'ensemble support en écriture est $W_S - v$. La sémantique amnésique de la substitution d'importation est donnée ci-dessous.

$\{\!|(F_R : F_R, W_Q : R_Q) \leftarrow ic(F_W : F_W, R_T : W_T)|\!\} = \{R_S - v\} \times \{W_S - v\} \times \{W_S - v\} \times f_{ic}$, où $f_{ic} = \{x^\omega \mapsto y^\omega | \exists (R_S, W_S, W_S^a, W_i, W_i^a, X^\omega, Y^\omega, Z^\omega) \cdot ($
$(R_S, W_S, W_S^a, (X^\omega = \langle X_0, X_1, ... \rangle) \mapsto Y^\omega) \in \{\!|S|\!\} \wedge$
$(\varnothing, W_i, W_i^a, Z^\omega = \langle Z_0, Z_1, ... \rangle) \in \{\!|I|\!\} \wedge$
$select_{W_i \rightarrow R_S \cap W_i}(Z_0) = select_{R_S \rightarrow R_S \cap W_i}(X_0) \wedge$
$x^\omega = select_{R_S \rightarrow R_S - v}(X^\omega) \wedge$
$y^\omega = select_{W_S \rightarrow W_S - v}(Y^\omega)$
$)\}$

Nous pouvons maintenant comparer la sémantique amnésique de la substitution d'importation du modèle ic donnée ci-dessus avec la sémantique amnésique du bloc sous-modèle $\mathcal{B}_v(I, S)$ rappelée ci-dessous.

$\{\!|\mathcal{B}_v(I,S)|\!\} = \{R_S - v\} \times \{W_S - v\} \times \{W_S^a - v\} \times rel^\omega_{R_{\mathcal{B}_v(I,S)} \times W_{\mathcal{B}(v,I,S)}}(\mathcal{B}_v(I,S))$

$rel^\omega_{R_{\mathcal{B}_v(I,S)} \times W_{\mathcal{B}_v(I,S)}}(\mathcal{B}_v(I,S))$
$= \{x^\omega \mapsto y^\omega | x^\omega \in d^\omega(R_S - v) \wedge y^\omega \in d^\omega(W_S - v) \wedge$
$\exists (x_S^\omega, y_S^\omega) \cdot (x_S^\omega \in d^\omega(R_S) \wedge y_S^\omega \in d^\omega(W_S) \wedge z^\omega \in d^\omega(v) \wedge$
$(R_S, W_S, W_S^a, x_S^\omega \mapsto y_S^\omega) \in \{\!|S|\!\} \wedge$
$(\varnothing, v, v, z^\omega) \in \{\!|I|\!\} \wedge$
$select_{v \rightarrow R_S \cap v}(z_0) = select_{R_S \rightarrow R_S \cap v}((x_S)_0) \wedge$
$x^\omega = select_{R_S \rightarrow R_S - v}(x_S^\omega) \wedge$
$y^\omega = select_{W_S \rightarrow W_S - v}(y_S^\omega)$
$)\}$

La seule différence entre les deux sémantiques amnésiques est l'expression des ensembles supports total en écriture. En fait, les conditions de bonne formation d'un bloc sous-modèle imposent que $W_S - v = W_S^a - v$, on peut donc conclure que les deux sémantiques amnésiques sont égales.

$$\{\!|\mathcal{B}_v(I,S)|\!\} = \{\!|(F_R : F_R, W_Q : R_Q) \leftarrow ic(F_W : F_W, R_T : W_T)|\!\}$$

Puisque les sémantiques amnésiques de la substitution d'importation de ic et du bloc sous-modèle $\mathcal{B}_v(I, S)$ sont égales, le bloc peut être remplacé par l'importation de ic, comme cela a été expliqué au chapitre précédent. De plus, si un autre modèle D a la même sémantique que ic, son importation a la même sémantique amnésique. Ainsi, le bloc sous-modèle peut être remplacé par l'importation de tout modèle ayant la même sémantique que ic

Propriété 20 (Remplacement d'un bloc sous-modèle par une importation) *Un bloc sous-modèle $\mathcal{B}_v(I, S)$ peut être remplacé par l'importation de tout modèle qui a la même sémantique que le modèle ic défini dans la propriété 19.*

13.3 Résumé

Ce chapitre a montré comment on peut introduire une importation de composant dans un modèle BHDL. Cela se fait en passant d'abord par un bloc sous-modèle, puis ce bloc est remplacé par une substitution d'importation. Le bloc sous-modèle a l'avantage de localiser certaines variables à un bloc de code, ces variables étant supposées ne plus faire partie du modèle lorsque le remplacement par une substitution d'importation sera faite. Nous avons montré que ce remplacement conserve bien la sémantique du modèle, sous l'hypothèse que le bloc sous-modèle et le composant importé ont la même sémantique.

Chapitre 14

Traduction vers des langages de description de circuit

Au lieu de définir plusieurs traductions du langage BHDL vers des langages de description de circuits (comme VHDL, Verilog ou SystemC), nous définissons ici un langage de référence vers lequel la traduction est faite. Ce langage de référence utilise des concepts proches des langages traditionnels de description de circuits, ainsi la traduction à partir du langage de référence vers les langages traditionnels est très simple et consiste essentiellement en un changement de syntaxe, pas de changement de concepts. Les transformations principales ont lieu durant la traduction de BHDL vers le langage de référence. De plus, utiliser ce langage de référence permet d'éviter d'avoir à écrire des sémantiques formelles des langages comme VHDL et SystemC. Donner des sémantiques formelles à ces langages relève d'une entreprise plus ambitieuse. A titre d'exemple, le livre [64] est entièrement consacré à la sémantique de VHDL, et le langage SystemC est une bibliothèque ajoutée au dessus de C++, écrire une sémantique formelle de C++ est loin d'être une simple affaire. D'où l'intérêt de notre approche de modélisation basée sur la méthode B, déjà doté d'une sémantique formelle. Les sémantiques formelles que nous avons données à BHDL sont d'ailleurs inspirées des sémantiques qui ont été étudiées pour VHDL. Le langage de référence est également doté d'une sémantique formelle, décrite dans la suite de cette section. Cette sémantique suit le même principe que la sémantique prédicative de BHDL, et elle nous permet de prouver la correction des règles de traduction. Dans la suite, nous désignons le langage de référence par l'acronyme HDL (*Hardware description language*). Aussi, comme nous désignons par le mot *modèle* une représentation formelle d'un circuit en BHDL, nous utilisons le mot *description* pour désigner une représentation d'un circuit dans le langage de référence.

Dans la suite de cette section, après avoir défini quelques opérateurs qui nous seront utiles, nous définissons la syntaxe du langage de référence puis sa sémantique en termes de prédicat sur les traces. Nous donnons ensuite les règles de traduction permettant de générer des descriptions de circuits dans ce langage de référence à partir d'un modèle BHDL.

14.1 Quelques opérateurs utiles

Nous définissons quelques opérateurs qui nous seront utiles pour accéder aux variables ou aux types des variables à partir des constructeurs du langage de référence. Ces opérateurs sont équivalents à certains opérateurs déjà définis dans le contexte du langage BHDL, opérateurs que nous ne réutilisons pas pour éviter toute confusion entre les deux langages.

L'opérateur *var* prend en argument une liste de prédicats séparés par des points virgules (;). Les prédicats sont des prédicats de typage de variables, de la forme $x \in F$ pour une variable x de type F. Cet opérateur retourne l'ensemble des variables (sans leurs types) qui correspond à la liste de typage donnée en argument.

$$var(V;W) = var(V) \cup var(W)$$
$$var(x \in F) = \{x\}$$

L'opérateur *tvar* prend en argument une variable x et une liste de typage L (de même nature que les listes prises en argument par l'opérateur *var* défini précédemment). Il retourne le typage de la variable x dans la liste L (un prédicat de la forme $x \in F$), tel qu'il est mentionné dans la liste L. L'opérateur *tvar* peut également prendre en argument un ensemble de variables (au lieu d'une seule variable), dans ce cas, il retourne une liste de typage qui correspond exactement à l'ensemble des variables donné en argument. La définition de *tvar* utilise l'opérateur *itype* qui donne le type d'une variable dans une liste donnée, et l'opérateur *ityped* dit si une variable est typée ou non dans une liste de typage.

$$tvar(v, tp) = v \in itype(v, tp)$$
$$tvar(v \cup V, tp) = v \in itype(v, tp)\ ;\ tvar(V, tp),\ V \neq \varnothing$$

$$itype(s, s \in F) = F$$
$$itype(s, P; Q) = itype(s, P) \text{ si } ityped(s, P)$$
$$itype(s, P; Q) = itype(s, Q) \text{ si } ityped(s, Q)$$

$$ityped(s, s \in F) = true$$
$$ityped(x, s \in F) = false \text{ si } x \neq s$$
$$ityped(s, P; Q) = ityped(s, P) \vee ityped(s, Q)$$

14.2 Syntaxe du langage de référence

La syntaxe du langage de référence est organisée en clauses, comme le langage BHDL. Il y a une clause DESIGN qui spécifie le nom du circuit qui est décrit (ce nom est censé être unique et permet de désigner la description lors d'une importation), une autre clause, IMPORTS, donne la liste des descriptions importées, encore une autre clause, USES, donne une liste de nom de machines déclaratives (comme dans le cas du langage BHDL) qui définissent des constantes et des types. Ensuite, il y a une clause INPUTS qui donne la liste des entrées et une autre clause OUTPUTS qui donne la liste des sorties. Une différence avec BHDL est que les listes des entrées et des sorties sont des listes de typage (listes constituées des noms des variables, associés avec leurs types), nous n'avons donc pas de clause INVARIANT comme en BHDL.

Contrairement au langage BHDL, les variables sont séparées en deux groupes. D'une part les registres, et d'autre part les signaux. La clause REGISTERS donne la liste des registres et la façon dont ils évoluent au passage d'un cycle d'horloge à l'autre, et de quelle façon ils sont initialisés. La clause SIGNALS donne la liste de typage (nom + types) des signaux. La partie combinatoire du circuit est décrite dans la clause DO. Elle s'exprime par une liste de blocs BEGIN ...END. Chaque bloc a sa propre liste d'entrées et de sorties, qui correspondent respectivement aux signaux qui sont lus et écrits par le bloc.

Une grammaire du langage de référence est donnée sur la figure 14.1. Nous ne redonnons pas le sous-langage des expressions et des identificateurs qui est le même que pour le langage BHDL.

Le langage de référence propose deux façons de spécifier un registre, selon que le registre ait une initialisation asynchrone ou non. Les deux formes possibles sont données ci-dessous.

PRE $r \in F$ POST r' DO $r \longleftarrow E$ END or
PRE $r \in F$ POST r' END

La première forme signifie que le nom du registre est r, son type est F et ce registre prend la valeur de r' au front montant de l'horloge. Lorsque le signal de réinitialisation asynchrone est activé, le registre est initialisé à la valeur E. La seconde forme est le cas dans lequel il n'y a pas d'initialisation asynchrone du registre (la registre n'est pas sensible au signal de réinitialisation asynchrone). A l'intérieur de la partie combinatoire (clause DO), les identificateurs r et r' sont considérés comme des signaux mais ils ne sont pas déclarés dans la clause SIGNALS.

La construction $x \longleftarrow E$ est l'équivalent de la substitution $x := E$ du langage BHDL et la construction $x \longleftarrow y$ WHEN g OTHERWISE z est l'équivalent de la substitution conditionnelle de BHDL. Dans le langage de référence, l'opérateur parallèle $\|$ n'existe pas explicitement mais la construction $A; B$ dans un bloc n'est pas une composition séquentielle mais une mise en concurrence : A et B sont concurrents

FIG. 14.1 – Langage de référence (HDL)

HDL	: :=	DESIGN *ident* IMPORTS *limport* USES *lident* INPUTS *ltvar* OUTPUTS *ltvar* REGISTERS *lreg* SIGNALS *ltvar* DO *C*
C	: :=	*C*; *C* | BEGIN INPUTS *ltvar* OUTPUTS *ltvar* DO *D* END | BEGIN INPUTS *ltvar* OUTPUTS *ltvar* SIGNALS *ltvar* DO *D* END | BEGIN INPUTS *ltvar* OUTPUTS *ltvar* IMPORTS *import* DO *lport* END
D	: :=	*C* | *var* ⟵ *exp* | *var* ⟵ *var* WHEN *var* OTHERWISE *var*
lreg	: :=	*lreg*; *reg* | *reg*
reg	: :=	PRE *tvar* POST *var* DO *var* ⟵ *exp* END | PRE *tvar* POST *var* END
lvar	: :=	*lvar* , *var* | *var*
ltvar	: :=	*ltvar*; *tvar* | *tvar*
lexp	: :=	*lexp* , *exp* | *exp*
lident	: :=	*lident* , *ident* | *ident*
var	: :=	*ident*
tvar	: :=	*ident* ∈ *ident*
limport	: :=	*limport*, *import* | *import*
import	: :=	*ident* : *ident*
exp	: :=	...
ident	: :=	...

(modèle du parallélisme généralement utilisé dans les langages de description de circuits). Par ailleurs, les blocs spécifiés dans la liste de blocs de la clause DO sont des blocs concurrents.

La partie combinatoire (clause DO) est composée d'une liste de blocs concurrents. La description d'un bloc peut suivre l'une des trois syntaxes suivantes :

BEGIN INPUTS *i* OUTPUTS *o* DO *T* END	BEGIN INPUTS *i* OUTPUTS *o* SIGNALS *S* DO *T* END	BEGIN INPUTS *i* OUTPUTS *o* IMPORTS *ic* : *Comp* DO *Lconnect* END

La première syntaxe signifie que le bloc est sensible aux signaux i et modifie les signaux o. La façon dont il modifie les signaux o en fonction des signaux i est décrite par l'ensemble d'affectations concurrentes T. En utilisant la deuxième syntaxe, on peut utiliser les signaux locaux S à l'intérieur du bloc (comme

signaux intermédiaires par exemple). Ces signaux ne sont pas visibles de l'extérieur du bloc. La troisième syntaxe sert à importer des descriptions extérieures. Le nom *Comp* est le nom d'une description existante, cette description est instanciée sous le nom *ic*. Le contenu *Lconnect* du bloc est alors uniquement constitué des listes de connexions, pour les entrées et les sorties. Les listes de signaux i et o correspondant alors aux signaux qui sont connectés respectivement aux entrées et aux sorties du composant importé.

14.3 Ensembles supports dans le langage de référence

Comme pour le langage BHDL, nous définissons des ensembles supports en lecture et en écriture pour le langage de référence. Ici les définitions sont plus simples puisque chaque bloc a une liste de ses entrées et de ses sorties, qui correspondent respectivement aux ensembles supports en lecture et en écriture. Par ailleurs, toutes les affectations sont concurrentes, il n'y a donc pas de composition (comme les compositions séquentielles ou conditionnelles) qui peuvent modifier les ensembles supports.

Ensemble support en écriture L'ensemble support en écriture d'un ensemble d'affectations concurrentes est l'ensemble des signaux qui sont modifiés par les affectations.

$write(x \longleftarrow E) \equiv \{x\}$

$write(A; B) \equiv write(A) \cup write(B)$

$write(x \longleftarrow y \text{ WHEN } g \text{ OTHERWISE } z) \equiv \{x\}$

$$write\begin{pmatrix}\text{BEGIN}\\ \text{INPUTS } i\\ \text{OUTPUTS } o\\ \text{DO}\\ T\\ \text{END}\end{pmatrix} \equiv var(o) \qquad write\begin{pmatrix}\text{BEGIN}\\ \text{INPUTS } i\\ \text{OUTPUTS } o\\ \text{SIGNALS } S\\ \text{DO}\\ T\\ \text{END}\end{pmatrix} \equiv var(o) \qquad write\begin{pmatrix}\text{BEGIN}\\ \text{INPUTS } i\\ \text{OUTPUTS } o\\ \text{IMPORTS}\\ ic : Comp\\ \text{DO}\\ Lconnect\\ \text{END}\end{pmatrix} \equiv var(o)$$

Ensemble support en lecture

$read(Exp) \equiv$ variables libres de Exp

$read(x \longleftarrow E) \equiv read(E)$

$read(A; B) \equiv read(A) \cup read(B)$

$read(x \longleftarrow y \text{ WHEN } g \text{ OTHERWISE } z) \equiv read(y) \cup read(z) \cup read(g)$

$$read\begin{pmatrix}\text{BEGIN}\\ \text{INPUTS } i\\ \text{OUTPUTS } o\\ \text{DO}\\ T\\ \text{END}\end{pmatrix} \equiv var(i) \qquad read\begin{pmatrix}\text{BEGIN}\\ \text{INPUTS } i\\ \text{OUTPUTS } o\\ \text{SIGNALS } S\\ \text{DO}\\ T\\ \text{END}\end{pmatrix} \equiv var(i) \qquad read\begin{pmatrix}\text{BEGIN}\\ \text{INPUTS } i\\ \text{OUTPUTS } o\\ \text{IMPORTS}\\ ic : Comp\\ \text{DO}\\ Lconnect\\ \text{END}\end{pmatrix} \equiv var(i)$$

14.4 Sémantique du langage de référence

La sémantique du langage de référence est donnée par un prédicat définissant un ensemble de fonctions de traces, comme la sémantique prédicative de BHDL. Nous notons cette sémantique de traces du langage de référence par $[\![\cdot]\!]$. Comme pour BHDL, nous définissons des prédicats *avant-après* pour la partie combinatoire de la description, le prédicat *avant-après* est noté $|(\cdot)|$.

Dans ce langage, la description du circuit est faite par une collection de blocs concurrents. Chaque bloc est une affectation de signal, ou une importation d'une description existante de circuit.

bloc : BEGIN INPUTS i OUTPUTS o SIGNALS s DO D END

Cette collection de blocs est la description de la partie combinatoire du circuit. Les registres sont séparés de la partie combinatoire, ils sont mis dans la clause REGISTERS de la description. Cette clause est une collection de structures de la forme suivante :

PRE $r \in F$ POST r' DO $r \longleftarrow E$ END, ou
PRE $r \in F$ POST r' END s'il n'y a pas d'initialisation

où $r \longleftarrow E$ est l'initialisation du registre r lorsque le signal de réinitialisation asynchrone est activé.

14.4.1 Sémantique de trace d'une description HDL

Toutes les descriptions HDL sont supposées se trouver dans un espace commun de descriptions ICS et les noms des descriptions dans un espace commun de noms $ICNS$. L'opérateur $IComp$ associe à chaque nom la description HDL correspondante.

$$IComp \in ICNS \leftrightarrow ICS$$

Nous nous dotons de quelques opérateurs nous permettant d'extraire de l'information d'une description HDL à partir du nom de la description. Nous définissons les opérateurs $ILReg$, $IReg$, $ILSig$ et IOp. L'opérateur $ILReg$ donne la liste des registres de la description alors que $IReg$ extrait directement le contenu de la clause REGISTERS. De la même façon, l'opérateur $ILSig$ donne la liste des signaux et $ISig$ renvoie le contenu de la clause SIGNALS. Pour extraire la liste des registres et des signaux à partir des clauses REGISTERS et SIGNALS, nous utilisons les opérateurs $Lreg$ et $Lsig$. L'opérateur IOp renvoie le contenu de la clause DO, c'est-à-dire la collection de blocs décrivant la partie combinatoire du circuit.

$$IComp(D) = \begin{cases} \text{DESIGN } D \\ \text{USES } param \\ \text{IMPORTS } imp \\ \text{INPUTS } i \\ \text{OUTPUTS } o \\ \text{REGISTERS } Reg \\ \text{SIGNALS } Sig \\ \text{DO } C \end{cases} \Rightarrow \begin{cases} IReg(D) = Reg \\ ISig(D) = Sig \\ IOp(D) = C \\ ILReg(D) = Lreg(IReg(D)) \\ ILSig(D) = Lsig(ISig(D)) \end{cases}$$

Les opérateurs $Lreg$ et $Lsig$ sont définis de la manière suivante.

$$\begin{array}{l} Lreg(A;B) = Lreg(A) \cup Lreg(B) \\ Lreg(\text{PRE } r \in F \text{ POST } r' \text{ DO } r \longleftarrow E \text{ END}) = \{r\} \\ Lreg(\text{PRE } r \in F \text{POST } r' \text{ END}) = \{r\} \end{array}$$

$$\begin{array}{l} Lsig(A;B) = Lsig(A) \cup Lsig(B) \\ Lsig(s \in T) = \{s\} \end{array}$$

La sémantique d'une description HDL est donnée par un prédicat qui définit une fonction associant les traces des sorties aux traces des entrées. La forme de la sémantique est la même que la sémantique prédicative du langage BHDL.

$$\left|\left[\begin{array}{l} \text{DESIGN } D \\ \text{USES } param \\ \text{IMPORTS } imp \\ \text{INPUTS } i \\ \text{OUTPUTS } o \\ \text{REGISTERS } Reg \\ \text{SIGNALS } Sig \\ \text{DO } C \end{array}\right]\right| (f) \equiv \begin{cases} \text{let} \\ \quad R = ILReg(D) \\ \quad S = ILSig(D) \\ \quad IR = IRenamedImportedReg(D) \\ \quad IS = IRenamedImportedSig(D) \\ \quad in = var(i) \\ \quad out = var(o) \\ \quad as = S \cup IS \\ \quad ar = R \cup IR \\ \quad tp = i; o; Sig \\ \text{in} \\ \quad \forall in^\omega = (in_0, in_1, ...) \exists out^\omega = (out_0, out_1, ...) \\ \quad f(in^\omega) = out^\omega \wedge \\ \quad \exists S^\omega = (S_0, S_1, ...), R^\omega = (R_0, R_1, ...) \\ \quad \exists IS^\omega = (IS_0, IS_1, ...), IR^\omega = (IR_0, IR_1, ...) \\ \quad \circledast \end{cases}$$

$$\circledast \equiv \begin{cases} |[Reg]| \ \wedge \ \bigwedge_{s \in S \cup in \cup out} s_0 \in itype(s, tp) \ \wedge \\ \bigwedge_{(n,C) \in imported(D)} \left(\begin{array}{l} \text{let} \\ \quad iReg = IReg(C) \\ \quad rg = ILReg(C) \\ \quad iSig = ISig(C) \\ \quad sg = ILSig(C) \\ \text{in} \\ \quad [rg_0 := n.rg_0] \, |[iReg]| \wedge \\ \quad \bigwedge_{s \in sg} n.s_0 \in itype(s, iSig) \end{array} \right) \\ \forall k \geq 0 \ \begin{bmatrix} ar', ar, & & ar_{k+1}, ar_k, \\ as', as, & := & as_k, as_k, \\ in, out', out & & in_k, out_k, out_k \end{bmatrix} |(C)| \end{cases}$$

14.4.2 Sémantique de traces de la clause REGISTER

Dans le cas où un registre est sensible au signal de réinitialisation asynchrone, une initialisation est donnée dans la clause DO de la description du registre. Dans ce cas, la sémantique consiste en un prédicat qui spécifie le premier élément de la trace du registre ($r_0 = E$ lorsque l'initialisation est $r \longleftarrow E$). Si le registre n'est pas sensible au signal de réinitialisation, le prédicat spécifie simplement le type du registre.

- $\left| \begin{bmatrix} \text{PRE } r \in F \\ \text{POST } r' \\ \text{DO } r \longleftarrow E \\ \text{END} \end{bmatrix} \right| \equiv r_0 = E$

- $\left| \begin{bmatrix} \text{PRE } r \in F \\ \text{POST } r' \\ \text{END} \end{bmatrix} \right| \equiv r_0 \in F$

La sémantique de l'ensemble des registres est la conjonction des prédicats de chaque registre.

- $|[S;T]| \ \equiv \ |[S]| \ \wedge \ |[T]|$

14.4.3 Sémantique *avant-après* de la partie combinatoire

Comme pour le langage BHDL, la sémantique de la partie combinatoire est d'abord donnée par un prédicat *avant-après*, ce prédicat étant par la suite élevé au rang de prédicat sur les traces par la définition de la sémantique de la description HDL (cf. la sémantique ci-dessus).

En comparaison avec la sémantique du langage BHDL, nous ajoutons aux prédicats *avant-après* des conditions de bonne formation sur la description HDL. Ces conditions de bonne formation consistent à spécifier que les variables qui sont spécifiées comme des entrées (clause INPUTS d'un bloc) soient les variables qui sont lues par le bloc, et de la même façon que les sorties soient les variables qui sont écrites par le bloc. Les prédicats *avant-après* de chaque constructeur du langage de référence apparaissant dans la partie combinatoire sont listés ci-dessous.

- Le cas le plus simple d'un bloc est celui d'un bloc qui n'utilise pas de signaux locaux. Le prédicat *avant-après* est alors le prédicat *avant-après* de la description contenue dans le bloc.

$$\left| \begin{pmatrix} \text{BEGIN} \\ \text{INPUTS } i \\ \text{OUTPUTS } o \\ \text{DO} \\ \quad S \\ \text{END} \end{pmatrix} \right| \equiv |(S)| \ \wedge \ var(i) = read(S) \ \wedge \ var(o) = write(S)$$

- Le cas suivant est le cas plus avancé d'un bloc qui peut utiliser des signaux locaux. Dans le prédicat *avant-après*, les signaux locaux sont introduits par une quantification existentielle.

$$\left|\left(\begin{array}{l}\text{BEGIN}\\ \text{INPUTS } i\\ \text{OUTPUTS } o\\ \text{SIGNALS } S\\ \text{DO}\\ \quad T\\ \text{END}\end{array}\right)\right| \equiv \exists S \cdot |(T)| \ \wedge var(i) = read(T) - var(S) \wedge var(o) = write(T) - var(S)$$

- Dans le cas où le bloc consiste à importer une description HDL existante, le prédicat *avant-après* est le prédicat *avant-après* de la partie combinatoire de la description importée. Dans ce prédicat *avant-après*, les entrées et les sorties doivent être remplacées par les variables locales auxquelles elles sont connectées. Les autres variables (c'est à dire les signaux qui ne sont pas des entrées ou des sorties, et les registres) sont renommées en leur ajoutant le nom d'instance de la description importée.

$$\left|\left(\begin{array}{l}\text{BEGIN}\\ \text{INPUTS } i\\ \text{OUTPUTS } o\\ \text{IMPORTS } ic : Comp\\ \text{DO}\\ \quad Lconnect\\ \text{END}\end{array}\right)\right| \equiv \left\{\begin{array}{l}\text{let}\\ \quad vc = ILReg(Comp) \cup ILSig(Comp)\\ \quad is = IRenamedImportedSig(Comp)\\ \quad ir = IRenamedImportedReg(Comp)\\ \quad av = vc \cup is \cup ir\\ \quad (p, l) = PortLoc(Lconnect)\\ \text{in}\\ \quad \left[\begin{array}{l}p, p',\\ av, av'\end{array} := \begin{array}{l}l, l',\\ ic.av, ic.av'\end{array}\right] |(IOp(Comp))|\end{array}\right.$$

- Le prédicat *avant-après* d'une affectation concurrente est un prédicat qui spécifie que la valeur du signal est la valeur à laquelle l'expression qui lui est affectée est évaluée.

$$|(x \longleftarrow E)| \ \equiv \ x' = E$$

- Dans le cas où il s'agit d'un affectation décrivant un multiplexeur, le prédicat *avant-après* est la conjonction des deux prédicats des deux alternatives, chacun étant gardé par la condition ou la négation de la condition de choix d'une des deux possibilités.

$$|(x \longleftarrow y \text{ WHEN } g \text{ OTHERWISE } z)| \ \equiv \ g \Rightarrow |(x \longleftarrow y)| \ \wedge \ \neg g \Rightarrow |(x \longleftarrow z)|$$

- Lorsqu'on a affaire à une liste d'affectations concurrentes, ou une liste de blocs concurrents, le prédicat *avant-après* est la conjonction des prédicats *avant-après*.

$$|(S;T)| \ \equiv \ |(S)| \ \wedge \ |(T)|$$

14.5 Règles de traduction d'un modèle BHDL vers une description HDL

Section retirée à la demande de KeesDA pour des raisons de confidentialité. On donne ci-dessous le théorème principal de cette section.

Propriété 21 (Correction de la traduction) *L'opérateur de traduction trad défini précédemment préserve la sémantique. La sémantique de la description HDL obtenue par traduction est la même que la sémantique du modèle BHDL original. Formellement, pour tout modèle BHDL D :*

$$||[trad(D)]|| \ \equiv \ [|D|]$$

14.6 Conclusion

Nous avons exposé dans ce chapitre les règles de traduction de BHDL vers d'autres langages de description de circuits. Plutôt que de montrer la traduction vers un langage donné, nous avons défini un

langage de référence, à partir duquel il est assez simple de passer à VHDL ou SystemC par exemple. Nous avons donné une sémantique à ce langage et nous avons montré que les règles de traduction que nous avons données préservent bien la sémantique du modèle.

Le traducteur BHDL® développé par KeesDA utilise des règles similaires mais possède des règles d'optimisation supplémentaires.

Quatrième partie

Travaux connexes

Chapitre 15

Réutilisation de composants avec ACL2

La démarche présentée dans cette thèse est de partir de la spécification en langage naturel, d'en faire une abstraction qui est ensuite raffinée pas à pas jusqu'à obtenir un modèle implantable, modèle décrit dans le langage BHDL.

Dans ce chapitre, nous suggérons une approche intermédiaire (voir la figure 15.1) en spécifiant le circuit en B (en fait dans le sous langage BHDL) à un niveau d'abstraction où l'interface du circuit modélisé correspond à l'interface de l'IP à vérifier. Ensuite, le modèle BHDL est traduit vers ACL2. Dans le même temps, la description VHDL de l'IP est également traduite vers ACL2 [88] et nous utilisons ACL2 pour prouver l'équivalence entre les deux modèles.

Ce chapitre décrit la traduction de B vers ACL2. L'étape principale est l'aplatissement du modèle B. Cela consiste à construire un modèle B dans lequel l'évolution de chaque variable du modèle soit spécifiée en utilisant uniquement des références aux entrées ou aux registres du circuit, sans n'utiliser aucune variable intermédiaire. Ceci permet la construction d'un modèle compact similaire à celui utilisé en ACL2 pour un circuit VHDL. L'avantage de faire cette étape sur le modèle B, et non pas sur un modèle ACL2 qui aurait été obtenu autrement, est que le processus d'aplatissement est un raffinement au sens de B. En fait, le modèle aplati est même équivalent au modèle original.

Ce travail a été fait en collaboration avec Diana Toma du laboratoire IMAG-TIMA et a donné lieu à une publication dans le cadre de la conférence ZB2005 [94].

15.1 Traduction de BHDL vers ACL2

Un modèle BHDL se compose en deux parties principales : la clause INITIALISATION qui spécifie comment les registres sont initialisés lorsque le signal *reset* du circuit est actif, et la clause OPERATIONS qui spécifie la partie combinatoire du circuit.

La description d'un circuit en ACL2 que nous utilisons dans notre approche est constituée d'une

FIG. 15.1 – Utilisation d'ALC2 pour la réutilisation d'IP en B

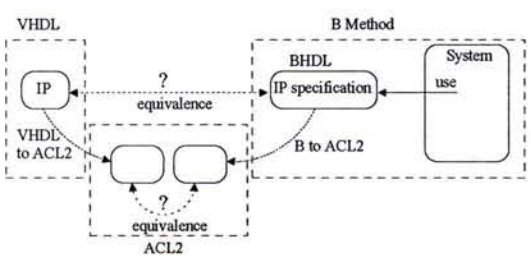


FIG. 15.2 – (a) Modèle BHDL d'un compteur 3-bits; (b) Mélange des clauses INITIALISATION et OPERATIONS.

(a)

```
INITIALISATION
    compt := 0||
    rst :∈ BOOL||
    alm :∈ BOOL
OPERATIONS
    alm := bool(compt = 7)
    ;
    IF rst = true THEN
        compt := 0
    ELSE
        IF alm = false THEN
            compt := compt + 1
        END
    END
```

(b)

```
IF reset = true THEN
    compt := 0
END
;
alm := bool(compt = 7)
;
IF rst = true THEN
    compt := 0
ELSE
    IF alm = false THEN
        compt := compt + 1
    END
END
```

fonction par signal, fonction qui calcule la valeur du signal en fin de cycle d'horloge en fonction des entrées et des registres du circuit. Une fonction supplémentaire est dédiée à la simulation d'un cycle d'horloge en appelant toutes les fonctions des signaux.

Le processus de traduction de BHDL vers ACL2 est le suivant :

1. Le modèle est converti dans un modèle dans lequel la clause INITIALISATION et la clause OPERATIONS sont réunies en une seule substitution. Un modèle dont la clause INITIALISATION est *Init* et la clause OPERATIONS est *Op* est transformé vers la substitution ci-dessous. De plus, les substitutions non déterministes de la substitution *Init* sont supprimées.

   IF *reset* = *true* THEN Init END ; Op

   Le signal d'entrée *reset* est introduit explicitement : si le signal *reset* est actif alors les substitutions *Init* et *Op* sont appliquées. Dans le cas contraire, seule la substitution *Op* est appliquée. Ceci correspond à la sémantique d'un modèle BHDL : l'état du circuit entre l'initialisation *Init* et la première application de *Op* n'est pas observable. Nous donnons sur la figure 15.2b l'événement correspondant à la description BHDL de la figure 15.2a : le signal *reset* est rendu explicite et l'initialisation est directement introduite dans l'événement.

   Notons qu'une hypothèse de modélisation est que le signal *reset* est actif lorsque le système démarre et qu'il demeure inactif ensuite.

2. Aplatissement de la substitution résultant de l'étape précédente.
3. Traduction de la substitution aplatie en ACL2.

15.1.1 Forme aplatie d'une substitution

La substitution séquentielle produit des résultats intermédiaires pouvant être réutilisés ensuite dans des expressions. Le modèle ACL2 est fonctionnel et, dans notre approche, les sorties sont des fonctions des entrées et des registres, sans utiliser aucune variable intermédiaire. Pour générer un modèle ACL2, nous commençons par *aplatir* le modèle BHDL en retirant les substitutions séquentielles. Par exemple, la substitution $x := in + z; out := x + 1$ est d'abord transformée en $x := in + z \| out := in + z + 1$.

Dans la définition d'une substitution aplatie, nous nous référons uniquement aux substitutions utilisées en BHDL. Une substitution est *aplatie* lorsque les conditions suivantes sont remplies :

- elle ne contient aucune composition séquentielle,

FIG. 15.3 – Substitution plate correspondant au compteur

```
IF reset = true THEN
    alm := bool(0 = 7)
ELSE
    alm := bool(compt = 7)
END

||

IF reset = true THEN
   IF rst = true THEN
      compt := 0
   ELSE
      IF bool(0 = 7) = false THEN
         compt := 0 + 1
      ELSE
         compt := 0
      END
   END
ELSE
   IF rst = true THEN
      compt := 0
   ELSE
      IF bool(compt = 7) = false THEN
         compt := compt + 1
      ELSE
         compt := compt
      END
   END
END
```

- il s'agit d'un composition parallèle de substitutions, chacune d'elles ne modifiant qu'une seule variable. Deux de ces substitutions ne peuvent pas modifier les mêmes variables et toutes les variables doivent être écrites par l'une des substitutions.

Notons qu'aucune de ces substitutions composées en parallèle ne peut contenir elle-même de composition parallèle puisqu'une seule variable doit être écrite. Elle ne peut pas non plus contenir de composition séquentielle. Donc, par rapport au langage BHDL une telle substitution est nécessairement un arbre (se réduisant éventuellement à une feuille) de IF imbriqués avec des substitutions simples de la forme $v := E$ comme feuilles.

On peut formaliser ceci en donnant la grammaire suivante, où *BoolExp* est la grammaire des prédicats, *Exp* la grammaire des expressions et *var* la grammaire des identificateurs. Le prédicat $card(W_S) = 1$ est une condition de bonne formation pour assurer que chaque substitution de la composition parallèle ne modifie qu'une seule variable. En particulier, dans le cas de la substitution conditionnelle, les substitutions $S^{(1)}$ et $S^{(2)}$ doivent écrire la même variable. La condition requise que deux substitutions ne doivent pas écrire la même variable est assurée par la condition de bonne formation traditionnelle de la composition parallèle.

$$
\begin{array}{rcll}
\mathit{FlatS} & \leftarrow & S & card(W_S) = 1 \\
 & & \mid \mathit{FlatS} \parallel \mathit{FlatS} & \\
S & \leftarrow & \text{IF } \mathit{BoolExp} \text{ THEN } S^{(1)} \text{ ELSE } S^{(2)} \text{ END} & \\
 & & \mid \mathit{var} := \mathit{Exp}. &
\end{array}
$$

Exemple d'une substitution aplatie

Pour illustrer comment un modèle BHDL est transformé, nous donnons sur la figure 15.3 la forme plate du compteur donné comme exemple sur la figure 15.2. Il s'agit de deux substitutions composées en parallèle. La première spécifie l'évolution de la variable *alm* et la seconde de la variable *compt*. Chacune de ces substitutions ne dépend que des entrées (*reset* and *rst*) et des registres (*compt*). En particulier, l'expression de *compt* ne dépend plus de la valeur de la variable *alm*.

15.1.2 Traduction d'une substitution plate en ACL2

La troisième étape est relativement simple une fois l'aplatissement du modèle effectué. Elle consiste simplement à réécrire les substitutions en utilisant la syntaxe d'ACL2. La syntaxe de la substitution

FIG. 15.4 – Modèle ACL2 du compteur

```
(defun B_alm (compt reset)
   (if (equal reset 1)
      (if (equal 0 7) 1 0)
         (if (equal compt 7) 1 0))
(defun B_compt (compt rst reset)
   (if (equal reset 1)
      (if (equal rst 1)
         0
         (if (equal (equal 0 7) nil) (+ 0 1) 0 )
      )
      (if (equal rst 1)
         0
         (if (equal (equal compt 7) nil) (+ compt 1) compt )
      )))
```

respecte la grammaire donnée dans la section précédente. La traduction vers ACL2 se fait simplement en utilisant l'opérateur *acl2* défini ci-dessous. Cet opérateur ne s'applique que sur des substitutions plates.

Les substitutions $S_1 ... S_n$, S et T se réfèrent aux substitutions qui ne contiennent aucune composition parallèle. Ce sont des substitutions simples ou des substitutions conditionnelles plates. Nous utilisons la notation u_k pour dénommer la variable écrite par la substitution S_k, cela est utilisé pour donner un nom à la fonction ACL2 qui est créée (B_u_k).

$acl2(S_1 \| ... \| S_n) =$

pour chaque substitution S_k, cette fonction ACL2 est créée :

(defun B_u_k $acl2(S_k)$)

où $\{u_k\} = W_S$, u_k est la variable écrite par S_k

$acl2(\text{IF } C \text{ THEN } S \text{ ELSE } T \text{ END}) = (\text{if } acl2(C)\ acl2(S)\ acl2(T))$

$acl2(v := E) = (\ acl2exp(E)\ )$

où $acl2exp$ est la traduction des expression B en expression ACL2.

La traduction du compteur donné dans les sections 1.8.3 et 15.1.1 produit les fonctions ACL2 données sur la figure 15.4.

15.2 Aplatissement

Aplatir une substitution S consiste à construire une nouvelle substitution qui a le même effet que S mais qui est plate. La transformation principale est l'enlèvement des compositions séquentielles $(S;T)$ en propageant les effets de la première substitution (S) dans la seconde (T). Nous donnons ci-dessous les définitions de règles d'aplatissement, une preuve de correction de ces règles est donnée en annexe D.

15.2.1 Règles d'aplatissement

Le processus d'aplatissement est basé sur trois opérateurs. L'opérateur principal *flat* aplatit une substitution. Il utilise les deux opérateurs *extract* et *integrate*. L'opérateur $extract(v, S)$ donne une substitution qui a le même effet que S sur la variable v et qui a exactement v comme ensemble support en écriture. L'opérateur $integrate(S, T)$ intègre la substitution S dans la substitution T : si la substitution T lit une variable v qui est écrite par S, elle lit la variable v telle qu'elle est après l'application de S.

Dans cette section, nous utilisons la notation $\|_{v \in E} S(v)$ pour noter la composition parallèle des substitutions $S(v)$ pour chaque variable v qui se trouve dans l'ensemble de variables E. Par exemple, si $E = \{v_1, ..., v_k\}$ alors $\|_{v \in E} S(v) = S(v_1) \| ... \| S(v_k)$.

Aplatissement d'une substitution

Une substitution simple $v := E$ est déjà plate et une composition parallèle est plate si les deux substitutions composées sont plates.

$$\begin{aligned} flat(v := E) &= v := E \\ flat(A \| B) &= flat(A) \| flat(B) \end{aligned}$$

Dans une composition conditionnelle, les deux substitutions alternatives peuvent écrire plusieurs variables. Une substitution conditionnelle plate n'écrit qu'une seule variable. Par conséquent, la règle de transformation crée une substitution conditionnelle pour chaque variable écrite et les compose en parallèle ($\|_{v \in W_A \cup W_B}$). Dans l'expression $extract(v, flat(A)))$, $flat(A)$ est plate, il s'agit au pire d'une composition parallèle de substitutions, chacune d'elle n'écrivant qu'une seule variable. L'opérateur *extract* sélectionne parmi elles celle qui écrit la variable v.

$$\begin{aligned} &flat(\text{IF } C \text{ THEN } A \text{ ELSE } B \text{ END}) = \\ &\quad \|_{v \in W_A} \text{IF } C \text{ THEN } extract(v, flat(A)) \text{ ELSE } extract(v, flat(B)) \text{ END} \end{aligned}$$

$$\begin{aligned} &flat(\text{IF } C \text{ THEN } A \text{ END}) = \\ &\quad \|_{v \in W_A \cup W_B} \text{IF } C \text{ THEN } extract(v, flat(A)) \text{ ELSE } v := v \text{ END} \end{aligned}$$

La composition séquentielle n'existe pas dans la forme plate d'une substitution. Elle doit être transformée en une substitution plate équivalente. Ceci est fait par propagation des transformations spécifiées par la première substitution dans la seconde.

Le principe pour aplatir une substitution $S;T$ est le suivant. Pour toute variable v écrite par S et lue par T, la valeur de v utilisée par T est remplacée dans T par la valeur de v après l'application de S, i.e. v est remplacée dans T par l'expression spécifiée par S. Par exemple, l'intégration de $x := E; y := 6$ dans $x := x + 1$ donne la substitution $x := E + 1$. À cette intégration, il faut ensuite ne pas oublier de remettre en parallèle les substitutions plates de S correspondant aux variables écrites par S et pas par T. Dans notre petit exemple, l'aplatissement de $x := E; y := 6; x := x + 1$ donne la substitution $y := 6 \parallel x := E + 1$.

L'intégration est réalisée par l'opérateur *integrate*, qui est défini par la suite. Il retourne une substitution plate qui a le même ensemble support en écriture que T. Cela signifie que les variables qui sont écrites par S mais pas par T ne sont pas écrites par le résultat de l'intégration. Nous ajoutons donc la substitution plate $S_{/W_S - W_T}$ qui a le même comportement que $flat(S)$ sur les variables de $W_S - W_T$ et pour laquelle l'ensemble support en écriture est exactement $W_S - W_T$ (voir l'opérateur *extract* ci-dessous).

$$flat(S;T) = S_{/W_S - W_T} \| integrate(flat(S), flat(T))$$

Extraction d'une substitution

L'opérateur $extract(v, S)$ retourne une substitution qui a le même effet que la substitution S sur la variable v et qui a exactement $\{v\}$ comme ensemble support en écriture. L'opérateur *extract* est ici défini uniquement sur des substitutions plates. Ceci simplifie les définitions car une substitution plate est au plus une composition parallèle de substitutions, chacune d'entre elles n'écrivant qu'une seule variable. Ainsi, l'extraction consiste simplement à rechercher la substitution correspondant à la variable v que l'on souhaite extraire.

$$\begin{aligned} extract(v, S) &= v := v \quad \text{si } v \notin W_S \\ extract(v, S) &= v := v \quad \text{si } W_S = \{v\} \\ extract(v, S_1 \| S_2) &= extract(v, S_1) \quad \text{si } v \in W_{S_1} \\ extract(v, S_1 \| S_2) &= extract(v, S_2) \quad \text{si } v \in W_{S_2} \end{aligned}$$

Intégration

L'opérateur $integrate(S, T)$ propage les effets de la substitution S dans la substitution T : si une variable est modifiée par S et utilisée par T, la substitution T est transformée de façon à utiliser la valeur que la variable a après l'application de S. L'opérateur *integrate* est ici défini seulement sur des substitutions plates, ceci permet des simplifications dans les définitions ci-dessous.

Il découle de cette définition que si nous intégrons une substitution qui écrit certaines variables v dans une deuxième substitution qui ne lit pas v du tout, alors l'intégration n'a aucun effet. Par exemple, intégrer $x := 2$ dans $x := y$ produit la substitution $x := y$.

$$integrate(S, T) = T \text{ si } W_S \cap R_T = \varnothing$$

Intégrer une substitution simple consiste à appliquer cette substitution à toutes les expressions. Par exemple, intégrer la substitution $x := 2$ dans $x := x+1$ produit la substitution $x := 2+1$. On remarquera que la partie gauche de la deuxième substitution n'est pas affectée par l'intégration. Par contre, toute les parties droites sont affectées. En particulier, lorsque la deuxième substitution est une composition conditionnelle, on applique également la substitution à la condition de la composition conditionnelle.

$$integrate(x := E, y := F) \equiv y := [x := E]F$$

$$integrate(x := E, \text{ IF } P \text{ THEN } S \text{ ELSE } T \text{ END})$$
$$\equiv \text{IF } [x := E]P \text{ THEN } integrate(x := E, S) \text{ ELSE } integrate(x := E, T) \text{ END}$$

L'intégration d'une composition conditionnelle est la composition conditionnelle de l'intégration de chaque alternative.

$$integrate(\text{IF } P \text{ THEN } S \text{ ELSE } T \text{ END}, G)$$
$$\equiv \text{IF } P \text{ THEN } integrate(S, G) \text{ ELSE } integrate(T, G) \text{ END}$$

L'intégration d'une substitution S dans une substitution qui est une composition parallèle consiste simplement à intégrer S dans chacune des substitutions composées en parallèle. Par exemple, l'intégration de $x := 2$ dans $x := x + 1 || y := x - 2$ produit la substitution $x := 2 + 1 \| y := 2 - 2$.

$$integrate(S, A \| B) = integrate(S, A) \| integrate(S, B)$$

L'intégration d'une substitution parallèle $A \| B$ dans une substitution T consiste à intégrer A et B dans T. Cependant, comme des variables modifiées par B peuvent être lues par A, et vice versa, on ne peut pas intégrer d'abord A puis B. Par exemple, si B contient $x := E$ et A contient $y := x$, l'intégration de $A \| B$ dans $z := y + x$ produit la substitution $z := x + E$. Si nous intégrons d'abord A, nous obtenons $z := x + x$; en intégrant B à la suite de A nous obtenons alors $z := E + E$, ce qui n'est pas le résultat recherché. Un exemple de ce que nous recherchons est l'intégration de $S \equiv x := y + 1 \| \text{IF } x = y \text{ THEN } y := x + 2 \text{ ELSE } y := 1 \text{ END}$ dans la substitution $T \equiv x := x + y$. Le résultat est la substitution $\text{IF } x = y \text{ THEN } x := y + 1 + x + 2 \text{ ELSE } x := y + 1 + 1 \text{ END}$.

Il s'agit du problème de remplacer une substitution parallèle par une suite de substitutions. Pour résoudre ce problème, nous nous y prenons comme lorsque l'on souhaite inverser la valeur de deux variables : en utilisant une troisième variable. Pour des raisons de confidentialité, nous ne donnons pas la règle que nous avons utilisée.

Une autre méthode consiste à se ramener à des cas plus simples en utilisant les règles de transformation données dans le BBook [1] page 310 pour transformer $A \| B$ en une substitution dans laquelle seulement des substitutions simples sont composées en parallèle ($x := E \| y := F$). Le résultat de cette transformation n'est pas une substitution plate (puisque des compositions conditionnelles pourront écrire plusieurs variables). Il faudra donc adapter les règles données ici pour des substitutions non plates, en particulier l'opérateur *extract* qui est très simplifié sur les substitution plates.

Simplifications

Les règles d'aplatissement données ci-dessus créent beaucoup de compositions conditionnelles imbriquées. Ceci est dû au fait qu'en B, excepté dans les formes plates, une composition conditionnelle peut contenir plusieurs substitutions simples ($v := E$). L'opérateur d'aplatissement *flat* les éclate en plusieurs compositions conditionnelles et l'opérateur *integrate* les éparpille et les imbrique les unes dans les autres. Le résultat est une substitution qui grossit exponentiellement.

Pour simplifier ces substitutions, les branches inutiles des compositions conditionnelles sont coupées et les compositions peuvent être simplifiées lorsque deux branches sont égales. La substitution S^s (resp. T^s) est le résultat de la simplification de S (resp. T).

$simpl(C, NC,$ IF P THEN S ELSE T END) =
si $P \in C$ alors $simpl(C, NC, S)$
sinon si $P \in NC$ alors $simpl(C, NC, T)$
sinon
Soit $S^s = simpl(C \cup \{P\}, NC, S)$ et $T^s = simpl(C, NC \cup \{P\}, T)$ dans
si $S^s = T^s$ alors S^s
sinon IF P THEN S^s ELSE T^s END

Le paramètre C est l'ensemble des conditions qui sont connues pour être vraies et NC l'ensemble des conditions qui sont connues pour ne pas être vraies. Ces ensembles proviennent du fait que la substitution qui est en train d'être simplifiée se situe quelque-part sur une branche d'un arbre de compositions conditionnelles. Initialement $C = NC = \varnothing$.

Implantation

Les règles d'aplatissement définies ci-dessus ont été implantées en Prolog. Les expérimentations faites ont montré l'utilité de la règle de simplification. Sans elle, les substitutions construites pendant le processus d'aplatissement sont de plus en plus imposantes. Pour être utile, la règle de simplification doit être appliquée régulièrement pendant le processus d'aplatissement, ou être directement intégrée dans les règles d'aplatissement.

Sans simplification, l'aplatissement d'un modèle BHDL d'environ 200 lignes a nécessité plus de 100Mo de RAM et le processus prend plusieurs heures pour arriver à terme. Avec la règle de simplification appliquée régulièrement, l'aplatissement du même modèle prend environ une seconde.

15.2.2 L'aplatissement est un raffinement

Le processus d'aplatissement est un raffinement. En fait, le résultat de l'aplatissement est une substitution qui est équivalente à la substitution originale.

Théorème 2 (L'aplatissement donne une substitution équivalente) *Pour toute substitution S du langage BHDL et tout prédicat Q,*

$$[flat(S)]Q \Leftrightarrow [S]Q$$

La preuve est donnée dans l'annexe D

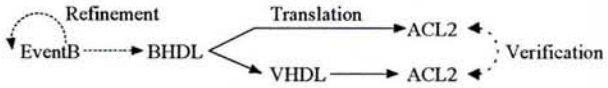
15.3 Études de cas

Nous présentons deux études de cas. La première illustre la méthodologie sur un exemple non trivial, la seconde utilise l'exemple du compteur pour expliquer le processus de vérification en ACL2.

15.3.1 Contrôleur d'un bus série

La première étude de cas concerne un contrôleur de bus série (standard SAE J1708 [83]) : plusieurs composants reliés par un bus en série peuvent envoyer des messages aux autres composants en utilisant le bus. Chaque composant possède un contrôleur qui a la charge d'envoyer les messages bit par bit sur

FIG. 15.5 – Connexion des traducteurs



le bus et de gérer les conflits d'accès au bus (lorsque deux contrôleurs tentent d'accéder au bus en même temps).

Le développement B [93] a commencé par la modélisation du système complet à un niveau très abstrait où les propriétés importantes du système ont été introduites. Le processus de raffinement proposé par la méthode B a été utilisé pour dériver une implantation prouvée du protocole et du contrôleur jusqu'au niveau transfert de registres (RTL). Ce niveau correspond au langage BHDL, à partir duquel le circuit a été traduit vers VHDL puis synthétisé. On se référera au chapitre 7 de ce document pour plus de détails.

Notre but ici n'est plus de valider la description VHDL (puisqu'elle est correcte par construction) mais de confirmer la méthodologie expliquée dans ce chapitre sur un exemple non trivial que nous savons déjà être correct. C'est aussi l'occasion d'associer trois traducteurs développés séparément. Nous sommes partis de la description BHDL du circuit validé par la méthode B. Cette description a été traduite deux fois :

- une première fois en ACL2, comme cela a été expliqué dans ce chapitre,
- et une deuxième fois en VHDL, en utilisant le traducteur développé par KeesDA. Le code VHDL fait environ 400 lignes et utilise 140 signaux internes.

Ensuite, la description VHDL a été traduite en ACL2 en utilisant le traducteur développé par le laboratoire IMAG-TIMA.

Nous obtenons donc en fin de compte deux descriptions ACL2 provenant du même BHDL mais obtenues de deux façons différentes. Notre but est de confirmer que ces deux descriptions sont bien équivalentes. Les trois traducteurs sont basés sur trois approches différentes et ont été validés séparément. Le fait que la preuve d'équivalence puisse se faire facilement donne une confiance supplémentaire dans l'implantation de ces traducteurs et confirme notre méthodologie consistant à utiliser ACL2 comme un intermédiaire entre VHDL et B.

Le modèle ACL2 obtenu à partir du VHDL possède 148 fonctions et le modèle ACL2 provenant directement de la description BHDL possède 21 fonctions. Pour la preuve d'équivalence, nous avons défini 65 théorèmes, et nous avons utilisé une bibliothèque préexistante sur les vecteurs de bits et les opérations sur les vecteurs de bits. Le processus de preuve n'a pas été difficile, il a pris quelques heures pour arriver à terme (contre plusieurs semaines pour le développement B originel). La preuve elle-même prend 17,23 minutes sur un processeur Ultra Sparc3, 1.28 GHz, avec 8Go de mémoire.

Les modèles ont été modifiés à la main pour introduire des erreurs, en particulier sur les types et les expressions arithmétiques. Ceci a permis de vérifier que des erreurs non purement fonctionnelles peuvent aussi être détectées par ce processus, comme l'incompatibilité de l'implantation de certaines données (par exemple, un entier qui peut être évalué à 8 ne peut pas être implanté par un vecteur de 3 bits). Les erreurs ont été détectées car certaines conjectures se sont montrées fausses dans le système ACL2.

15.3.2 Compteur

Nous avons appliqué la vérification ACL2 à l'exemple du compteur (cf. section 1.8). Après traduction du modèle BHDL en ACL2, les fonctions sim-step, system, hyp-input and hyp-st ont été définies. Nous donnons ci-dessous les fonctions sim-step et system correspondant au modèle BHDL du compteur.

```
(defun b-sim-step (in st)
  (let ((reset (nth *b-reset* in)) (rst (nth *b-rst* in)) (compt (nth *b-compt* st)))
    (list (B_compt compt rst reset)
          (B_alm compt reset))))
(defun b-counter (input st)
  (if (atom input) st (b-counter (cdr input) (b-sim-step (car input) st))))
```

Nous avons deux modèles ACL2 : un correspondant au modèle VHDL (cf. figure 1.3 page 17) et l'autre au modèle BHDL (cf. figure 15.4 page 250). Les deux modèles sont des circuits synchrones et définis au niveau du cycle d'horloge (niveau transfert de registres, RTL).

Pour prouver la relation de bisimulation entre les deux modèles, un relation $Sim \subseteq ST_{VHDL} \times ST_{BHDL}$ est d'abord définie, où ST_{VHDL} est l'ensemble des états du modèle provenant de VHDL, et ST_{BHDL} est l'ensemble des états du modèle provenant de BHDL. La preuve que Sim est une relation de simulation est faite en deux étapes :

1. en partant d'un état similaire arbitraire, après un cycle d'horloge et lorsque *reset* vaut 1, les deux modèles sont dans un état similaire selon la relation Sim ;
2. en partant d'un état similaire arbitraire, *st-b* et *st-vhdl*, où (*st-vhdl*, *st-b*) $\in Sim$, après avoir consommé les mêmes entrées (et en prenant en compte les conversions de types nécessaires), les deux modèles sont dans un état similaire :

   $$(vhdl\text{-}system(inputs, st\text{-}vhdl),\ b\text{-}system(inputs, st\text{-}b)) \in Sim$$

   Ceci est prouvé par induction sur le nombre de cycles d'horloge, i.e. la longueur de la liste d'entrées. Le cas de base est que la fonction sim-step préserve la similarité.

Une seconde relation $Sim^{-1} \subseteq ST_{BHDL} \times ST_{VHDL}$ est définie et prouvée être l'inverse de Sim. De la même façon que cela a été fait pour Sim, Sim^{-1} est prouvée être une relation de simulation entre le modèle BHDL et le modèle VHDL.

Finalement, Sim est prouvée être une relation de bisimulation entre les modèles BHDL et VHDL.

Pour l'exemple du compteur, la relation Sim est définie de la façon suivante :

$$(st\text{-}vhdl, st\text{-}b) \in Sim \Leftrightarrow ((alm3 = alm) \wedge (get\text{-}1\text{-}pos(tc\_1) = compt))$$

Où $st\text{-}vhdl = (tc\_0, tc\_1, tc\_6, tc\_8, tc\_10, gd, rst, alm3)$ et $st\text{-}b = (compt,\ alm)$. La fonction *get-1-pos* prend un vecteur de bit en entrée et retourne la position de '1' dans le vecteur. Par exemple, $get\text{-}1\text{-}pos($ $(00100)) = 2$

La relation Sim^{-1} est définit de la façon suivante :

$$(st\text{-}b, st\text{-}vhdl) \in Sim^{-1} \Leftrightarrow ((alm = alm3) \wedge (tc\_1 = construct\text{-}table(compt)))$$

La fonction *construct-table* prend un entier n en entrée et retourne un vecteur de bit de taille 8 avec '1' à la $n$ième position, tous les autres bits sont à '0'.

La preuve utilise les bibliothèques ACL2 sur les entiers et les listes incluses dans la distribution publique du prouveur de théorèmes. Elle utilise également une bibliothèque sur les vecteur de bits qui a été définie précédemment pour la vérification de systèmes électroniques.

15.4 Conclusion

Nous avons présenté une nouvelle méthodologie permettant la réutilisation de composants qui n'ont pas été développés par la méthode B.

Le principe consiste à écrire une spécification du composant en B (en fait BHDL) et prouver que cette spécification correspond au composant en utilisant ACL2. Pour faire cela, les deux descriptions, BHDL d'une part et VHDL d'autre part, du composant sont traduites vers ACL2. Le prouveur de théorèmes ACL2 est utilisé pour prouver que les deux modèles sont équivalents (en fait qu'il existe une bisimulation entre les deux modèles).

La traduction de BHDL vers ACL2 a besoin d'aplatir le modèle BHDL. Les règles de traduction et de d'aplatissement ont été expliquées et nous avons prouvé que l'aplatissement conduit à un modèle équivalent à l'original.

La méthodologie a été appliquée sur un exemple non trivial pour vérifier son efficacité. Cela a également été l'occasion de combiner trois traducteurs qui ont été développés séparément et avec des approches différentes. Les expérimentations ont montré que la non équivalence de modèle est détectée par cette méthodologie.

Chapitre 16

Modèles transactionnels

SystemC est une librairie C++ destinée à la modélisation de systèmes électroniques. On parlera aussi du « langage » SystemC, constitué du langage C++ augmenté des primitives définies par SystemC. Comme tout programme C++, une description SystemC peut être compilée pour être exécutée. L'exécution d'un modèle SystemC correspond à la simulation du modèle SystemC du système électronique. Le langage SystemC est maintenu et standardisé par l'OSCI (Open SystemC Initiative) qui propose également une implantation open source de la librairie (aussi appelée simulateur). A l'origine, le langage SystemC dispose des principales primitives permettant de modéliser des circuits à des niveaux d'abstraction relativement bas, auxquelles on peut ajouter du traitement de plus haut niveau en utilisant le langage C++. Depuis peu, le langage SystemC a été augmenté par des primitives de plus haut niveau permettant de modéliser l'échange de données entre les modules du système par des transactions évitant ainsi d'avoir à modéliser explicitement les protocoles de bas niveau. Ce niveau de modélisation par des transactions est désigné par l'acronyme TLM (Transaction Level Model). Non seulement ceci simplifie la modélisation du système mais permet également une simulation plus rapide du système. Cette simulation plus rapide est utile pour appliquer un nombre imposant de vecteurs de tests, mais facilite également le développement des logiciels qui seront embarqués sur le système électronique modélisé. De plus, un modèle TLM est plus rapide à développer qu'un modèle de bas niveau, permettant ainsi de développer le logiciel en parallèle du développement du modèle synthétisable. Ceci permet d'économiser sur le temps de mise sur le marché. Un défaut actuel de cette démarche est de garantir que le modèle synthétisable à partir duquel sera fabriqué le système corresponde au modèle TLM utilisé pour développer le logiciel. Nous proposons d'utiliser la méthode B pour raffiner le modèle TLM formellement jusqu'au niveau synthétisable. La correspondance entre les deux modèles est alors garantie. Ce chapitre montre comment on peut modéliser dans le langage B les concepts de la modélisation TLM.

16.1 Concepts du niveau de modélisation TLM

L'originalité du niveau de modélisation appelé TLM (Transaction Level Model) est de proposer une manière abstraite de modéliser les échanges de données entre les différents modules composant un système. Au niveau de modélisation TLM, les échanges de données se font en utilisant des transactions. Suivant les fonctionnalités des modules, une transaction peut provoquer l'envoi d'une autre transaction par le module cible vers un module tierce. Une transaction est initiée par un module et reçue par un autre. Lorsqu'elle est initiée, une transaction porte des données qui sont fournies par l'initiateur au module cible. Dans le cas général, une transaction convoie également des données du module cible vers le module initiateur ; ces données constituent la réponse du module cible à la transaction. On peut distinguer dans le système des modules maîtres qui initient les transactions, et des modules esclaves qui sont chargés de répondre à ces transactions. Un même module peut être à la fois maître et esclave dans le sens où il est possible qu'un module puisse à la fois envoyer des transactions et en recevoir. Des modules d'un autre type, appelés « channel », peuvent être utilisés comme intermédiaire pour faire transiter les transactions d'un module à un autre. D'un premier abord un channel peut être vu comme une modélisation abstraite d'un bus,

mais il s'agit en fait d'un module pouvant contenir toute autre fonctionnalité comme un module SystemC en général. Au niveau TLM, le channel étant chargé de passer des transactions, il doit être capable de recevoir des transactions de la part de modules initiateurs et de fournir aux modules cibles le moyen d'obtenir les données portées par les transactions qui lui sont destinées. Le channel doit également être capable d'acheminer, le cas échéant, les réponses des modules cibles vers les modules initiateurs.

Une transaction peut se présenter sous différents formats selon la nature de la transaction et des données envoyées, et attendues en réponses, portées par la transaction. Le format que doit suivre une transaction est exprimé par une « interface ». En SystemC, une interface est une liste de noms de méthodes avec leurs signatures. Un module capable de recevoir des transactions selon une interface donnée doit implanter cette interface, c'est-à-dire définir les méthodes déclarées par l'interface. Un module peut implanter plusieurs interfaces, acceptant ainsi plusieurs types de transactions. Selon les concepts de la modélisation TLM, le module initiateur envoie une transaction en utilisant un « port » (et non en appelant directement la méthode concernée du module cible). De la même manière, le module cible reçoit les transactions sur un port spécial appelé « export ». Le modèle du système doit indiquer quels ports sont connectés à quels exports. Un module initiateur peut posséder plusieurs ports, lui permettant d'envoyer des transactions vers différents modules. De la même façon, un module cible peut posséder plusieurs exports, pour recevoir des transactions de plusieurs modules. A chaque port (et chaque export), est associé l'interface qui est utilisée pour les transactions. Un module initiateur peut donc disposer de plusieurs ports de différentes interfaces pour envoyer des transactions dans des formats différents. De la même façon, un module cible peut avoir des exports d'interfaces différentes pour recevoir différents types de transactions.

Lorsqu'un module initiateur veut envoyer une transaction vers un module cible mais que ces deux modules ont des interfaces différentes, on peut utiliser un module appelé « adaptateur ». Ce module est chargé de recevoir une transaction de la part du module initiateur en utilisant l'interface de l'initiateur et de passer cette transaction vers le module cible en utilisant l'interface du module cible. L'adaptateur a un rôle de traducteur dans le sens de l'envoie de la transaction, mais également dans le sens de réponse du module cible vers le module initiateur.

16.2 Notations ASCII du langage B

Dans ce chapitre nous présentons les modèles B en notation ASCII. Nous présentons dans le tableau ci-dessous les correspondances entre les notations ASCII et les notations mathématiques.

Notation mathématique	Notation ASCII	Signification
$\wedge$	`&`	"et" logique
$\times$	`*`	produit cartésien
$\widehat{=}$	`==`	définition
$\in$	`:`	appartenance à un ensemble
$x :\in E$	`x :: E`	substitution "x devient un élément de l'ensemble E"
$a \mapsto b$	`a |-> b`	couple (a,b)
$\geq$	`>=`	supérieur ou égal
$\rightarrow$	`-->`	fonction totale

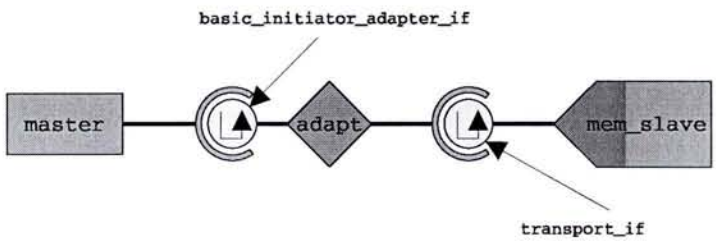
16.3 Exemple fourni par l'OSCI

Afin de montrer comment les concepts fondamentaux de la modélisation TLM en SystemC peuvent être modélisés dans le langage B, nous montrons la modélisation d'un exemple fourni par l'OSCI (Open SystemC Initiative, responsable de la standardisation de SystemC). Cet exemple simple reprend tous les concepts fondamentaux de la modélisation TLM. Nous montrons également de quelle manière ce modèle B peut être raffiné vers un nouveau modèle B plus concret, toujours au niveau de modélisation TLM.

16.3.1 Description de l'exemple

Le système est composé de deux modules principaux : un maître et un esclave. Le maître (master) envoie des transactions de lecture ou d'écriture vers un module esclave représentant une mémoire. L'esclave (mem_slave) reçoit des transactions et y répond en modifiant son état interne (lorsqu'il reçoit une requête d'écriture) ou en renvoyant au maître les données demandées (lorsqu'il reçoit une requête de lecture). Dans cet exemple, le maître ne s'adresse pas à l'esclave directement mais par l'intermédiaire d'un adaptateur qui transforme les transactions émises par le maître dans un format particulier vers des transactions conformes à celles attendues par l'esclave. Dans l'exemple initial fourni par l'OSCI, l'adaptateur n'était pas présent explicitement mais était un morceau de code à l'intérieur du module maître. L'analyse de l'architecture du SystemC par les concepts de la modélisation TLM a montré que ce morceau de code est en fait un adaptateur entre la façon dont le maître envoie des transactions et la façon dont l'esclave reçoit les transactions. Nous donnons sur la figure 16.1 une représentation graphique de l'architecture du système.

FIG. 16.1 – Architecture du système transactionnel



L'interface « basic_initiator_adapter_if » décrit le format des transactions qui se déroulent entre le maître et l'adaptateur, et l'interface « transport_if » décrit le format des transactions entre l'adaptateur et l'esclave.

En SystemC, l'interface « basic_initiator_adapter_if » est composée de deux méthodes : une méthode « read » qui permet au maître d'envoyer une requête de lecture et une méthode « write » qui permet au maître de générer une requête d'écriture. L'interface « transport_if » est composée d'une seule méthode « transport ». L'adaptateur transforme les transactions envoyées par le maître en utilisant les méthodes « read » et « write » vers des transactions utilisant la méthode « transport ».

Les transactions sont envoyées en utilisant des « ports » et reçues par l'intermédiaire de ports spéciaux appelés « exports » en SystemC. Les ports et les exports doivent être « liés » pour pouvoir communiquer.

16.3.2 Modèle B événementiel de l'exemple 3.2

Déclarations des ports et des interfaces

Dans le modèle B, nous déclarons les ports comme des variables et les exports sont définis comme des alias renvoyant vers les ports auxquels ils sont connectés. Les interfaces sont modélisées comme des ensembles définis en extension dont les membres sont les noms des méthodes fournies par l'interface. A chaque interface est associé un type correspondant aux données qui peuvent être envoyées et reçues par le biais des ports implantant ces interfaces.

Prenons d'abord l'exemple du lien de communication existant entre l'adaptateur et l'esclave. L'adaptateur envoie les transactions en utilisant un port que nous nommerons « transport_port », ces transactions sont reçues par l'esclave via un export nommé « target_port ». Ces ports implantent l'interface « transport_if_types ».

L'interface « transport_if » ne contient qu'une seule méthode, appelée « transport », nous définissons l'ensemble « transport_if_types » dans la clause SETS de modèle B de la façon suivante :

```
SETS
   transport_if_types = {transport_if_transport};
```

Ceci signifie que l'interface ne contient qu'une seule méthode appelée « transport ». A cette interface on associe le type « transport_if_type » correspondant au type de la méthode « transport » de l'interface « transport_if » et le type « port_transport_if » correspondant au type d'un port implantant l'interface « transport_if ». Les types ADDRESS et DATA sont les types utilisés respectivement pour représenter les adresses et les données dans cet exemple. Le type basic_request correspond au type de données envoyées par l'adaptateur à l'esclave par une transaction par la méthode transport, et le type basic_response correspond à la réponse faite par l'esclave.

```
SETS
   basic_request_type = {br_READ, br_WRITE};
   basic_status = {ERROR, SUCCESS};

CONSTANTS
   transport_if_type,
   transport_if
   basic_request,
   basic_response,
   ADDRESS,
   DATA

PROPERTIES

   ADDRESS = NATURAL &
   DATA = NATURAL &
   basic_request = basic_request_type * ADDRESS * DATA &
   basic_response = basic_status * basic_request_type * DATA &

   transport_if_type =
        basic_request *
        basic_response &

   port_transport_if =
        port_status *
        transport_if_types *
        transport_if_type &
```

Le type « transport_if_type » est constitué simplement des types de données transitant par la transaction. Le type « port_transport_if », correspondant au type des ports implantant cette interface, contient en plus un champ de type « port_status » contenant l'état dans lequel se trouve le port à un moment donné et un champ de type « transport_if_types » déjà défini plus haut, identifiant la méthode (et donc le type de transaction) que le port est en train de traiter. L'état du port est modélisé par un ensemble défini en extension et contient les deux valeurs « port_pending » pour dire qu'une transaction est en cours sur ce port et « port_free » pour dire que le port est libre et peut être utilisé pour initier une transaction.

```
SETS
   port_status = {port_pending, port_free};
```

Le port (appelé « transport_port ») lui-même est déclaré comme une variable de type « port_transport_if » et l'export (appelé « target_port ») de l'esclave correspondant est déclaré comme un alias vers ce port.

```
DEFINITIONS
   target_port == transport_port
VARIABLES
   transport_port
INVARIANT
   transport_port : port_transport_if
```

On opère de la même façon pour le lien de communication entre le maître et l'adaptateur. Cette fois-ci l'interface contient les deux méthodes « read » et « write ». Le port du maître s'appelle « initiator_port » et l'export de l'adaptateur s'appelle « in_port_adapt ».

```
DEFINITIONS
   in_port_adapt == initiator_port
SETS
   basic_initiator_adapter_if_types = {
             basic_initiator_adapter_read,
             basic_initiator_adapter_write}
CONSTANTS
   port_transport_if
PROPERTIES
   basic_initiator_adapter_if_type =
        basic_status *
        ADDRESS *
        DATA &
   port_basic_initiator_adapter_if =
        port_status *
        basic_initiator_adapter_if_types *
        basic_initiator_adapter_if_type

VARIABLES
   initiator_port
INVARIANT
   initiator_port : port_basic_initiator_adapter_if
```

Comme toute variable en B, les ports doivent être initialisés. Ils le sont en positionnant leur statut à « port_free ».

```
INITIALISATION
   initiator_port ::
         {port_free} *
         basic_initiator_adapter_if_types *
         basic_initiator_adapter_if_type ||

   transport_port ::
         {port_free} *
         transport_if_types *
         transport_if_type
```

Modélisation des transactions

L'envoie d'une transaction se modélise par deux événements : un événement qui se produit avant la transaction et un événement qui se produit après. L'événement qui se produit avant positionne l'état du port concerné à « port_pending », indique quelle méthode est utilisée pour la transaction et écrit les données à envoyer. L'événement qui se produit après se déclenche lorsque l'état du port est redevenu « port_free » et récupère éventuellement les données en réponse à la transaction et poursuit l'exécution du module.

Ci-dessous les événements correspondant au module maître. Celui-ci fait une première boucle dans laquelle il envoie des transactions d'écriture et ensuite une deuxième boucle dans laquelle il envoie des transactions de lecture.

```
VARIABLES
   a, d, mst_st
INVARIANT
   mst_st : 0..6 &
   a : ADDRESS &
   d : DATA

/* Initialisation de la boucle d'écritures */
master_init_loop_write =
   SELECT
      mst_st = 0
```

```
   THEN
      a := 0 ||
      mst_st := 1
   END;

/* Envoie d'une transaction d'écriture */
master_loop_write_bfw =
   ANY st WHERE
      mst_st = 1 &
      st : basic_status
   THEN
      initiator_port :=
         port_pending |->
         basic_initiator_adapter_write |->
         (st|->a|->(a+50)) ||
      mst_st := 2
   END;

/* Après la transaction d'écriture lorsque la boucle n'est pas terminée (a < 20) */
master_loop_write_afw_1 =
   SELECT
      mst_st = 2 & a < 20 &
      initiator_port :
           {port_free} *
           basic_initiator_adapter_if_types *
           basic_initiator_adapter_if_type
   THEN
      mst_st := 1 ||
      a := a + 1
   END;

/* Après la transaction d'écriture lorsque la boucle n'est pas terminée (a >= 20) */
master_loop_write_afw_2 =
   SELECT
      mst_st = 2 & a >= 20 &
      initiator_port :
           {port_free} *
           basic_initiator_adapter_if_types *
           basic_initiator_adapter_if_type
   THEN
      mst_st := 3
   END;

/* Initialisation de la boucle de lectures */
master_init_loop_read =
   SELECT
      mst_st = 3
   THEN
      a := 0 ||
      mst_st := 4
   END;

/* Envoie d'une transaction de lecture*/
master_loop_read_bfr =
   ANY dt, st WHERE
      mst_st = 4 &
      st : basic_status &
      dt : DATA
```

```
   THEN
      initiator_port :=
           port_pending|->
           basic_initiator_adapter_read|->
           (st|->a|->dt) ||
      mst_st := 5
   END;

/* Après la transaction de lecture lorsque la boucle n'est pas terminée */
master_loop_read_afr_1 =
   ANY dt, ad, st, tp WHERE
      mst_st = 5 & a < 20 &
      tp : basic_initiator_adapter_if_types &
      dt : DATA &
      ad : ADDRESS &
      st : basic_status &
      initiator_port = port_free|->tp|->(st|->ad|->dt)
   THEN
      d := dt ||
      a := a + 1||
      mst_st := 4
   END;

/* Après la transaction de lecture lorsque la boucle est terminée */
master_loop_read_afr_2 =
   ANY dt, ad, st, tp WHERE
      mst_st = 5 & a >= 20 &
      tp : basic_initiator_adapter_if_types &
      dt : DATA &
      ad : ADDRESS &
      st : basic_status &
      initiator_port = port_free|->tp|->(st|->ad|->dt)
   THEN
      d := dt ||
      mst_st := 6
   END;

/* fin du processus maître */
master_stop =
   SELECT
      mst_st = 6
   THEN
      skip
   END;
```

Ces transactions sont traitées par l'adaptateur qui renvoie des transactions vers l'esclave.

Le traitement d'une transaction par le module cible se fait par un événement dont la garde est sensible à la condition « statut du port = port_pending ». Après avoir traité la transaction, le module cible positionne le statut du port à « port_free ». Dans le cas de l'adaptateur, celui-ci renvoie une transaction à chaque fois qu'il en reçoit une. Pour chacune des transactions reçues, il y a donc deux événements. Le premier événement se déclenche lorsqu'il reçoit une transaction du maître et est chargé de renvoyer une transaction vers l'esclave. Le deuxième événement est l'événement qui se déclenche lorsque la transaction envoyée vers l'esclave est terminée et est chargé de terminer la transaction reçue par le maître. L'adaptateur pouvant recevoir deux types de transactions (transactions « read » et transactions « write »), le modèle de l'adaptateur est constitué des quatre événements suivants.

```
VARIABLES
   adpt_st, adpt_ad, adpt_dt
INVARIANT
   adpt_st : 0..1 &
```

```
   adpt_ad : ADDRESS &
   adpt_dt : DATA


/* Reçoit une transaction « write » du maître */
basic_initiator_adpater_write_1 =
   ANY st, ad, dt, brs WHERE
      adpt_st = 0 &

      dt : DATA &
      ad : ADDRESS &
      st : basic_status &
      in_port_adapt =
           port_pending |->
           basic_initiator_adapter_write |->
           (st|->ad|->dt) &

      brs : basic_response
   THEN
      adpt_ad := ad ||
      adpt_dt := dt ||
      /* envoie une transaction d'écriture vers l'esclave*/
      transport_port :=
           port_pending |->
           transport_if_transport |->
           ((br_WRITE|->ad|->dt)|->brs) ||
      adpt_st := 1
   END;


/* La transaction d'écriture envoyée vers l'esclave est terminée */
basic_initiator_adpater_write_2 =
   ANY tp, brq, st, dt WHERE
      adpt_st = 1 &

      tp : transport_if_types &
      brq : basic_request &
      st :  basic_status &
      dt : DATA &
      transport_port =
           port_free |->
           tp |->
           (brq|->(st|->br_WRITE|->dt))
   THEN
      /* termine la transaction « write » provenant du maître */
      in_port_adapt :=
           port_free |->
           basic_initiator_adapter_write |->
           (st|->adpt_ad|->adpt_dt) ||
      adpt_st := 0
   END;

/* Reçoit une transaction « read » du maître */
basic_initiator_adpater_read_1 =
   ANY st, ad, dt, brs WHERE
      adpt_st = 0 &
```

```
      dt : DATA &
      ad : ADDRESS &
      st : basic_status &
      in_port_adapt =
           port_pending |->
           basic_initiator_adapter_read |->
           (st|->ad|->dt) &

      brs : basic_response
   THEN
      adpt_ad := ad ||
      adpt_dt := dt ||
      /* envoie une transaction de lecture vers l'esclave*/
      transport_port :=
            port_pending |->
            transport_if_transport |->
            ((br_READ|->ad|->dt)|->brs) ||
      adpt_st := 1
   END;


/* La transaction de lecture envoyée vers l'esclave est terminée */
basic_initiator_adpater_read_2 =
   ANY tp, brq, st, dt WHERE
      adpt_st = 1 &
      tp : transport_if_types &
      brq : basic_request &
      st :  basic_status &
      dt : DATA &
      transport_port =
             port_free |->
             tp |->
             (brq|->(st|->br_READ|->dt))
   THEN
      /* termine la transaction «read» provenant du maître */
      in_port_adapt :=
             port_free |->
             basic_initiator_adapter_read |->
             (st|->adpt_ad|->adpt_dt) ||
      adpt_st := 0
   END;
```

L'esclave reçoit des transactions de la part de l'adaptateur. S'il s'agit d'une transaction de lecture, l'esclave met la donnée demandée à l'intérieur de la transaction de retour. Si la transaction est une transaction d'écriture, l'esclave modifie sa mémoire « memory » en utilisant les données contenues dans la transaction.

```
VARIABLES
   memory
INVARIANT
   memory : ADDRESS --> DATA


mem_slave_transport =
  ANY ad, dt, tp, brs WHERE
      ad : ADDRESS &
      dt : DATA &
      tp : basic_request_type &
      brs : basic_response &
      target_port =
```

```
            port_pending |->
            transport_if_transport |->
            ((tp|->ad|->dt)|->brs)
  THEN


   IF tp=br_WRITE THEN
     memory(ad) := dt ||
     target_port :=
            port_free |->
            transport_if_transport |->
            ((tp|->ad|->dt)|->(SUCCESS|->br_WRITE|->dt))

     ELSIF tp=br_READ THEN
     target_port :=
            port_free |->
            transport_if_transport |->
            ((tp|->ad|->dt)|->(SUCCESS|->br_READ|->memory(ad)))

     END
  END
```

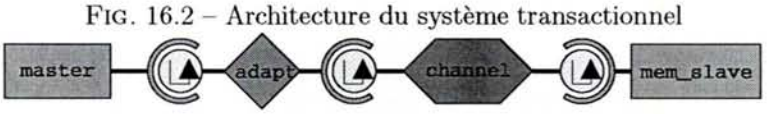
16.4 Raffinement de l'exemple

L'intérêt d'utiliser la méthode B pour modéliser les systèmes au niveau TLM est d'utiliser le mécanisme de raffinement formel proposé par la méthode B. Nous montrons dans cette section comment B permet de raffiner le modèle TLM décrit dans la section précédente vers un nouveau modèle TLM plus concret.

16.4.1 Architecture du système raffiné

Ce raffinement ajoute un module de communication (appelé channel en SystemC) entre le maître (en fait l'adaptateur) et l'esclave. Le module esclave devient initiateur de transactions vers le channel. Le channel reçoit les transactions envoyées par le master et qui étaient auparavant reçues par l'esclave. Le channel reçoit également des transactions de l'esclave. Les transactions envoyées par l'esclave au channel lui servent à récupérer les transactions qui ont été envoyées par le maître et qui sont maintenant traitées par le channel. La structure du nouveau système est représentée graphiquement sur le figure 16.2.

FIG. 16.2 – Architecture du système transactionnel



16.4.2 Modèle B événementiel du raffinement

Il y a deux nouvelles interfaces pour le lien entre l'esclave et le channel. Chacune de ces interfaces ne contient qu'une seule méthode. L'interface « tlm_bget_if_types » contient la méthode « tlm_bget » qui sert à l'esclave à récupérer une transaction à partir du channel. L'interface « tlm_bput_if_types » contient la méthode « tlm_bput » qui sert à l'esclave à renvoyer sa réponse vers le channel (qui est chargé de la renvoyer en direction du maître).

```
SETS
   tlm_bget_if_types = {tlm_bget};
   tlm_bput_if_types = {tlm_bput}
CONSTANTS
   tlm_bget_if_type,
   port_tlm_bget_if,
```

```
   tlm_bput_if_type,
   port_tml_bput_if

PROPERTIES
   tlm_bget_if_type = basic_request &

   port_tml_bget_if =
        port_status *
        tlm_bget_if_types *
        tlm_bget_if_type &


  tlm_bput_if_type = basic_response &

   port_tml_bput_if =
      port_status *
        tlm_bput_if_types *
        tlm_bput_if_type
```

Les modèles du maître et de l'adaptateur sont inchangés, nous ne les rappelons pas ici (cf. section précédente).

Le channel reçoit d'un côté des transactions de l'adaptateur via son export target_port. Cet export est en fait celui de l'esclave du modèle précédent, le raffinement remplace l'esclave par un channel et un nouvel esclave qui ne reçoit pas directement des transactions de l'adaptateur. L'export de l'esclave se retrouve donc dans le channel. Le comportement du channel lorsqu'il reçoit une transaction venant de l'adaptateur est de sauvegarder les requêtes reçues dans son état interne, en l'occurrence une FIFO. La modélisation de la FIFO se fait en utilisant les séquences du langage B. Le comportement lié à ce port est modélisé par deux événements. Le premier événement « tlm_req_rsp_transport_1 » est chargé de recevoir les transactions et de sauvegarder les requêtes dans la FIFO « request_fifo ». Le deuxième événement « mem_slave_transport » attend ensuite qu'une deuxième FIFO « response_fifo » ne soit pas vide et renvoie la réponse à la transaction initiale en envoyant la réponse se situant dans « response_fifo ». Ce deuxième événement est le raffinement de l'événement portant le même nom du modèle de la section précédente.

```
DEFINITIONS
   get_request_export == in_port;
   put_response_export == out_port

VARIABLES
   req_rsp_rst,
   request_fifo,
   response_fifo

INVARIANT
   req_rsp_rst : 0..1 &
   request_fifo : seq(basic_request)&
   response_fifo : seq(basic_response)


tlm_req_rsp_transport_1 =
   ANY req, rsp WHERE
      req_rsp_rst = 0 &

      req : basic_request &
      rsp : basic_response &

      target_port =
         port_pending |->
         transport_if_transport |->
         (req |-> rsp)
   THEN
```

```
      request_fifo := req -> request_fifo ||
      req_rsp_rst := 1
   END;



mem_slave_transport =
   SELECT
      response_fifo /= {} &
      req_rsp_rst = 1
   THEN
     response_fifo := tail(response_fifo) ||
     ANY req WHERE
        req : basic_request &
        target_port :
                 port_status*
                 transport_if_types*
                 ({req}*basic_response)
      THEN


      target_port :=
              port_free |->
              transport_if_transport |->
              (req|->first(response_fifo))
      END ||

      req_rsp_rst := 0
   END;
```

La FIFO « resquest_fifo » est consommée lorsque l'esclave envoie une transaction « tlm_bget » au channel, dans ce cas le channel renvoie la transaction présente dans la FIFO à l'esclave. Ceci est modélisé par l'événement « get_request_fifo ». Lorsque l'esclave donne sa réponse en utilisant la méthode tlm_bput (événement « put_response_fifo »), la réponse est mise dans la FIFO « response_fifo » et le channel est alors en mesure de renvoyer la réponse à la transaction initiale du maître.

```
get_request_fifo =
   SELECT
      get_request_export :
            {port_pending} *
            {tlm_bget} *
            basic_request &
      request_fifo /= {}
   THEN
      request_fifo := tail(request_fifo) ||
      get_request_export :=
            port_free |->
            tlm_bget |->
            first(request_fifo)
   END;


put_response_fifo =
   ANY rsp WHERE
      rsp : basic_response &
      put_response_export =
          port_pending |->
          tlm_bput |->
```

```
        rsp &
     response_fifo = {}
  THEN
     response_fifo := rsp -> response_fifo ||
     put_response_export ::
        {port_free} *
        {tlm_bput} *
        basic_response
  END;
```

Dans ce raffinement du système l'esclave ne reçoit plus lui-même de transaction mais génère des transactions vers le channel. On montre ci-dessous le modèle de l'esclave, qui est une mémoire. Le modèle est basé sur une machine d'état. Son rôle est d'envoyer des transactions au channel pour récupérer des requêtes. Lorsque la requête est une requête d'écriture, la mémoire modifie son état interne en conséquence. La mémoire est ici représentée par la variable « memory_fs » qui est un raffinement de la variable « memory » du modèle précédent (il est nécessaire de raffiner cette variable car la modification de la mémoire interne ne se fait plus dans le même événement).

```
VARIABLES
   memory_fs,
   in_port,
   out_port,
   mem_fifo_st,

INVARIANT
   in_port : port_tlm_bget_if &
   out_port : port_tml_bput_if &
   mem_fifo_st : 0..2 &
   memory_fs : ADDRESS --> DATA


mem_fifo_0 =
   SELECT
      mem_fifo_st =0
   THEN
      in_port ::
         {port_pending} *
         {tlm_bget} *
         basic_request ||
      mem_fifo_st := 1
   END;


mem_fifo_1read =
   ANY ad WHERE
      mem_fifo_st = 1 &

      ad : ADDRESS &
      in_port :
         {port_free} *
         tlm_bget_if_types *
         ({br_READ}*{ad}*DATA)
   THEN
      out_port :=
         port_pending |->
         tlm_bput |->
         (SUCCESS |-> br_READ |-> memory_fs(ad)) ||
      mem_fifo_st := 2
   END;
```

```
mem_fifo_1write =
   ANY ad, dt WHERE
      mem_fifo_st = 1 &

      ad : ADDRESS &
      dt : DATA &
      in_port :
         {port_free} *
         tlm_bget_if_types *
         ({br_WRITE}*{ad}*{dt})
   THEN
      memory_fs(ad) := dt ||
      out_port :=
         port_pending |->
         tlm_bput |->
         (SUCCESS |-> br_WRITE |-> dt) ||
      mem_fifo_st := 2
   END;


mem_fifo_2 =
   SELECT
      mem_fifo_st = 2 &
      out_port :
        {port_free} *
        tlm_bput_if_types *
        tlm_bput_if_type
   THEN
      mem_fifo_st := 0
   END
```

16.4.3 Invariants du raffinement

Un certain nombre d'invariants sont nécessaires pour pouvoir prouver la correction du raffinement. En effet, il faut montrer que le raffinement du système ne modifie pas la façon globale dont les transactions s'enchaînent dans le modèle précédent. Pour ce raffinement, il a été nécessaire d'écrire 43 invariants. Nous ne les écrivons pas tous ici. Nous en donnons quelques uns pour montrer le principe. La nécessité de chacun de ces invariants vient pendant l'effort de preuve pour prouver la correction du raffinement. Il n'est donc pas nécessaire de penser à écrire chacun d'entre eux a priori.

L'invariant suivant stipule par exemple que lorsque la FIFO « request_fifo » n'est pas vide alors la transaction entre l'adaptateur et le channel est nécessairement bloquée (en attente de réponse). En effet, cette FIFO ne sera vide que lorsque l'esclave aura récupéré son contenu, avant d'envoyer sa réponse.

```
(request_fifo /= {} => transport_port :
                       {port_pending}*
                       transport_if_types *
                       transport_if_type ) &
```

L'invariant qui suit naturellement est de dire que la transaction est également bloquée tant que la FIFO « response_fifo » n'est pas vide, puisque que cela signifie que la réponse n'a pas encore été renvoyée vers l'adaptateur.

```
(response_fifo /= {} => transport_port :
                        {port_pending}*
                        transport_if_types *
                        transport_if_type )
```

La plupart de ces invariants sont de ce type. Il s'agit en fait d'un invariant de collage permettant de spécifier de quelle manière on peut déduire l'état dans lequel se trouvent les ports (bloqués ou libres), selon l'état dans lequel se trouvent les modules (états exprimés par leurs variables internes).

16.5 Conclusion

Nous avons montré dans ce chapitre comment il est possible de modéliser des systèmes au niveau TLM dans le langage B. On peut constater sur les modèles qui ont été exposés que cela peut se faire sans obtenir des modèles trop complexes (il n'est pas nécessaire de modéliser l'implantation des primitives elles-mêmes par exemple). Par contre, nous avons constaté que lors du raffinement d'un modèle TLM, il est nécessaire d'écrire un nombre d'invariants relativement élevé (bien que simples). Puisque nous avons montré que la modélisation des concepts TLM dans le langage B peut se faire de manière systématique, nous proposons de définir un langage basé sur B mais dans lequel il n'est pas utile de modéliser explicitement les concepts TLM. Ce langage, nommé BTLM, a une position intermédiaire entre B et SystemC. Un modèle écrit en BTLM peut être traduit simplement dans un modèle B en utilisant les règles de modélisation décrites dans ce document. Un tel langage est également mieux adapté à une traduction vers SystemC puisqu'il utilise directement les mêmes concepts. Le langage BTLM est à la modélisation TLM ce que BHDL est à la modélisation RTL. Ce langage et le traducteur vers SystemC ont été développés au sein de la société KeesDA dans le cadre d'un projet ANVAR et a fait l'objet d'une coopération avec ST Microelectronics. Nous ne le décrivons pas ici pour des raisons de confidentialité.

Conclusion

Nos travaux ont commencé dans le cadre du projet européen PUSSEE par une étude de cas, choisie par Volvo et présentée dans le chapitre 7. Il s'agit du protocole décrit dans le standard de l'industrie automobile SAE J1708. Cette étude de cas nous a permis de présenter comment spécifier un modèle de circuit synchrone en B événementiel en partant d'un modèle abstrait jusqu'au niveau d'implantation où le cycle d'horloge lui-même est pris en compte.

Cette étude de cas concerne un bus série reliant plusieurs composants. Les composants peuvent utiliser le bus pour s'envoyer des messages. Un protocole distribué permet de résoudre les conflits d'accès au bus. L'étude de cas a commencé par une étude de la spécification en construisant un glossaire de l'étude de cas des différents concepts et des principes du protocole. Lors de cette étude préliminaire nous avons constaté que certains points de la spécification du standard pouvaient s'interpréter de manières différentes d'une personne à une autre. La construction du glossaire de l'étude de cas a donc également consisté à fixer notre interprétation de ces points. L'étude de la spécification avait aussi pour but de repérer les propriétés du système ; celles-ci ne sont pas explicites dans la spécification originale. La propriété principale que nous avons dégagé est qu'au plus un composant utilise le bus à un moment donné. C'est pourquoi notre premier modèle abstrait a consisté à représenter le bus comme un ensemble contenant les composants qui utilisent le bus : la propriété principale s'écrivant simplement en disant que cet ensemble contient au plus un élément. Les modèles suivants contiennent nécessairement cette propriété puisqu'ils sont obtenus par raffinement à partir de ce modèle initial. Les raffinements ont consisté à introduire les différents éléments du protocole un par un. La phase la plus importante sur laquelle nous nous sommes concentrés est la résolution des conflits d'accès au bus, c'est à dire du choix du composant qui obtient le bus lorsque plusieurs composants tentent d'y accéder. Ceci a également été modélisé d'abord de façon abstraite en définissant un ordre entre les composants puis raffiné par une succession d'éliminations de composants (phase de compétition) jusqu'à ce qu'il n'en reste plus qu'un. Tel que nous avons modélisé le standard SAE J1708, nous avons la certitude qu'il reste toujours un composant en fin de compétition. Ceci est une amélioration du standard original qui, selon les interprétations, pouvait aboutir au fait qu'aucun composant ne puisse obtenir le bus (et dans le pire des cas à un blocage du système si jamais aucun composant n'arrive à obtenir le bus).

Nous avons apporté une deuxième amélioration du standard en montrant comment il est possible de l'implanter de sorte que tous les composants puissent tenter d'accéder au bus au même moment : après l'écriture d'un message sur le bus, une période de latence est nécessaire. Dans le standard, il est possible que certains composants soient avantagés par rapport à d'autres car ils peuvent se contenter d'une période de latence plus courte que d'autres. Nous avons montré comment cette période de latence peut être implantée de sorte qu'elle se finisse au même moment pour tous les composants.

Le développement a été fait de façon incrémentale par des raffinements formels successifs. Les modèles les plus abstraits du développement constituent une spécification formelle du système. C'est une partie non négligeable du développement car il s'agit du passage de la spécification en langue naturelle à la spécification formelle. C'est pendant cette étape que le standard est interprété. Le modèle cyclique est obtenu en utilisant des variables supplémentaires pour ordonnancer les composants. Nous avons vu que cet ordonnancement par des variables se traduit par un ordonnancement du chemin des données du circuit modélisé. L'ordonnancement autorise les communications entre les composants.

Le modèle que nous avons obtenu est générique au sens où il peut contenir un nombre arbitraire de composants. Nous avons montré comment modéliser le système de façon à ce qu'il puisse comporter un nombre arbitraire de composants identiques sans faire exploser la taille du modèle.

Les événements du modèle B du contrôleur de bus ont ensuite été regroupés de façon à obtenir le "code B" du contrôleur. Grâce aux traducteurs que nous avons développés au sein de la société KeesDA nous avons pu traduire le modèle du contrôleur de bus en VHDL à partir duquel nous avons pu simuler le circuit. Cette description VHDL a été envoyée aux ingénieurs de Volvo qui étaient nos partenaires dans le projet PUSSEE. Le modèle VHDL a également été envoyé à la société Evatronix en Pologne pour réaliser la synthèse du circuit. Les deux retours principaux ont été la découverte d'un "glitch" à la sortie du contrôleur et le fait que le circuit avait

trop d'entrées. Le premier problème a été résolu en modifiant la manière dont les événements du modèle B ont été regroupés (de façon à ce que la sortie du contrôleur soit reliée à un registre). Il s'agit typiquement d'un problème de la spécification qui ne précisait pas que la sortie devait être constante tout au long d'un cycle (et donc reliée à une registre). Le deuxième problème a été résolu en poursuivant le raffinement : un petit protocole a été introduit de façon à obtenir le message à envoyer bit par bit au lieu de le prendre en entier d'un seul coup. On peut donc continuer le raffinement à partir d'un modèle implantable pour satisfaire des contraintes d'implantation exprimées par des experts.

Cette étude de cas nous a permis de dégager des règles de modélisation pour les circuits électroniques en B événementiel, en particulier des contraintes que doit respecter un modèle B pour être implantable par circuit électronique. Nous avons décrit dans ce document les différents principes de modélisation. La modélisation abstraite de circuits ne diffère pas de la modélisation abstraite en général en B mais nous avons vu qu'il était en général nécessaire de faire une analyse de la spécification en langue naturelle pour l'abstraire car, dans le domaine des circuits électroniques, les spécifications sont généralement données à un niveau très bas de description, généralement au niveau bit. Nous avons expliqué comment il est possible de modéliser l'environnement et l'intérêt d'avoir au niveau abstrait des événements modélisant le comportement des composants qui n'ont pas de calculs à effectuer. Nous avons montré comment modéliser les composants pour qu'ils soient indépendants et comment les faire communiquer entre eux. Nous avons pour cela introduit de l'ordonnancement entre les composants. Le cycle d'horloge est également modélisé par cet ordonnancement entre les événements du modèle. Nous avons également montré comment cet ordonnancement peut se représenter graphiquement pour faciliter la compréhension du système.

Cette étude de cas nous a également amené à définir un niveau d'implantation du langage B correspondant aux circuits électroniques, ce qui a donné naissance au langage BHDL. Un modèle BHDL est composé d'un seul événement qui modélise la façon dont le circuit électronique évolue pendant un cycle d'horloge. La clause INITIALISATION contient l'initialisation des registres et nous avons ajouté deux clauses pour spécifier les entrées et les sorties du circuit. Nous avons montré comment on peut classifier les variables du modèle en différentes catégories en définissant des ensembles supports. Ces ensembles supports son définis à partir de la notion de chemin syntaxique définie sur les substitutions généralisées. Un chemin syntaxique correspond à la notion de chemin d'exécution des langages de programmation. L'ensemble support en lecture d'une substitution est l'ensemble des variables lues par la substitution alors que l'ensemble support en écriture est l'ensemble des variables écrites par la substitution. Ces ensembles supports permettent en particulier de différencier automatiquement les variables modélisant des registres et les variables modélisant des fils du circuit électronique. Les ensembles supports permettent également de définir des conditions de bonne formation du modèle BHDL.

Nous avons défini trois sémantiques pour le langage BHDL. Une sémantique prédicative à partir de laquelle on peut définir la traduction vers d'autres formalismes de modélisation de circuits, et une sémantique relationnelle qui est compositionnelle sur la majeure partie du langage. Cette sémantique relationnelle est en fait composée de deux sémantiques : une sémantique amnésique qui considère les registre comme des entrées/sorties ne mémorisant pas les données et une sémantique complète dans laquelle les registres jouent leur rôle de mémoire.

Pour le niveau de modélisation RTL (niveau transferts de registres), nous avons implanté deux traducteurs au sein de la société KeesDA : de BHDL vers VHDL et de BHDL vers SystemC. Ces deux traducteurs ont été décrits en Logic Solver, langage de l'outil AtelierB, proche du langage Prolog. Ces deux traducteurs fonctionnent en deux étapes. Une première étape consiste en l'analyse du modèle BHDL (calcul des ensembles supports, vérification de la bonne formation du modèle) et en la traduction vers un langage intermédiaire. Cette première étape est commune aux deux traducteurs. La deuxième étape consiste en la traduction à partir du langage intermédiaire vers VHDL et SystemC. La majorité du travail est fait dans la première étape. La deuxième étape consistant principalement dans un changement de syntaxe. Le langage intermédiaire ressemble à celui décrit dans ce document et pour lequel nous avons prouvé que la traduction est correcte. Par rapport aux règles de traduction décrites dans ce document, les traducteurs développés au sein de la société KeesDA possèdent quelques règles supplémentaires d'optimisation, elles ne sont pas présentées dans ce document pour des raisons de confidentialité.

Une deuxième étude de cas nous a permis de confirmer les règles de modélisation établies par la première étude de cas. Cette deuxième étude de cas a consisté à modéliser un système spécifié par une équation récurrente (calcul d'une somme de convolution). Dans cette étude de cas, nous avons montré comment l'enchaînement des événements (l'ordonnancement) peut se modéliser par un réseau de Petri. Cette étude de cas a été faite en collaboration avec Stefan Hallerstede au sein de la société KeesDA.

Un des problèmes principaux de l'approche formelle pour le développement de circuits est la réutilisation de composants existants. Nous avons abordé ce problème de deux façons. D'une part nous proposons dans ce document d'ajouter deux substitutions au langage permettant la réutilisation de modèles BHDL et nous avons collaborer avec Diana Toma du laboratoire TIMA pour permettre la preuve d'équivalence entre un modèle BHDL et un modèle VHDL en utilisant le prouveur ACL2. Ces deux approches se complètent. Nous avons vu qu'un bloc

de code pouvait être remplacé par une substitution d'importation lorsque leurs sémantiques amnésiques sont les mêmes. La preuve d'équivalence entre un modèle BHDL et un modèle VHDL en utilisant ACL2 permet d'assurer l'égalité des sémantiques amnésiques des deux modèles. Pour faire la preuve en ACL2, les deux modèles BHDL et VHDL sont traduits dans le langage d'ACL2 (common Lisp). La traduction de VHDL vers ACL2 a été faite par le laboratoire TIMA, elle se fait par une simulation symbolique du circuit. Nous nous sommes occupés de la traduction de BHDL vers ACL2. Pour cela nous avons défini la notion d'aplatissement d'un modèle BHDL. Un modèle BHDL aplati est un modèle formé de plusieurs substitutions composées en parallèle, chacune des substitutions ne modifiant qu'une seule variable du modèle. Dans un tel modèle, la valeur de chaque variable est définie en fonction des entrées et des registres, sans faire appel à des variables intermédiaires. Les règles d'aplatissement ont été implantées en Prolog. Une fois le modèle BHDL aplati, la traduction vers ACL2 est simplement un changement de syntaxe.

Perspectives

Une question importante qui n'a pas été abordée durant cette thèse est celle du "mapping", c'est-à-dire le déploiement de modules fonctionnels sur une architecture donnée. L'hypothèse que nous avons faite est que la décomposition du système faite pendant le développement B correspond à l'architecture cible. Ceci est valable au sens où nous développons des circuits originaux entièrement prouvés. Les systèmes électroniques devenant de plus en plus complexes, les systèmes électroniques sur une puce utilisent maintenant des architectures existantes (dotées de processeurs, de modules de communication avec l'extérieur, d'unités spécialisées ...). Un travail important du concepteur consiste alors à faire correspondre à chaque unité fonctionnelle de son système une unité architecturale (la correspondance n'est pas nécessairement 1-1). Dans notre hypothèse de travail, ce problème n'existe pas puisque nous fabriquons une architecture originale en même temps que le développement du système.

Le travail effectué pendant cette thèse a consisté à modéliser des systèmes en partant d'un haut niveau d'abstraction (spécification) et en le raffinant jusqu'au niveau RTL (niveau transfert de registre que l'on considère comme le niveau d'implantation). Le but était de montrer comment la méthode B peut être utilisée pour le développement jusqu'à l'implantation de circuits électroniques sûrs. Pendant la chaîne de raffinement le système passe donc d'un niveau très abstrait à un niveau très concret en passant par des étapes intermédiaires. Le monde du développement de circuits s'intéresse de plus en plus à des niveaux plus abstraits que le niveau RTL dans le but d'obtenir des simulations plus rapides mais également pour permettre un cycle de développement plus rapide lorsque des logiciels et des circuits doivent être développés conjointement. Donner un modèle de circuit simulable de haut niveau aux personnes développant le logiciel permet le développement de celui-ci avant que le circuit ne soit complètement développé. Un niveau SystemC-TLM ("Transaction Level Modelling", niveau où les communications entre les composants sont exprimées par des transactions) a été standardisé cette année. Notre but est de proposer des niveaux de raffinement auxquels les modèles B peuvent être traduits vers du SystemC-TLM. Nous avons modélisé certains exemples proposés par le nouveau Standard SystemC-TLM en utilisant la méthode B afin d'identifier les concepts principaux de modélisation. En collaboration avec Diana Moisuc, nous avons défini le langage BTLM, langage intermédiaire entre B et SystemC, ainsi que les règles de traduction vers SystemC dans le but de proposer un traducteur automatique. Le langage BTLM est un langage permettant d'utiliser de façon simple les concepts du niveau TLM sans avoir à les modéliser à la main en B, ce langage peut être vu comme un "template" de modèle B, modèle B sur lequel nous pouvons utiliser les outils de preuve traditionnels.

Il reste encore du travail à faire pour rendre les modèles B de niveau TLM utiles. Il reste à définir la traduction du langage ad-hoc BTLM vers B. Par ailleurs, les modèles B qui pourraient actuellement être obtenus par traduction à partir de BTLM sont trop complexes pour envisager d'être prouvés facilement. Il faut également définir les guides de modélisation permettant de passer facilement d'un modèle TLM à un modèle RTL, autrement dit il faut encore définir comment on peut raffiner un modèle TLM sans avoir un nombre de preuves trop important à effectuer.

Une autre voie de recherche serait de raffiner l'expression des propriétés sur le système jusqu'au niveau de l'implantation en utilisant les invariants de collage définis pendant le processus de raffinement. En effet il est souvent difficile d'exprimer des propriétés systèmes au niveau de l'implantation car on a accès qu'à des variables de bas niveau. En B les propriétés systèmes sont exprimées et prouvées dans les modèles abstraits en utilisant des variables abstraites. Pendant le processus de raffinement ces variables sont raffinées par des variables concrètes, les invariant de collage exprimant la relation entre les variables abstraites et les variables concrètes. On peut donc envisager de réécrire les propriétés systèmes intéressantes à chaque pas de raffinement en effectuant le remplacement des variables abstraites par les variables concrètes. Une fois le processus de raffinement arrivé jusqu'au niveau de l'implantation, nous disposerions ainsi de l'expression de propriétés systèmes exprimées au niveau de l'implantation. Disposer de ces propriétés exprimées au niveau implantation peut permettre d'en disposer pour effectuer des vérifications sur des modèles existants non développés formellement. Elles pourraient aussi être

utilisée pour développer des "IP de vérification", c'est à dire des composants intégrés au système permettant de vérifier son bon fonctionnement pendant son exécution.

Bibliographie

[1] J-R Abrial. *The B-Book – Assigning programs to meanings.* Cambridge University Press, 1996.

[2] J-R Abrial. Event Driven Sequential Program Construction. http ://www.atelierb.societe.com/, October 2000.

[3] J-R Abrial. Guidelines to Formal System Studies. http ://www.atelierb.societe.com/, November 2000.

[4] J-R Abrial. Event Driven Electronic Circuit Construction. Unpublished, August 2001.

[5] J.-R. Abrial and L. Mussat. Introducing Dynamic Constraints in B. In D. Bert, editor, *B'98 : Recent Advances in the Development and Use of the B-Method*, volume 1393 of *LNCS*, pages 83–128, 1998.

[6] Jean-Raymond Abrial. Event based sequential program development : Application to constructing a pointer program. In *FME 2003 : Formal Methods*, volume 2805 of *Lecture Notes in Computer Science*, pages 51 – 74, 2003.

[7] Jean-Raymond Abrial and Dominique Cansell. Click'n'Prove : Interactive Proofs Within Set Theory. In D. Basin and B. Wolff, editors, *16th Intl. Conf. Theorem Proving in Higher Order Logics (TPHOLs'2003)*, volume 2758 of *LNCS*, pages 1–24. Springer Verlag, September 2003.

[8] Jean-Raymond Abrial, Dominique Cansell, and Dominique Méry. A mechanically proved and incremental development of ieee 1394 tree identify protocol. *Formal Aspects of Computing*, 14(3) :215–227, Apr 2003.

[9] J.R. Abrial. Extending B without changing it (for developing distributed systems). In *Proc. of the 1st B Conference, Nantes, France*, 1996.

[10] alaide.com. Ordinateur asynchrone. http ://www.alaide.com/dico.php ?q=Ordinateur+asynchrone&ix=3290.

[11] Aldec. ActiveHDL. http ://www.aldec.com/products/active-hdl/.

[12] Ammar Aljer, Philippe Devienne, Sophie Tison, Jean-Louis Boulanger, and Georges Mariano. B-HDL : Circuit Design in B. In *ACSD 2003, International conference on Application of Concurrency to System Design*, pages 241–242, 2003.

[13] Laurent Arditi. Vérification formelle des microprocesseurs : une première expérimentation avec coq. In *Journées du GDR de programmation. Grenoble (France)*, 1995.

[14] B. Brock, M. Kaufmann, and JS. Moore. ACL2 theorems about commercial microprocessors. In M. Srivas and A. Camilleri, editors, *First international conference on formal methods in computer-aided design*, volume 1166, pages 275–293, Palo Alto, CA, USA, 1996. Springer Verlag.

[15] B-Core (UK), Harwell, United Kingdom. *B-Toolkit.* Software, www.b-core.com/btoolkit.html.

[16] Franck Bardoult, Patrice Quinton, Sanjay Rajopadhye, and Tanguy Risset. Synthesis of Data-Flow Interfaces for Regular Parallel Programs. Technical-report, INRIA, 1999.

[17] Ilan Beer, Shoham Ben-David, Cindy Eisner, and Avner Landver. RuleBase : an industry-oriented formal verification tool. In *DAC 96*, 1996. http ://www.haifa.il.ibm.com/projects/verification/RB_Homepage/publications.html.

[18] Didier Bert. Preuve de propriétés d'équité en B : étude du protocole du bus SCSI-3. In *Actes de l'Atelier AFADL-2001*, pages 221–241, 2001.

[19] D. Borrione and D. Toma. SHA formalization. In *ACL2 Workshop*, USA, 2003.

[20] Jean-Louis Boulanger, Ammar Aljer, and Georges Mariano. Formalization of digital circuits using the B method. In *AFIS, EUSEC 3rd European Systems Engineering Conference*, pages 281–290, 2002.

[21] M. C. Browne, E. M. Clarke, and O. Grümberg. Characterizing finite kripke structures in propositional temporal logic. *Theor. Comput. Sci.*, 59(1-2) :115–131, 1988.

[22] Cadence. Encounter conformal equivalence checking solutions. http ://www.cadence.com/products/functional_ver/conformal/index.aspx.

[23] D. Cansell, G. Gopalakrishnan, M. Jones, D. Méry, and A. Weinzoepflen. Incremental Proof of the Producer/Consumer Property for the PCI Protocol. In D. Bert, editor, *ZB 2002*, volume 2272 of *lncs*, 2002.

[24] Dominique Cansell and Jean-Raymond Abrial. Click'n'Prove. http ://www.loria.fr/~cansell/cnp.html.

[25] Dominique Cansell and Dominique Méry. Logical foundations of the B method. *Computers and Informatics*, 22, 2003.

[26] Dominique Cansell and Dominique Méry. Integration of the proof process in the system development through refinement steps. In *procedings of FDL'02*, volume 2, September 2002.

[27] Clearsy. Clearsy web page. www.clearsy.com.

[28] ClearSy – Systems Engineering, Aix-en-Provence, France. *Atelier B.* Software, atelierb.societe.com.

[29] Cleasy. B4free. http ://b4free.com.

[30] Verilog DOT COM. Verilog ressources. www.verilog.com.

[31] Solange Coupet-Grimal and Line Jakubiec. Vérification formelle de circuits avec Coq. In *Journées du GDR de programmation.*, 1994.

[32] Solange Coupet-Grimal and Line Jakubiec. Coq and hardware verification : A case study. In J. von Wright, J. Grundy, and J. Harrison, editors, *TPHOLs'96, Turku, Finland, 26–30 Aug 1996*, volume 1125 of *LNCS*, pages 125–139. Springer-Verlag, Berlin, 1996.

[33] D. Cyrluk, S. Rajan, N. Shankar, and M. Srivas. Effective theorem proving for hardware verification. In *Proceedings of Theorem Provers in Circuit Design*, 1994.

[34] David Cyrluk. Microprocessor verification in PVS : A methodology and simple example. Technical report, SRI International, Computer Science Laboratory, 1994.

[35] Florent Dupont de Dinechin, Tanguy Risset, M. Manjunathaiah, and Mike Spivey. Design of highly parallel architectures with Alpha and Handel. In *FDL'02*, Marseille, France, 2002.

[36] Michel Diaz, editor. *Les réseaux de Petri, Modèles fondamentaux.* Hermès Science Publications, 2001.

[37] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[38] Edsger W. Dijkstra and Carel S. Scholten. *Predicate calculus and program semantics.* Springer-Verlag New York, Inc., 1990.

[39] Doron Drusinsky and David Harel. Using Statecharts for Hardware Description and Synthesis. *IEEE Transactions on Computer-Aided Design*, 8(7) :798–807, July 1989.

[40] Steve Dunne. A Theory of Generalised Substitutions. In *ZB2002 : Formal Specification and Development in Z and B*, LNCS 2272, pages 270–290, Grenoble, January 2002.

[41] Equipe MOSEL. www.loria.fr/equipes/mosel.

[42] European Organisation for Civil Aviation Equipment. , http ://www.eurocae.org/.

[43] A.C.J Fox. Formal specification and verification of ARM6. In D. Basin and B. Wolff, editors, *TPHOLs '03*, volume 2758 of *LNCS*, pages 25–40. Springer, 2003.

[44] Max Fuchs and Michael Mendler. A Functional Semantics for Delta-Delay VHDL Based on FOCUS. In C. D. Kloos and P. T. Breuer, editors, *Formal Semantics for VHDL*, pages 9–42. Kluwer Academic Publishers, 1995.

[45] Max Fuchs and Michael Mendler. *Formal Semantics for VHDL*, chapter A Functional Semantics for Delta-Delay VHDL based on Focus. Kluwer Academic Publishers, 1995.

[46] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. *Proceedings of the IEEE*, 79(9) :1305–1320, September 1991.

[47] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying real-time systems by means of the synchronous data-flow programming language lustre. *IEEE Transactions on Software Engineering, Special Issue on the Specification and Analysis of Real-Time Systems*, September 1992.

[48] S. Hallerstede. Specification and Refinement of Hardware Components in B. In *FDL 2002*, 2002.

[49] Stefan Hallerstede. BHDL user guide, 2004. KeesDA.

[50] Stefan Hallerstede and Yann Zimmermann. Circuit Design by refinement in EventB. In *Proc. of FDL'04*, 2004.

[51] Scott Hazelhurst and Carl-Johan H. Seger. Symbolic Trajectory Evaluation. In T. Kropf, editor, *Formal Hardware Verification : Methods and Systems in Comparison*, volume 1287 of *LNCS*, pages 3–78. Springer-Verlag, 1997.

[52] Yoav Hollanderl, Matthew Morley, and Amos Noy. The e Language : A Fresh Separation of Concerns, 2000. http ://www.cadence.com/whitepapers/.

[53] IBM. PSL/Sugar Homepage. http ://www.haifa.ibm.com/projects/verification/sugar/index.html.

[54] IEEE, editor. *Standard VHDL - Language Reference Manual.* IEEE Computer Society Press, USA, 1988.

[55] IEEE. Ieee standard hardware description language based on the verilog hardware description language, 1995. IEEE Std. 1364-1995.

[56] IEEE. Ieee standard systemc language reference manual, 1995. IEEE Std. 1666-2005s.

[57] INRIA. The Coq proof assistant. coq.inria.fr/.

[58] International Electrotechnical Commission. , http ://www.iec.ch/61508/.

[59] M. Kaufmann, P. Manolios, and J S. Moore. *Computer-Aided reasoning : ACL2 An approach*, volume 1. Kluwer Academic Press, 2000.

[60] M. Kaufmann, P. Manolios, and J S. Moore. *Computer-Aided reasoning : ACL2 Case Studies*, volume 2. Kluwer Academic Press, 2000.

[61] Philip J. Kuekes, Duncan R. Stewart, and R. Stanley Williams. The crossbar latch : Logic value storage, restoration, and inversion in crossbar circuits. *Journal of Applied Physics*, 97(034301), 2005.

[62] SRI Computer Science Laboratory. http ://www.csl.sri.com/programs/formalmethods/.

[63] Leslie Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3) :872–923, 1994.

[64] Carlos Deldago Loos and Peter T. Breuer, editors. *Formal Semantics for VHDL.* Kluwer Academic Publishers, 1995.

[65] LORIA. www.loria.fr.

[66] Peter Martin. A hardware implementation of a genetic programming system using FPGAs and Handel-C. *Genetic Programming and Evolvable Machines*, 2(4) :317–343, December 2001.

[67] K. L. McMillan. The SMV system. http ://www.cs.cmu.edu/~modelcheck/smv.html.

[68] George H. Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34(5) :1045–1079, 1955.

[69] J. Mermet, editor. *UML-B - Specification for Proven Embedded Systems Design.* Kluwer Academic Publishers, 2004.

[70] Traian Muntean and Olivier Rolland. Refining Distributed Systems Using the B Method. In *RCS'02*, 2002. www.esil.univ-mrs.fr/~spc/rcs02/papers/Muntean_Rolland.pdf.

[71] Alan V. Oppenheim and Ronald W. Schafer. *Discrete-Time Signal Processing.* Prentice Hall Signal Processing Series. Prentice Hall, Englewood Cliffs, NJ, USA, 1989.

[72] S. Owre, J. M. Rushby, N. Shankar, and M. K. Srivas. A tutorial on using PVS for hardware verification. In Ramayya Kumar and Thomas Kropf, editors, *Theorem Provers in Circuit Design (TPCD '94)*, volume 901 of *Lecture Notes in Computer Science*, pages 258–279, Bad Herrenalb, Germany, sep 1994. Springer-Verlag.

[73] Telecom Paris. Historique rapide. http ://www.comelec.enst.fr/hdl/vhdl_intro.html.

[74] Telecom Paris. Historique rapide. http ://www.comelec.enst.fr/hdl/verilog_intro.html.

[75] Telecom Paris. A propos de SystemC. http ://www.comelec.enst.fr/hdl/sc_intro.html.

[76] C. Paulin-Mohring. Circuits as streams in Coq : Verification of a sequential multiplier. In S. Berardi and M. Coppo, editors, *Types for Proofs and Programs, TYPES'95*, volume 1158 of *Lecture Notes in Computer Science*, 1996.

[77] J. Plosila and K. Sere. Action systems in pipelined processor design. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 156–166. IEEE Computer Society Press, 1997.

[78] Juha Plosila and Tiberiu Seceleanu. Synchronous action systems. Technical Report TUCS-TR-192, Turku Centre for Computer Science, Finland, 1998.

[79] Cyril Proch. *Assistance au développement incrémental et prouvé de systèmes enfouis.* PhD thesis, Université Henri Poincaré, 2006.

[80] PUSSEE. project IST-2000-30103, 2002–2004. www.keesda.com/pussee.

[81] Radio Technical Commission for Aeronautics. , http ://www.rtca.org/.

[82] D. M. Russinoff. A Case Study in Formal Verification of Register-Transfer Logic with ACL2 : The Floating Point Adder of the AMD Athlon Processor. In *FMCAD 2000*, 2000.

[83] SAE International. SAE J1708 revised OCT93, serial data communication between microcomputer systems in heavy-duty vehicule applications, www.sae.org, 1993.

[84] Ib Sørensen. Using b to specify, verify and design hardware circuits. In *ZUM '98 : Proceedings of the 11th International Conference of Z Users on The Z Formal Specification Notation*, pages 60–66, London, UK, 1998. Springer-Verlag.

[85] M. Srivas, H. Rueß, and D. Cyrluk. Hardware Verification Using PVS. In T. Kropf, editor, *Formal Hardware Verification : Methods and Systems in Comparison*, volume 1287 of *LNCS*, pages 156–205. Springer-Verlag, 1997.

[86] Synopsys. Synopsys products : Formal verification - formality. http ://www.synopsys.com/products/verification/verification.html.

[87] SystemC. SystemC home page. systemc.org.

[88] D. Toma, D. Borrione, and G. Al-Sammane. Combining several paradigms for circuit validation and verification. In *CASSIS*, 2004.

[89] VERIMAG. Gloups. http ://www-verimag.imag.fr/~synchron/index.php ?page=gloups.

[90] Verimag. Lesar. http ://www-verimag.imag.fr/~raymond/tools/lv4-distrib.html.

[91] VERIMAG. Nbac. http ://www.irisa.fr/prive/bjeannet/nbac/nbac.html.

[92] VSI Working Group. IEEE P1076.6/D2.01 – Draft Standard For VHDL Register Transfer Level Synthesis. Unapproved Draft, IEEE, 2001.

[93] Yann Zimmermann, Stefan Hallerstede, and Dominique Cansell. Formal modelling of electronic circuits using event-B, case study : SAE J708 serial communication link. In Jean Mermet, editor, *UML-B - Specification for Proven Embedded Systems Design*. Kluwer Academic Publishers, 2004.

[94] Yann Zimmermann and Diana Toma. Component reuse in B using ACL2. In *ZB2005*, number 3455 in LNCS, pages 280–299.

Glossaire

Circuit asynchrone : circuit électronique contenant des éléments de mémorisation mais dont la synchronisation n'est pas subordonnée à une horloge. On utilise par exemple le principe du "rendez-vous" pour la synchronisation des différentes parties du circuit.

Circuit combinatoire : circuit électronique ne contenant pas d'élément de mémorisation.

Circuit synchrone : circuit contenant des éléments de mémorisation synchronisés par une horloge. Par exemple les éléments de mémorisation changent de valeur uniquement au front montant d'horloge.

Front montant d'horloge : instant où l'horloge cyclique passe de l'état bas ('0') à l'état haut ('1').

Glitch : courte instabilité d'un signal électrique. Par exemple un signal qui devrait être constamment dans l'état '0' passe dans l'état '1' pour revenir aussitôt dans l'état '0'. Ce phénomène peut se produire par exemple à la sortie d'un circuit combinatoire le temps que les signaux se stabilisent (le temps que les nouvelles entrées aient parcouru l'ensemble du circuit jusqu'aux sorties). Pour éviter ce genre de problème on fait en sorte que les sorties d'un circuit soient reliées à des registres (stables pendant toute la durée d'un cycle).

Machine de Mealy : machine cyclique théorique possédant des entrées, des sorties et un état interne. Les sorties et l'état interne d'un cycle donné sont exprimés en fonction des entrées au même cycle et de l'état du cycle précédent.

Machine de Moore : machine de Mealy dans laquelle les sorties sont exprimées en utilisant uniquement l'état interne (sans utiliser les entrées).

Netlist : description d'un circuit par un réseau de portes logiques. C'est à partir de cette description que la fabrication du circuit est faite.

RTL : Niveau transfert de registres pour la description d'un circuit. Le circuit est décrit par des registres et une partie combinatoire. Niveau à partir duquel on peut utiliser des synthétiseurs automatiques pour obtenir une description par *netlist*.

SoC : Acronyme pour *System On Chip, système enfoui*

Cinquième partie

Annexes

Annexe A

Grammaire du langage BHDL

$$
\begin{array}{rcl}
BHDL & ::= & \left[\begin{array}{l}\text{MACHINE } ident \\ \text{IMPORTS } limport \\ \text{SEES } lident \\ \text{INPUTS } lvar \\ \text{OUTPUTS } lvar \\ \text{VARIABLES } lvar \\ \text{INVARIANT } pred \\ \text{INITIALISATION } inits \\ \text{OPERATIONS } C \\ \text{END}\end{array}\right. \\
C & ::= & C \parallel C \\
 & | & C;\ C \\
 & | & \text{BEGIN } C \text{ END} \\
 & | & \text{IF } pred \text{ THEN } C \\
 & & \quad \text{ELSE } C \text{ END} \\
 & | & \text{IF } pred \text{ THEN } C \text{ END} \\
 & | & var := exp \\
 & | & var :\in exp \\
 & | & lport \leftarrow ident\ (\ lport\ ) \\
inits & ::= & inits \parallel inits \\
 & | & var := const \\
 & | & var :\in const
\end{array}
$$

$$
\begin{array}{rcl}
lident & ::= & lident\ ,\ ident \\
 & | & ident \\
limport & ::= & limport,\ import \\
 & | & import \\
import & ::= & ident : ident \\
lport & ::= & lport,\ port \\
 & | & port \\
port & ::= & const : var \\
var & ::= & ident \\
exp & ::= & exp\_bool \\
 & | & exp\_arith \\
 & | & lambda \\
lambda & ::= & \lambda.ident\ (pred \mid exp) \\
 & | & lambda \cup lambda \\
exp\_arith & ::= & var \mid const \mid const\_num \\
 & | & exp\_arith\ op\_arith\ exp\_arith \\
exp\_bool & ::= & var \mid const \mid const\_bool \\
 & | & \text{bool}(pred) \\
pred & ::= & exp\_bool\ op\_bool\ exp\_arith \\
 & | & \neg exp\_bool \\
 & | & exp\_arith\ op\_bool\_arith\ exp\_arith \\
op\_arith & ::= & + \mid - \mid * \mid / \mid \text{mod} \\
op\_bool & ::= & \wedge \mid \vee \mid \Rightarrow \\
op\_bool\_arith & ::= & \geq \mid \leq \mid < \mid > \mid = \\
const\_num & ::= & \text{nombre entier} \\
const\_bool & ::= & \text{true} \mid \text{false} \\
ident & ::= & \text{B identifer} \\
var & ::= & ident \in (inputs \cup ouputs \cup variables) \\
const & ::= & ident \notin (inputs \cup ouputs \cup variables) \\
 & | & const\_num \mid const\_bool
\end{array}
$$

$$
\begin{array}{rcl}
PACKAGE & ::= & \left[\begin{array}{l}\text{MACHINE } ident \\ \text{SEES } lident \\ \text{CONSTANTS } lident \\ \text{PROPERTIES } ldef \\ \text{END}\end{array}\right. \\
ldef & ::= & ldef \wedge def \\
 & | & def \\
def & ::= & ident = const \\
 & | & ident = const..const \\
 & | & ident = const \rightarrow const \\
 & | & ident = \{\ lident\ \}
\end{array}
$$

Annexe B

Calcul compositionnel des ensembles supports

On prouve la compositionnalité en présentant le calcul des ensembles supports *read*, *write* et *writeall*. Ceci se fait par induction sur la structure d'une substitution.

Les preuves utilisent principalement les deux transformations suivantes :

$\forall(x,y)\cdot(\mathcal{P}(x)\vee\mathcal{Q}(y))\Leftrightarrow\forall x\cdot\mathcal{P}(x)\vee\forall y\cdot\mathcal{Q}(y)$

$\exists(x,y)\cdot(\mathcal{P}(x)\vee\mathcal{Q}(y))\Leftrightarrow\exists x\cdot\mathcal{P}(x)\vee\exists y\cdot\mathcal{Q}(y)$

Substitution simple

Soit S une substitution simple, $S\equiv x:=E\vee S\equiv x:\in E\vee S\equiv(O)\leftarrow ic(I)$. Dans chaque cas, nous avons $spaths(S)=\{S\}$. Ainsi nous avons l'équation suivante :

$read(S)=\{v|\exists p\cdot(p\in spaths(S)\wedge v\in read^s(p))\}=read^s(S)$

$write(S)=\{v|\exists p\cdot(p\in spaths(S)\wedge v\in write^s(p))\}=write^s(S)$

$writeall(S)=\{v|\forall p\cdot(p\in spaths(S)\Rightarrow v\in write^s(p))\}=write^s(S)$

Composition séquentielle

- Ensemble support en lecture

$read(S;T)=\{v|\exists p\cdot(p\in spaths(S;T)\wedge v\in read^s(p))\}$

$=\{v|\exists(S_p,T_p)\cdot(S_p\in spaths(S)\wedge T_p\in spaths(T)\wedge v\in read^s(S_p;T_p))\}$

$read^s(S_p;T_p)=read^s(S_p)\cup(read^s(T_p)-write(S_p))$

$v\in read^s(S_p;T_p))\Leftrightarrow v\in read^s(S_p)\vee(v\in read^s(T_p)\wedge\neg v\in write(S_p))$

$\exists(S_p,T_p)\cdot(S_p\in spaths(S)\wedge T_p\in spaths(T)\wedge v\in read^s(S_p;T_p))$

$\Leftrightarrow$

$\exists S_p\cdot(S_p\in spaths(S)\wedge v\in read^s(S_p))$

$\vee\ \exists(S_p,T_p)\cdot(S_p\in spaths(S)\wedge T_p\in spaths(T)\wedge v\in read^s(T_p)\wedge\neg v\in write(S_p))$

$\Leftrightarrow$

$\exists S_p\cdot(S_p\in spaths(S)\wedge v\in read^s(S_p))$

$\vee\ \exists S_p\cdot(S_p\in spaths(S)\wedge v\in read^s(T_p))$

$\wedge\ \exists S_p\cdot(S_p\in spaths(S))\wedge\neg v\in write(S_p))$

$\Leftrightarrow$

$\exists S_p\cdot(S_p\in spaths(S)\wedge v\in read^s(S_p))$

$\vee\ \exists S_p\cdot(S_p\in spaths(S)\wedge v\in read^s(T_p))$

$\wedge\ \neg\forall S_p\cdot(S_p\in spaths(S))\Rightarrow v\in write(S_p))$

$\Leftrightarrow$

$v\in read(S)\vee(v\in read(T)\wedge\neg v\in writeall(S))$

Donc, $read(S;T)=read(S)\cup(read(T)-writeall(S))$

- Ensemble support en écriture

$write(S;T) = \{v|\exists p \cdot (p \in spaths(S;T) \wedge v \in write^s(p))\}$
$= \{v|\exists (S_p,T_p) \cdot (S_p \in spaths(S) \wedge T_p \in spaths(T) \wedge v \in write^s(S_p;T_p))\}$
$write^s(S_p;T_p) = write^s(S_p) \cup write^s(T_p)$
$v \in write^s(S_p;T_p) \Leftrightarrow v \in write^s(S_p) \vee v \in write^s(T_p)$
$\exists (S_p,T_p) \cdot (S_p \in spaths(S) \wedge T_p \in spaths(T) \wedge v \in write^s(S_p;T_p))$
$\Leftrightarrow$
$\exists (S_p) \cdot (S_p \in spaths(S) \wedge v \in write^s(S_p)) \vee \exists (T_p) \cdot (T_p \in spaths(T) \wedge v \in write^s(T_p))$
$\Leftrightarrow$
$v \in write(S) \vee v \in write(T)$
Donc, $write(S;T) = write(S) \cup write(T)$

- Ensemble support total en écriture

$writeall(S;T) = \{v|\forall p \cdot (p \in spaths(S;T) \Rightarrow v \in write^s(p))\}$
$= \{v|\forall (S_p,T_p) \cdot (S_p \in spaths(S) \wedge T_p \in spaths(T) \Rightarrow v \in write^s(S_p;T_p))\}$
$v \in write^s(S_p;T_p) \Leftrightarrow v \in write^s(S_p) \vee v \in write^s(T_p)$
$\forall (S_p,T_p) \cdot (S_p \in spaths(S) \wedge T_p \in spaths(T) \Rightarrow v \in write^s(S_p;T_p))$
$\Leftrightarrow$
$\forall (S_p,T_p) \cdot (S_p \in spaths(S) \wedge T_p \in spaths(T) \Rightarrow (v \in write^s(S_p) \vee v \in write^s(T_p)))$
$\Leftrightarrow$
$\forall S_p \cdot (S_p \in spaths(S) \Rightarrow v \in write^s(S_p))$
$\vee \forall T_p \cdot (T_p \in spaths(T) \Rightarrow v \in write^s(T_p))$
$\Leftrightarrow$
$v \in writeall(S) \vee v \in writeall(T)$
Donc, $writeall(S;T) = writeall(S) \cup writeall(T)$

Composition parallèle

- Ensemble support en lecture

$read(S\|T) = \{v|\exists p \cdot (p \in spaths(S\|T) \wedge v \in read^s(p))\}$
$= \{v|\exists (S_p,T_p) \cdot (S_p \in spaths(S) \wedge T_p \in spaths(T) \wedge v \in read^s(S_p\|T_p))\}$
$read^s(S_p\|T_p) = read^s(S_p) \cup read^s(T_p)$
$\exists (S_p,T_p) \cdot (S_p \in spaths(S) \wedge T_p \in spaths(T) \wedge v \in read^s(S_p\|T_p))\}$
$\Leftrightarrow \exists S_p \cdot (S_p \in spaths(S) \wedge v \in read^s(S_p)) \vee \exists T_p \cdot (T_p \in spaths(T) \wedge v \in read^s(T_p))$
$\Leftrightarrow v \in read(S) \vee v \in read(T)$
Donc, $read(S\|T) = read(S) \cup read(T)$

- Ensemble support en écriture

$write(S\|T) = \{v|\exists p \cdot (p \in spaths(S\|T) \wedge v \in write^s(p))\}$
$= \{v|\exists (S_p,T_p) \cdot (S_p \in spaths(S) \wedge T_p \in spaths(T) \wedge v \in write^s(S_p\|T_p))\}$
$write^s(S_p\|T_p) = write^s(S_p) \cup write^s(T_p)$
$\exists (S_p,T_p) \cdot (S_p \in spaths(S) \wedge T_p \in spaths(T) \wedge v \in write^s(S_p\|T_p))\}$
$\Leftrightarrow \exists S_p \cdot (S_p \in spaths(S) \wedge v \in write^s(S_p)) \vee \exists T_p \cdot (T_p \in spaths(T) \wedge v \in write^s(T_p))$
$\Leftrightarrow v \in write(S) \vee v \in write(T)$
Donc, $write(S\|T) = write(S) \cup write(T)$

- Ensemble support total en écriture

$writeall(S\|T) = \{v|\forall p \cdot (p \in spaths(S\|T) \Rightarrow v \in write^s(p))\}$
$= \{v|\forall (S_p,T_p) \cdot (S_p \in spaths(S) \wedge T_p \in spaths(T) \Rightarrow v \in write^s(S_p\|T_p))\}$
$v \in write^s(S_p\|T_p) \Leftrightarrow v \in write^s(S_p) \vee v \in write^s(T_p)$
$\forall (S_p,T_p) \cdot (S_p \in spaths(S) \wedge T_p \in spaths(T) \Rightarrow v \in write^s(S_p\|T_p))$
$\Leftrightarrow$
$\forall (S_p,T_p) \cdot (S_p \in spaths(S) \wedge T_p \in spaths(T) \Rightarrow (v \in write^s(S_p) \vee v \in write^s(T_p)))$
$\Leftrightarrow$

$$\forall S_p \cdot (S_p \in spaths(S) \Rightarrow v \in write^s(S_p)) \vee \forall T_p \cdot (T_p \in spaths(T) \Rightarrow v \in write^s(T_p))$$
$$\Leftrightarrow$$
$$v \in writeall(S) \vee v \in writeall(T)$$

Donc, $writeall(S \| T) = writeall(S) \cup writeall(T)$

Composition conditionnelle

- Ensemble support en lecture

$read(\text{IF } \mathcal{C} \text{ THEN } S \text{ ELSE } T \text{ END})$
$= \{v | \exists p \cdot (p \in spaths(\text{IF } \mathcal{C} \text{ THEN } S \text{ ELSE } T \text{ END}) \wedge v \in read^s(p))\}$
$= \{v | \exists (S_p, T_p) \cdot (S_p \in spaths(S) \wedge T_p \in spaths(T) \wedge (v \in read^s(\xi := bool(\mathcal{C}); S_p) \vee read^s(\xi := bool(\mathcal{C}); T_p)))$

$read^s(\xi := bool(\mathcal{C}); S_p) = read^s(\xi := bool(\mathcal{C})) \cup (read^s(S_p) - write^s(\xi := bool(\mathcal{C})) = free(\mathcal{C}) \cup read^s(S_p)$

$read^s(\xi := bool(\mathcal{C}); T_p) = read^s(\xi := bool(\mathcal{C})) \cup (read^s(T_p) - write^s(\xi := bool(\mathcal{C})) = free(\mathcal{C}) \cup read^s(T_p)$

$\exists (S_p, T_p) \cdot (S_p \in spaths(S) \wedge T_p \in spaths(T) \wedge (v \in read^s(\xi := bool(\mathcal{C}); S_p) \vee read^s(\xi := bool(\mathcal{C}); T_p)))\}$
$\Leftrightarrow$
$\exists S_p \cdot (S_p \in spaths(S) \wedge v \in free(\mathcal{C}) \cup read^s(S_p)) \vee \exists T_p \cdot (T_p \in spaths(T) \wedge v \in free(\mathcal{C}) \cup read^s(T_p))$
$\Leftrightarrow$
$v \in free(\mathcal{C}) \vee \exists S_p \cdot (S_p \in spaths(S) \wedge v \in read^s(S_p)) \vee \exists T_p \cdot (T_p \in spaths(T) \wedge v \in read^s(T_p))$
$\Leftrightarrow$
$v \in free(\mathcal{C}) \vee v \in read(S) \vee v \in read(T)$

Donc, $read(\text{IF } \mathcal{C} \text{ THEN } S \text{ ELSE } T \text{ END}) = free(\mathcal{C}) \cup read(S) \cup read(T)$

- Ensemble support en écriture

$write(\text{IF } \mathcal{C} \text{ THEN } S \text{ ELSE } T \text{ END})$
$= \{v | \exists p \cdot (p \in spaths(\text{IF } \mathcal{C} \text{ THEN } S \text{ ELSE } T \text{ END}) \wedge v \in write^s(p))\}$
$= \{v | \exists (S_p, T_p) \cdot (S_p \in spaths(S) \wedge T_p \in spaths(T) \wedge (v \in write^s(\xi := bool(\mathcal{C}); S_p) \vee write^s(\xi := bool(\mathcal{C}); T_p)))$

$write^s(\xi := bool(\mathcal{C}); S_p) = write^s(\xi := bool(\mathcal{C})) \cup write^s(S_p) = write^s(S_p)$ (ξ is never considered to be written, $write^s(\xi := ...) = \varnothing$)

$write^s(\xi := bool(\mathcal{C}); T_p) = write^s(T_p)$ (idem)

$\exists (S_p, T_p) \cdot (S_p \in spaths(S) \wedge T_p \in spaths(T) \wedge (v \in write^s(\xi := bool(\mathcal{C}); S_p) \vee write^s(\xi := bool(\mathcal{C}); T_p)))$
$\Leftrightarrow$
$\exists (S_p, T_p) \cdot (S_p \in spaths(S) \wedge T_p \in spaths(T) \wedge (v \in write^s(S_p) \vee v \in write^s(T_p)))$
$\Leftrightarrow$
$\exists S_p \cdot (S_p \in spaths(S) \wedge v \in write^s(S_p)) \vee \exists T_p \cdot (T_p \in spaths(T) \wedge v \in write^s(T_p))$
$\Leftrightarrow$
$v \in write(S) \vee v \in write(T)$

Donc, $write(\text{IF } \mathcal{C} \text{ THEN } S \text{ ELSE } T \text{ END}) = write(S) \cup write(T)$

- Ensemble support total en écriture

$writeall(\text{IF } \mathcal{C} \text{ THEN } S \text{ ELSE } T \text{ END})$
$= \{v | \forall p \cdot (p \in spaths(\text{IF } \mathcal{C} \text{ THEN } S \text{ ELSE } T \text{ END}) \Rightarrow v \in write^s(p))\}$
$= \{v | \forall (S_p, T_p) \cdot (S_p \in spaths(S) \wedge T_p \in spaths(T)$

$$
\begin{array}{l}
\qquad \Rightarrow\ (v \in write^s(\xi := bool(\mathcal{C}); S_p) \wedge write^s(\xi := bool(\mathcal{C}); T_p))) \\
= \{v | \forall (S_p, T_p) \cdot (S_p \in spaths(S) \wedge T_p \in spaths(T) \\
\qquad \Rightarrow\ (v \in write^s(S_p) \wedge v \in write^s(T_p))) \\
\forall (S_p, T_p) \cdot (S_p \in spaths(S) \wedge T_p \in spaths(T) \Rightarrow (v \in write^s(S_p) \vee v \in write^s(T_p))) \\
\quad \Leftrightarrow \\
\qquad \forall S_p \cdot (S_p \in spaths(S) \Rightarrow v \in write^s(S_p)) \\
\qquad\quad \wedge\ \forall T_p \cdot (T_p \in spaths(T) \Rightarrow v \in write^s(T_p)) \\
\quad \Leftrightarrow \\
\qquad v \in writeall(S) \wedge writeall(T)
\end{array}
$$

Donc, $writeall(\text{IF } \mathcal{C} \text{ THEN } S \text{ ELSE } T \text{ END}) = writeall(S) \cap writeall(T)$

Annexe C

Correction de la traduction

Annexe supprimée pour des raisons de confidentialité.

Annexe D

Correction de l'aplatissement

Lemme 4 (Mesure intégration substitution simple) *Soit T une substitution plate et T n'est pas une composition parallèle. Pour toute substitution simple de la forme $x := E$ où x est une variable et E une expression.*

On pose les définitions suivantes :

$mesure(y := F) = 0$

$mesure(\text{IF } Cond \text{ THEN } A \text{ ELSE } B \text{ END}) = mesure(A) + mesure(B) + 1$

Dans ces conditions, le résultat de l'intégration de la substitution simple $x := E$ dans T à la même mesure que T.

$$mesure(integrate(x := E, T)) = mesure(T)$$

La preuve se fait par induction sur la structure de la substitution T.

- Base.

  Le cas de base est le cas où T est une substitution simple de la forme $y := F$. Dans cas, $integrate(x := E, y := F) = y := [x := E]F$, et $mesure(y := [x := E]F) = 0$, comme la mesure de $y := F$. La propriété est vérifiée.

- Induction.

  Le seul cas d'induction à prendre en compte est la composition conditionnelle car T est plate et différente d'une composition parallèle.

  $integrate(x := E, \text{IF } Cond \text{ THEN } A \text{ ELSE } B \text{ END})$

  $= \text{IF } [x := E]Cond \text{ THEN } integrate(x := E, A) \text{ ELSE } integrate(x := E, B) \text{ END}$

  Par hypothèse d'induction, on a $mesure(integrate(x := E, A)) = mesure(A)$ et $mesure(integrate(x := E, B)) = mesure(B)$. Donc,

  $mesure(integrate(x := E, \text{IF } Cond \text{ THEN } A \text{ ELSE } B \text{ END}))$

  $= mesure(A) + mesure(B) + 1$

  $= mesure(\text{IF } Cond \text{ THEN } A \text{ ELSE } B \text{ END})$

  La propriété est donc vérifiée

□(Lemme 4)

Lemme 5

$$mesure(\|_{v \in W_B} \xi_v := v) \leq mesure(B)$$

□(Lemme 5)

Lemme 6 (Support en écriture d'une intégration)

$$write(integrate(S, T)) = write(T)$$

La preuve est faite par induction sur S. La mesure utilisée pour l'induction est la suivante :

$mesure(y := F) = 0$

$mesure(\text{IF } Cond \text{ THEN } A \text{ ELSE } B \text{ END}) = mesure(A) + mesure(B) + 1$

$mesure(A \| B) = mesure(A) + mesure(B) + 1$

- Base : $mesure(S) = 0$.
  Le seul cas est quand S et T sont deux substitutions simples.
  $write(integrate(x := E, y := F)) = write(y := [x := E]F) = \{y\} = write(y := F)$
- Induction : propriété vraie pour $mesure(S) \leq n$, on prouve la propriété pour $mesure(S) = n + 1$.
  Deux cas sont à étudier : composition conditionnelle et composition parallèle.
  - composition conditionnelle
    $write(integrate(\text{IF } Cond \text{ THEN } A \text{ ELSE } B \text{ END}, T))$
    $= write(\text{IF } Cond \text{ THEN } integrate(A,T) \text{ ELSE } integrate(B,T) \text{ END})$
    $= write(integrate(A,T)) \cup write(integrate(B,T))$
    Comme $mesure(A) \leq n$ et $mesure(B) \leq n$, par hypothèse d'induction on a
    $= write(T) \cup write(T)$
    $= write(T)$
  - composition parallèle
    $write(integrate(A \| B, T))$
    $= write(integrate_{v \in W_B}(\xi_v := v, integrate(B, integrate(integrate_{v \in W_B}(v := \xi_v, A), T))))$
    $mesure(\|_{v \in W_B} \xi_v := v) \leq n$ (Lemme 5)
    $= write(integrate(B, integrate(integrate_{v \in W_B}(v := \xi_v, A), T)))$
    $mesure(B) \leq n$
    $= write(integrate(integrate_{v \in W_B}(v := \xi_v, A), T))$
    $mesure(integrate_{v \in W_B}(v := \xi_v, A)) \leq n$ (lemme 4)
    $= write(T)$

□(Lemme 6)

Lemme 7 (Ensemble support en écriture d'un aplatissement)

$$write(flat(S)) = write(S)$$

Par induction sur la structure de S.

- Base.
  Le cas de base est la substitution simple de la forme $x := E$. Dans ce cas, $flat(x := E) \equiv x := E$. La propriété est évidente.
- Induction.
  Trois cas : compositions conditionnelle, parallèle et séquentielle
  - composition conditionnelle (on a en fait pas besoin de l'induction dans ce cas)
    $write(flat(S))$
    $= write(flat(\text{IF } Cond \text{ THEN } A \text{ ELSE } B \text{ END}))$
    $= write(\|_{v \in W_A \cup W_B} \text{IF } Cond \text{ THEN } extract(v, flat(A)) \text{ ELSE } extract(v, flat(B)) \text{ END})$
    $= \bigcup_{v \in W_A \cup W_B} (write(extract(v, flat(A))) \cup write(extract(v, flat(B))))$
    $= \bigcup_{v \in W_A \cup W_B} (\{v\} \cup \{v\})$
    $= W_A \cup W_B$
    $= write(\text{IF } Cond \text{ THEN } A \text{ ELSE } B \text{ END})$
    $= write(S)$
  - composition parallèle
    $write(flat(S))$
    $= write(flat(A \parallel B))$
    $= write(flat(A) \parallel flat(B))$
    $= write(flat(A) \parallel flat(B))$
    $= write(flat(A)) \parallel write(flat(B))$
    Par hypothèse d'induction
    $= write(A) \parallel write(B)$
    $= write(A \parallel B)$
    $= write(S)$

- composition séquentielle

$write(flat(S))$
$= write(flat(A\,;B))$
$= write(S_{/W_A - W_B} \parallel integrate(flat(A), flat(B)))$
$= write(S_{/W_A - W_B}) \cup write(integrate(flat(A), flat(B)))$

En utilisant le lemme 6 on a $write(integrate(flat(A), flat(B))) = write(flat(B))$ et par définition $write(S_{/W_A - W_B}) = W_A - W_B$.

$= W_A - W_B \cup write(flat(B))$

Par hypothèse d'induction

$= W_A - W_B \cup W_B$
$= W_A \cup W_B$
$= write(A\,;B)$
$= write(S)$

□(Lemme 7)

Lemme 8 (Correction de l'extraction)
Dans un modèle dont l'ensemble des variables est V, pour toute substitution S plate dans ce modèle, toute variable v de V et toute variable z externe au modèle ($x \notin V$, v est une nouvelle variable, la substitution obtenue par extraction de v depuis S a le même effet que S sur v :

$$v \in V \wedge z \notin V \Rightarrow [extract(v, S)](v = z) \equiv [S](v = z)$$

On notera que dans tous les cas, S ou $extract(v, S)$ ne peut avoir aucun effet sur la variable z puisqu'elle est étrangère au modèle. La preuve se fait par induction sur la structure parallèle de S, c'est-à-dire que l'on considère $\parallel$ comme un constructeur, les autres substitutions étant des cas de base de l'induction.

- Cas de base.

  La substitution S est plate, il y a deux cas de base : la substitution simple $S \equiv x := E$, et la substitution conditionnelle. Dans les deux cas, la preuve est la même.

  Comme S est plate, on sait qu'il existe une variable x du modèle telle que $W_S = W_A = W_B = \{x\}$.

  Nous avons deux cas possible, soit $x \neq v$, soit $x = v$. Dans le premier cas, $extract(v, S) \equiv v := v$ car $v \notin W_S$, donc $[extract(v, S)](v = z) \equiv x = z$. Par ailleurs, comme $v \notin W_S$, on a aussi $[S](v = z) \equiv v = z$.

  Dans le deuxième cas ($x = v$), on a $extract(v, S) \equiv S$ par définition de $extract$, la propriété est alors évidente.

- Induction.

  Nous faisons maintenant l'hypothèse d'induction que la propriété est vraie pour toute substitution (plate) dont le nombre de constructeurs $\parallel$ est inférieur à un nombre n. On montre que la propriété reste vraie pour les substitutions (plates) dont le nombre de constructeurs est $n + 1$. Ces substitutions ne peuvent avoir qu'un seule forme : $A \parallel B$, où A et B sont deux substitutions dont le nombre de constructeurs $\parallel$ est inférieur ou égal à n.

  Le calcul de $extract(v, A \parallel B)$ amènent à trois cas possible :

  - $extract(v, A \parallel B) \equiv extract(v, A)$ si $v \in W_A$
  - $extract(v, A \parallel B) \equiv extract(v, B)$ si $v \in W_B$
  - $extract(v, A \parallel B) \equiv v := v$ si $v \notin W_A \cup W_B$

  Dans les deux premiers cas, on applique l'hypothèse d'induction. Prenons le premier cas en faisant l'hypothèse $v \in W_A$. Dans ce cas $[extract(v, A \parallel B)](v = z) \equiv [extract(v, A)](v = z) \equiv_{hyp.} [A](v = z)$. Par ailleurs on sait que $[A \parallel B](v = z) \equiv [A](v = z)$ puisque par hypothèse $v \in W_A$. Le deuxième est prouvé de la même façon.

  Le troisième cas, en faisant l'hypothèse $v \notin W_A \cup W_B$, conduit à $[extract(v, A \parallel B)](v = z) \equiv v := v](v = z) \equiv v = z$. Par ailleurs on sait que $[A \parallel B](v = z) \equiv v = z$ puisque $v \notin W_A \cup W_B$. La propriété est donc prouvée.

□(Lemme 8)

Lemme 9 (effet d'une intégration sur une variable) *Soient A et B deux substitutions plates. Soit v une variable du modèle écrite par B et z et une nouvelle variable qui ne fait pas partie du modèle. On prouve la propriété suivante :*

$$[A;B](v=z) \Leftrightarrow [integrate(A,B))(v=z)$$

On prouve cette propriété par induction sur le couple (A,B) avec la mesure $mesure((A,B)) = mesure(A) + mesure(B)$.

- Base : $(A,B) = (x := E, v := F)$.
- Induction.
  - $(x := E, \text{IF } Cond \text{ THEN } G \text{ ELSE } H \text{ END})$
    $[integrate(A,B))(v=z)$
    $\Leftrightarrow [(integrate(x := E, \text{IF } Cond \text{ THEN } G \text{ ELSE } H \text{ END})](v=z)$
    $\Leftrightarrow [\text{IF } [x := E]Cond \text{ THEN } integrate(x := E, G) \text{ ELSE } integrate(x := E, H) \text{ END}](v=z)$
    $\Leftrightarrow ([x := E]Cond \Rightarrow [integrate(x := E, G)](v = z)) \wedge (\neg[x := E]Cond \Rightarrow [integrate(x := E, G)](v = z))$
    Par hypothèse d'induction, $[x := E; G](v=z) \Leftrightarrow integrate(x := E, G)](v=z))$ et $[x := E; H](v = z) \Leftrightarrow integrate(x := E, H)](v=z))$
    $\Leftrightarrow ([x := E]Cond \Rightarrow [x := E; G](v=z)) \wedge (\neg[x := E]Cond \Rightarrow [x := E; G](v=z))$
    $\Leftrightarrow [\text{IF } [x := E]Cond \text{ THEN } x := E; G \text{ ELSE } x := E; H \text{ END}](v=z)$
    $\Leftrightarrow [x := E \ ; \ \text{IF } Cond \text{ THEN } G \text{ ELSE } H \text{ END}](v=z)$
    $\Leftrightarrow [A;B](v=z)$
  - $(\text{IF } Cond \text{ THEN } G \text{ ELSE } H \text{ END}, B)$
    $[integrate(A,B))(v=z)$
    $\Leftrightarrow [(integrate(\text{IF } Cond \text{ THEN } G \text{ ELSE } H \text{ END}, B)](v=z)$
    $\Leftrightarrow [\text{IF } Cond \text{ THEN } integrate(G,B) \text{ ELSE } integrate(H,B) \text{ END}](v=z)$
    $\Leftrightarrow (Cond \Rightarrow [integrate(G,B)](v=z)) \wedge (\neg Cond \Rightarrow [integrate(H,B)](v=z))$
    $\Leftrightarrow (Cond \Rightarrow [G;B](v=z)) \wedge (\neg Cond \Rightarrow [H;B](v=z))$
    $\Leftrightarrow [\text{IF } Cond \text{ THEN } G; B \text{ ELSE } H; B \text{ END}](v=z)$
    $\Leftrightarrow [\text{IF } Cond \text{ THEN } G \text{ ELSE } H \text{ END} \ ; \ B](v=z)$
    $\Leftrightarrow [A;B](v=z)$
  - $(A, G \parallel H)$ pour toute substitution plate A.
    $integrate(A, G \parallel H) = integrate(A,G) \parallel integrate(A,H)$
    $[integrate(A,B))(v=z)$
    $\Leftrightarrow [integrate(A, G \parallel H)(v=z)$
    $\Leftrightarrow [integrate(A,G) \parallel integrate(A,H)](v=z)$
    Comme $integrate(A,G)$ et $integrate(A,H)$ ont des ensembles supports en écriture différents, on a

    $$[integrate(A,G) \parallel integrate(A,H)](v=z) \Leftrightarrow [integrate(A,G)](v=z)$$

    ou

    $$[integrate(A,G) \parallel integrate(A,H)](v=z) \Leftrightarrow [integrate(A,H)](v=z)$$

    Faisons l'hypothèse de la première équivalence, la preuve est la même dans le deuxième cas.
    $\Leftrightarrow [integrate(A,G)](v=z)$ (hypothèse : $v \in W_G$)
    $\Leftrightarrow [A;G](v=z)$
    $\Leftrightarrow [A;G \| H](v=z)$ (hypothèse : $v \in W_G$)
    $\Leftrightarrow [A;B](v=z)$
  - $(G \parallel H, B)$
    $[integrate(A,B)](v=z)$
    $\Leftrightarrow [integrate(G \parallel H, B)](v=z)$
    Comme, pour des raisons de confidentialité, nous n'avons pas donné la définition de $integrate(G \parallel H, B)$, nous ne pouvons pas donner cette preuve.
    $\Leftrightarrow [G \| H; B](v=z)$
    $\Leftrightarrow [A;B](v=z)$

□(Lemme 9)

Propriété 22 (Correction de l'aplatissement) *Pour un modèle donné dont l'ensemble des variables est noté V, pour toute substitution S dans ce modèle, les substitutions S et $flat(S)$ sont égales.*

$$\forall Q.(Q \in \mathcal{P} \Rightarrow [flat(S)]Q \equiv [S]Q$$

La preuve de fait par induction sur la structure de la substitution S. Les constructeurs sont les opérateurs de composition conditionnelle, parallèle et séquentielle. On indique ci-dessous la mesure utilisée pour l'induction :

$mesure(x := E) = 0$
$mesure(\text{IF } Cond \text{ THEN } A \text{ ELSE } B \text{ END}) = max(mesure(A), mesure(B)) + 1$
$mesure(\text{IF } Cond \text{ THEN } A \text{ END}) = mesure(A) + 1$
$mesure(A \| B) max(mesure(A), mesure(B)) + 1$
$mesure(A; B) max(mesure(A), mesure(B)) + 1$

- Cas de base.

  Le seul cas de base est celui de la substitution simple $x := E$, où x est une variable du modèle et E une expressions sur les variables du modèle. Comme, par définition de $flat$, $flat(x := E) \equiv x := E$, la propriété est évidente.

- Induction.

  - composition conditionnelle

    $flat(\text{IF } Cond \text{ THEN } A \text{ ELSE } B \text{ END})$

    $$= \|_{v \in W_A \cup W_B} \text{IF } Cond \text{ THEN } extract(v, flat(A)) \text{ ELSE } extract(v, flat(B)) \text{ END}$$

    On numérote toutes les variables de l'ensemble $W_A \cup W_B$ de 1 à n, où $n = card(W_A \cup W_B)$. On a alors $\{v_1, ..., v_n\} = W_A \cup W_B$ avec $v_i \neq v_j$ pour i différent de j.

    $flat(\text{IF } Cond \text{ THEN } A \text{ ELSE } B \text{ END})$

    $$= \|_{k \in 1..n} \text{IF } Cond \text{ THEN } extract(v_k, flat(A)) \text{ ELSE } extract(v_k, flat(B)) \text{ END}$$

    La propriété $Q(v_1, ..., v_n)$ peut se décomposer de la façon suivante, où les variables z_k sont de nouvelles variables n'appartenant pas à V et $Q(z_1, ..., z_n)$ est Q dans lequel toute occurrence de v_k est remplacée par z_k.

    $$Q(v_1, ..., v_n) \equiv Q(z_1, ..., z_n) \wedge \bigwedge_{k \in 1..n} (v_k = z_k)$$

    Prenons un nombre k dans l'intervalle $1..n$. Nous avons la propriété suivante car v_k ne faite pas partie de l'ensemble support en écriture de $\|_{k \in (1..n)-k} \text{IF } Cond \text{ THEN } extract(v_k, flat(A)) \text{ ELSE } extract(v_k, flat(B)) \text{ END}$

    $[flat(\text{IF } Cond \text{ THEN } A \text{ ELSE } B \text{ END})](v_k = z_k)$

    $$\equiv [\text{IF } Cond \text{ THEN } extract(v_k, flat(A)) \text{ ELSE } extract(v_k, flat(B)) \text{ END}](v_k = z_k)$$

    Par ailleurs, d'après le lemme 8, comme $flat(A)$ et $flat(B)$ sont par définition plates, nous avons les propriétés

    $$extract(v, flat(A))](v_k = z_k) \equiv [flat(A)](v_k = z_k)$$
    $$extract(v, flat(B))](v_k = z_k) \equiv [flat(B)](v_k = z_k)$$

    D'où la propriété suivante, pour tout $k \in 1..n$.

    $[flat(\text{IF } Cond \text{ THEN } A \text{ ELSE } B \text{ END})](v_k = z_k)$

    $$\equiv [\text{IF } Cond \text{ THEN } flat(A) \text{ ELSE } flat(B) \text{ END}](v_k = z_k)$$
    $$\equiv (Cond \Rightarrow [flat(A)](v_k = z_k)) \wedge (\neg Cond \Rightarrow [flat(B)](v_k = z_k))$$

    Comme la mesure de A et la mesure de B sont inférieures[5] à la mesure de S, on peut appliquer l'hypothèse d'induction, on obtient les deux propriétés suivantes.

[5] Remarquons au passage que les mesures de $flat(A)$ et $flat(B)$ ne sont pas nécessairement inférieurs à la mesure de S.

$$[A](v_k = z_k) \equiv [flat(A)](v_k = z_k)$$
$$[B](v_k = z_k) \equiv [flat(B)](v_k = z_k)$$

Nous poursuivons en effectuant les remplacements.
$[flat(\text{IF } Cond \text{ THEN } A \text{ ELSE } B \text{ END})](v_k = z_k)$
$\equiv (Cond \Rightarrow [A](v_k = z_k) \wedge (\neg Cond \Rightarrow [B](v_k = z_k))$
$\equiv [\text{IF } Cond \text{ THEN } A \text{ ELSE } B \text{ END}](v_k = z_k)$
Nous avons donc montré que pour toute variable v_k, et pour toute substitution conditionnelle S :

$$[S](v_k = z_k) = [flat(S)](v_k = z_k)$$

et par conséquent

$$[S]\left(\bigwedge_{k\in 1..n}(v_k = z_k)\right) \equiv [flat(S)]\left(\bigwedge_{k\in 1..n}(v_k = z_k)\right)$$

Par ailleurs on sait que par construction, $Q(z_1, ..., z_n)$ ne contient aucune variable susceptible d'être modifiée par S ou par $flat(S)$[6], donc

$$[S]Q(z_1, ..., z_n) = Q(z_1, ..., z_n) \equiv [flat(S)]Q(z_1, ..., z_n)$$

A partir des deux dernières propriétés, on peut reconstituer Q et obtient le résultat attendu :

$$[S]\left(Q(z_1, ..., z_n) \wedge \bigwedge_{k\in 1..n}(v_k = z_k)\right) \equiv [flat(S)]\left(Q(z_1, ..., z_n) \wedge \bigwedge_{k\in 1..n}(v_k = z_k)\right)$$

c'est-à-dire

$$[S]Q = [flat(S)]Q$$

– composition parallèle
Pour la composition parallèle on ne prouve pas l'égalité de deux substitutions en utilisant la propriété avec Q mais par l'équivalence des leurs prédicats avant-après.
Comme $flat(A \parallel B) = flat(A) \parallel flat(B)$, on calcule le prédicat avant-après de cette substitution :
$pred_{write(flat(A))\cup write(flat(B))}(flat(A) \parallel flat(B))$
$= pred_{write(flat(A))}(flat(A)) \wedge pred_{write(flat(B))}(flat(B))$
En utilisant le lemme 7, on sait que $write(flat(A)) = W_A$ et $write(flat(B)) = W_B$.
$= pred_{W_A}(flat(A)) \wedge pred_{W_B}(flat(B))$
Par hypothèse d'induction, on a $pred_{W_A}(flat(A)) = pred_{W_A}(A)$ et $pred_{W_B}(flat(B)) = pred_{W_B}(B)$
$= pred_{W_A}(A) \wedge pred_{W_B}(B)$
$= pred_{W_A \cup W_B}(A \parallel B)$
$= pred_{W_S}(S)$

– composition séquentielle
On rappelle la définition de $flat(A, B)$:

$$flat(A; B) = A_{/W_A - W_B} \parallel integrate(flat(A), flat(B))$$

Intuitivement $integrate(A, B)$ est une substitution qui a le même ensemble support en écriture que $A; B$ et qui est équivalent à $A; B$ sur cet ensemble support. Par rapport aux variables écrites par B, $integrate(A, B)$ est équivalent à $A; B$, mais pour les variables qui ne sont pas écrites par B, $integrate(A, B)$ est équivalent à *skip*.

[6]Ici nous n'utilisons pas le fait que S et $flat(S)$ ait le même ensemble support en écriture, mais le fait que les variables s_k ont été bien choisies

On note par w_b les variables écrites par B (éventuellement également écrite par A) et par x_b des variables (en même nombre que w_b) qui ne sont écrites ni par A ni par B (x_b est une nouvelle variable qui n'est pas dans le modèle). Le lemme 9 nous permet d'avoir la propriété suivante :

$$[A;B](x_b = w_b) \Leftrightarrow [integrate(A,B)](x_b = w_b) \tag{D.1}$$

Supposons qu'il existe des variables écrites par A mais pas par B, on les note w_{a-b}. On note par x_{a-b} des variables (en même nombre que w_b) qui ne sont écrites ni par A ni par B. Alors nous avons la propriété suivante :

$$[A;B](x_{a-b} = w_{a-b}) \Leftrightarrow [A](x_{a-b} = w_{a-b}) \tag{D.2}$$

Soit Q un prédicat sur les variables écrites par A et B, on note ce prédicat $Q(w_b, w_{a-b})$ où w_b est une variable écrite par B et w_{a-b} est une variable écrite par A mais pas par B (en supposant qu'il en existe). Le prédicat $Q(w_b, w_{a-b})$ peut être récrit $x_b = w_b \wedge x_{a-b} = w_{a-b} \wedge Q(x_b, x_{a-b})$, où x_b et x_{a-b} sont des nouvelles variables. Il y a une quantificateur existentiel implicite $\exists(x_b, x_{a-b}) \cdot (...)$ que nous n'écrivons pas. On sait que

$$[A;B]Q(w_b, w_{a-b}) \Leftrightarrow [A;B](x_b = w_b) \wedge [A;B](x_{a-b} = w_{a-b}) \wedge [A;B]Q(x_b, x_{a-b}) \tag{D.3}$$

Supposons que nous avons une substitution A' telle que son ensemble support soit $W_A - W_B$ et que, pour tout prédicat P sur le même ensemble, on ait la propriété suivante :

$$[A']P \Leftrightarrow [A]P \tag{D.4}$$

Pour toute substitution A qui a un ensemble support en écriture différent de celui de A' (W_B par exemple), on peut dire que $[A' \| A](x_b = w_b) \Leftrightarrow [A](x_b = w_b)$ car ni x_b ni w_b ne sont écrites par A'. En particulier, nous avons la propriété suivante :

$$[A' \| integrate(A,B)](x_b = w_b) \Leftrightarrow [integrate(A,B)](x_b = w_b) \tag{D.5}$$

De la même façon, puisque les variables x_{a-b} et w_{a-b} ne sont pas dans l'ensemble support en écriture de la substitution $integrate(A,B)$, nous avons la propriété suivante :

$$[A' \| integrate(A,B)](x_{a-b} = w_{a-b}) \Leftrightarrow [A'](x_{a-b} = w_{a-b}) \tag{D.6}$$

A partir de (D.1) et (D.5), on déduit (D.7), et de (D.2), (D.4) et (D.6) on déduit (D.8).

$$[A;B](x_b = w_b) \Leftrightarrow [A' \| integrate(A,B)](x_b = w_b) \tag{D.7}$$

$$[A;B](x_{a-b} = w_{a-b}) \Leftrightarrow [A' \| integrate(A,B)](x_{a-b} = w_{a-b}) \tag{D.8}$$

Finalement, de (D.3), (D.7), (D.8) et parce qu'il n'y a aucune variable écrite par $A' \| integrate(A,B)$ dans $Q(x_b, x_{a-b})$, on déduit la propriété (D.9).

$$[A;B]Q(w_b, w_{a-b}) \Leftrightarrow \begin{cases} [A' \| integrate(A,B)](x_b = w_b) \wedge \\ [A' \| integrate(A,B)](x_{a-b} = w_{a-b}) \wedge \\ [A' \| integrate(A,B)]Q(x_b, x_{a-b}) \end{cases} \tag{D.9}$$

A partir de (D.9), on peut recomposer Q pour obtenir la propriété suivante :

$$[A;B]Q(w_b, w_{a-b}) \Leftrightarrow [A' \| integrate(A,B)]Q(w_b, w_{a-b}) \tag{D.10}$$

Le prédicat Q est défini sur l'ensemble support en écriture de $A;B$ (qui est le même ensemble support que $A' \| integrate(A,B)$). On peut généraliser à tout prédicat Q car $A;B$ et $A' \| integrate(A,B)$ n'ont pas d'effet sur d'autres variables que celles de leur ensemble support. Nous avons également fait l'hypothèse que l'ensemble de variables $W_A - W_B$ n'est pas vide. Le cas où toutes les variables écrites par A sont aussi écrites par B est un cas plus simple qui conduit au même résultat.
Nous avons également fait l'hypothèse de l'existence de la substitution A' qui a $W_A - W_B$ comme ensemble support en écriture et telle que $[A']P \Leftrightarrow [A]P$ pour tout prédicat P sur le même ensemble. Ces conditions sont remplies par la substitution $A_{/W_A - W_B}$ utilisée dans la définition de $flat(A;B)$. Par conséquent, avec $A' \equiv A_{/W_A - W_B}$ et (D.10), on peut conclure que $flat(A;B)$ est équivalent à $A;B$.

$$[A;B]Q \Leftrightarrow [flat(A;B)]Q$$

□(Propriété 22)

Annexe E

Propriétés sur les vecteurs de bits

Cette machine est uniquement utilisée pour prouver un propriété utilisée dans le sixième raffinement de l'étude de cas du contrôleur de bus série.

Ce qui nous intéresse est la définition de *nb* en utilisant le *MID* : le *MID* est simplement la représentation binaire de *nb*. La propriété la plus importante est la dernière. Elle est utilisée dans le sixième raffinement lorsque nous raffinons la phase de compétition.

On notera que la première assertion est un théorème d'induction [26] exprimé en logique du premier ordre, en utilisant la théorie des ensembles. Ce théorème a été prouvé et réutilisé pour prouver une large partie des autres assertions.

MACHINE

codeserial

SETS

ND

DEFINITIONS

$BIT == \{0,1\}$

CONSTANTS

CHAR,
MID,
MAX_BIT_CHAR , *nb*,*power2*, *nubof*, *suc*

PROPERTIES

$nb \in ND \rightarrowtail \mathbb{N} \wedge$
$MAX\_BIT\_CHAR \in \mathbb{N}_1 \wedge$
$CHAR = 0 \mathinner{..} MAX\_BIT\_CHAR \rightarrow BIT \wedge$
$\forall\, cc.(cc \in CHAR \Rightarrow cc(0) = 0 \wedge$
$cc(MAX\_BIT\_CHAR) = 1) \wedge$
$MID \in ND \rightarrowtail CHAR \wedge$

$suc \in \mathbb{N} \rightarrow \mathbb{N} \wedge$
$\forall\, xx.(xx \in \mathbb{N} \Rightarrow suc(xx)=xx+1) \wedge$

$power2 \in \mathbb{N} \rightarrow \mathbb{N} \wedge$
$power2(0) = 1 \wedge$
$\forall\, kk.(kk \in \mathbb{N} \Rightarrow power2(kk+1)=2 \times power2(kk)) \quad \wedge$

$nubof \in CHAR \times \mathbb{N} \rightarrow \mathbb{N} \wedge$

$\forall\, code.(code \in CHAR \Rightarrow nubof(code, 0)=0) \wedge$
$\forall\, (code, posn).(code \in CHAR \wedge posn \in \mathbb{N}$

$\Rightarrow$ *nubof*(*code*, *posn*+1)= *code*(*posn*+1)+2 × *nubof*(*code*,*posn*)) ∧

∀ *nd*.(*nd* ∈ *ND* ⇒ *nb*(*nd*) = *nubof*(*MID*(*nd*), *MAX_BIT_CHAR*))

ASSERTIONS

∀ *PP*.(*PP* ⊆ ℕ ∧
0 ∈ *PP* ∧
suc[*PP*] ⊆ *PP*
⇒
ℕ ⊆ *PP*);

∀ *nn*.(*nn* ∈ ℕ ⇒ *power2*(*nn*) ≥ 1);

∀ (*code*, *posn*).(*code* ∈ *CHAR* ∧ *posn* ∈ ℕ
⇒ (*posn* ≤ *MAX_BIT_CHAR* ⇒ *nubof*(*code*,*posn*) < *power2*(*posn*)));

∀ (*code1*, *code2*, *posn*).(*code1* ∈ *CHAR* ∧ *code2* ∈ *CHAR* ∧ *posn* ∈ 0 .. *MAX_BIT_CHAR* ∧
∀ *pp*.(*pp* ∈ 0 .. *posn* ⇒ *code1*(*pp*)=*code2*(*pp*))
⇒ *nubof*(*code1*, *posn*)=*nubof*(*code2*, *posn*));

∀ (*code1*, *code2*, *posn*).(*code1* ∈ *CHAR* ∧ *code2* ∈ *CHAR* ∧ *posn* ∈ 0 .. *MAX_BIT_CHAR* ∧
∀ *pp*.(*pp* ∈ 0 .. *posn*-1 ⇒ *code1*(*pp*)=*code2*(*pp*)) ∧
code1(*posn*)=0 ∧ *code2*(*posn*)=1
⇒ *nubof*(*code1*, *posn*)<*nubof*(*code2*, *posn*));

∀ (*code1*, *code2*, *posn*).(*code1* ∈ *CHAR* ∧ *code2* ∈ *CHAR* ∧ *posn* ∈ 0 .. *MAX_BIT_CHAR* ∧
∀ *pp*.(*pp* ∈ 0 .. *posn*-1 ⇒ *code1*(*pp*)=*code2*(*pp*)) ∧
code1(*posn*)=0 ∧ *code2*(*posn*)=1
⇒ *nubof*(*code2*, *posn*)=1+*nubof*(*code1*, *posn*));

∀ (*code*, *posn*, *uu*, *nn*).(*code* ∈ *CHAR* ∧ *posn* ∈ 0 .. *MAX_BIT_CHAR* ∧
uu ∈ *BIT* ∧ *nn* ∈ *posn*+1 .. *MAX_BIT_CHAR*
⇒ *nubof*(*code* ⩤ {*nn* ↦ *uu*},*posn*)=*nubof*(*code*,*posn*));

∀ (*code1*, *code2*, *posn*, *uu*, *vv*).(*code1* ∈ *CHAR* ∧ *code2* ∈ *CHAR* ∧
posn ∈ 0 .. *MAX_BIT_CHAR*-1 ∧ *uu* ∈ *BIT* ∧ *vv* ∈ *BIT*
⇒ (*nubof*(*code2*,*posn*)<*nubof*(*code1*,*posn*)
⇒ *nubof*(*code2* ⩤ {*posn*+1 ↦ *uu*},*posn*+1)<*nubof*(*code1* ⩤ {*posn*+1 ↦ *vv*},*posn*+1)));

∀ (*code1*, *code2*, *posn*).(*code1* ∈ *CHAR* ∧ *code2* ∈ *CHAR* ∧ *posn* ∈ 0 .. *MAX_BIT_CHAR*-1
⇒ (*nubof*(*code2*,*posn*)<*nubof*(*code1*,*posn*) ⇒ *nubof*(*code2*,*posn*+1)<*nubof*(*code1*,*posn*+1)));

∀ (*code1*, *code2*, *posn*, *CCC*).(*code1* ∈ *CHAR* ∧ *code2* ∈ *CHAR* ∧
posn ∈ 0 .. *MAX_BIT_CHAR* ∧ *CCC* ∈ *posn*+1 .. *MAX_BIT_CHAR*
⇒ (*nubof*(*code2*,*posn*)<*nubof*(*code1*,*posn*) ⇒ *nubof*(*code2*, *CCC*)<*nubof*(*code1*, *CCC*)));

∀ (*code1*, *code2*, *posn*).(*code1* ∈ *CHAR* ∧ *code2* ∈ *CHAR* ∧ *posn* ∈ 0 .. *MAX_BIT_CHAR*
⇒ (*nubof*(*code2*,*posn*)<*nubof*(*code1*,*posn*)
⇒ *nubof*(*code2*,*MAX_BIT_CHAR*)<*nubof*(*code1*,*MAX_BIT_CHAR*)));

∀ (*code1*, *code2*, *posn*).(*code1* ∈ *CHAR* ∧ *code2* ∈ *CHAR* ∧ *posn* ∈ 0 .. *MAX_BIT_CHAR* ∧
∀ *pp*.(*pp* ∈ 0 .. *posn*-1 ⇒ *code1*(*pp*)=*code2*(*pp*)) ∧
code1(*posn*)=0 ∧ *code2*(*posn*)=1

$\Rightarrow$ *nubof*(*code1*,*MAX_BIT_CHAR*)<*nubof*(*code2*,*MAX_BIT_CHAR*)) ;

$$\forall\ (nd1,\ nd2,\ posn).(nd1 \in ND \wedge nd2 \in ND \wedge posn \in 0\ ..\ MAX\_BIT\_CHAR \wedge ($$
$$\forall\ pp.(pp \in 0\ ..\ posn\text{-}1 \Rightarrow (MID(nd1))(pp) = (MID(nd2))(pp))) \wedge$$
$$(MID(nd1))(posn) = 0 \wedge (MID(nd2))(posn) = 1$$
$$\Rightarrow nb(nd1) < nb(nd2)\ )$$

END

Annexe F

Système B implantable de l'étude de cas du contrôleur de bus

VARIABLES

Variables d'état des nœuds

$Booleanwr$, $BooleanCP$,
$TotalAT$, $IsInDomAT$,
pos, $TotalLONGMESSAGE$, $Totalpos\_message$, $Totalpos\_msg\_char$, $TotalMESSAGE$, fc,

Inputs and Outputs

in_tt ,
in_lmsg, in_msg,
in_wri,
out,

Variables d'ordonnancement

$E4$,$E2$, $VER2$, WRI,

Variable modélisant le bus

$bbus$

INITIALISATION

$BooleanCP := ND \times \{FALSE\}$ ||
$Booleanwr := ND \times \{FALSE\}$ ||
$TotalLONGMESSAGE :\in ND \rightarrow \mathbb{N}_1$ ||
$TotalMESSAGE :\in ND \rightarrow (0 \,..\, LONG\_MAX\_MESSAGE\text{-}1 \rightarrow CHAR)$ ||
$Totalpos\_message :\in ND \rightarrow \mathbb{N}$ ||
$Totalpos\_msg\_char :\in ND \rightarrow 0 \,..\, MAX\_BIT\_CHAR$ ||
$TotalAT$, $IsInDomAT := ND \times \{0\}$, $ND \times \{FALSE\}$ ||
pos, $fc := ND \times \{0\}$, $ND \times \{0\}$ ||
$in\_wri :\in ND \rightarrow BOOL$ ||
$in\_tt :\in ND \rightarrow \mathbb{N}_1$ ||
$in\_msg :\in ND \rightarrow (0 \,..\, LONG\_MAX\_MESSAGE\text{-}1 \rightarrow CHAR)$ ||
$in\_lmsg :\in ND \rightarrow 1 \,..\, LONG\_MAX\_MESSAGE$ ||
$out :\in ND \rightarrow BIT$ ||
$VER2$, $E2$, WRI, $E4 := \varnothing$, $\varnothing$, $\varnothing$, ND ||
$bbus :\in BIT$

EVENTS

bus_write = ANY nd WHERE
$nd \in WRI \wedge$

$BooleanCP(nd) = TRUE \quad \wedge \neg (IsInDomAT(nd) = TRUE)$
THEN
$pos(nd) := pos(nd) + 1 \parallel$
$out(nd) := (MID(nd))(pos(nd)+1) \parallel$
$E2 := E2 \cup \{nd\} \parallel$
$WRI := WRI - \{nd\}$
END ;

bus_write_first = ANY nd WHERE
$nd \in WRI \wedge$
$IsInDomAT(nd) = TRUE \wedge TotalAT(nd)=0$
THEN
$out(nd) := 0 \parallel$
$WRI := WRI - \{nd\} \parallel$
$E2 := E2 \cup \{nd\}$
END ;

bus_write_nothing= ANY nd WHERE
$nd \in WRI \wedge$
$IsInDomAT(nd) = TRUE \wedge \neg (TotalAT(nd)=0)$
THEN
$out(nd) := 1 \parallel$
$WRI := WRI - \{nd\} \parallel$
$E2 := E2 \cup \{nd\}$
END ;

send_next_char = ANY nd WHERE
$nd \in WRI \wedge$
$Booleanwr(nd) = TRUE \wedge$
$\neg (Totalpos\_message(nd) = TotalLONGMESSAGE(nd)) \wedge$
$Totalpos\_msg\_char(nd) = MAX\_BIT\_CHAR \wedge$
$\neg (BooleanCP(nd) = TRUE) \wedge \neg (IsInDomAT(nd) = TRUE)$
THEN
$out(nd) := TotalMESSAGE(nd)(Totalpos\_message(nd))(Totalpos\_msg\_char(nd)) \parallel$
$Totalpos\_message(nd) := Totalpos\_message(nd)+1 \parallel$
$Totalpos\_msg\_char(nd) := 0 \parallel$
$WRI := WRI - \{nd\} \parallel$
$E2 := E2 \cup \{nd\}$
END ;

send = ANY nd WHERE
$nd \in WRI \wedge$
$Booleanwr(nd) = TRUE \wedge \neg (Totalpos\_message(nd) = TotalLONGMESSAGE(nd)) \wedge$
$\neg (Totalpos\_msg\_char(nd) = MAX\_BIT\_CHAR) \wedge$
$\neg (BooleanCP(nd) = TRUE) \wedge \neg (IsInDomAT(nd) = TRUE)$
THEN
$Totalpos\_msg\_char(nd) := Totalpos\_msg\_char(nd)+1 \parallel$
$out(nd) := TotalMESSAGE(nd)(Totalpos\_message(nd))(Totalpos\_msg\_char(nd)) \parallel$
$WRI := WRI - \{nd\} \parallel$
$E2 := E2 \cup \{nd\}$
END ;

sender_nothing = ANY nd WHERE
$nd \in WRI \wedge$
$Booleanwr(nd) = TRUE \wedge Totalpos\_message(nd) = TotalLONGMESSAGE(nd) \wedge$
$\neg (BooleanCP(nd) = TRUE) \wedge \neg (IsInDomAT(nd) = TRUE)$
THEN
$out(nd) := 1 \parallel$

$WRI := WRI - \{nd\} \parallel$
$E2 := E2 \cup \{nd\}$
END ;

bus_nothing= ANY nd WHERE
$nd \in WRI \wedge$
$\neg (Booleanwr(nd) = TRUE) \wedge$
$\neg (BooleanCP(nd) = TRUE) \wedge$
$\neg (IsInDomAT(nd) = TRUE)$
THEN
$out(nd) := 1 \parallel$
$WRI := WRI - \{nd\} \parallel$
$E2 := E2 \cup \{nd\}$
END ;

bus_p =
WHEN
$WRI = \varnothing \wedge E2 \neq \varnothing$
THEN
$E2 := \varnothing \parallel$
$VER2 := ND \parallel$
$bbus := \min(out[ND])$
END ;

rise = ANY nd WHERE
$nd \in VER2 \wedge fc(nd)=0 \wedge \quad E2 = \varnothing \wedge$
$\neg (IsInDomAT(nd) = TRUE) \wedge$
$\neg (BooleanCP(nd) = TRUE) \wedge$
$\neg (Booleanwr(nd) = TRUE) \wedge$
$bbus=1 \wedge$
$in\_wri(nd) = TRUE$
THEN
$TotalAT(nd) := in\_tt(nd) \parallel$
$IsInDomAT(nd) := TRUE \parallel$
$pos(nd) := 0 \parallel$
$VER2 := VER2 - \{nd\} \parallel E4 := E4 \cup \{nd\} \parallel$
$fc(nd) := (1\text{-}bbus) \times COUNT\_FREE + bbus \times max(\{fc(nd) - 1,0\})$
END ;

rise_count = ANY nd WHERE
$nd \in VER2 \wedge \quad E2 = \varnothing \wedge$
$IsInDomAT(nd) = TRUE \wedge$
$\neg (TotalAT(nd) = 0) \wedge$
$bbus=1$
THEN
$TotalAT(nd) := TotalAT(nd)\text{-}1 \parallel$
$VER2 := VER2 - \{nd\} \parallel E4 := E4 \cup \{nd\} \parallel$
$fc(nd) := (1\text{-}bbus) \times COUNT\_FREE + bbus \times max(\{fc(nd) - 1,0\})$
END ;

switch_ko = ANY nd WHERE
$nd \in VER2 \wedge \quad E2 = \varnothing \wedge$
$IsInDomAT(nd)=TRUE \wedge \neg (TotalAT(nd)=0) \wedge$
$\neg (bbus = 1)$
THEN
$IsInDomAT(nd) := FALSE \parallel$
$VER2 := VER2 - \{nd\} \parallel E4 := E4 \cup \{nd\} \parallel$
$fc(nd) := (1\text{-}bbus) \times COUNT\_FREE + bbus \times max(\{fc(nd) - 1,0\})$

END ;

switch_ok = ANY *nd* WHERE
nd ∈ *VER2* ∧ *E2* = ∅ ∧
IsInDomAT(*nd*) = *TRUE* ∧ *TotalAT*(*nd*)=0
THEN
BooleanCP(*nd*) := *TRUE* ||
IsInDomAT(*nd*) := *FALSE* ||
VER2 := *VER2* - {*nd*} || *E4* := *E4* ∪ {*nd*} ||
fc(*nd*) := (1-*bbus*) × *COUNT_FREE* + *bbus* × *max*({*fc*(*nd*) - 1,0})
END ;

check_ok = ANY *nd* WHERE
nd ∈ *VER2* ∧ *E2* = ∅ ∧
BooleanCP(*nd*) = *TRUE* ∧ ¬ (*pos*(*nd*) = *MAX_BIT_CHAR*) ∧
out(*nd*) = *bbus* ∧
¬ (*IsInDomAT*(*nd*) = *TRUE*)
THEN
VER2 := *VER2* - {*nd*} || *E4* := *E4* ∪ {*nd*} ||
fc(*nd*) := (1-*bbus*) × *COUNT_FREE* + *bbus* × *max*({*fc*(*nd*) - 1,0})
END ;

check_ko = ANY *nd* WHERE
nd ∈ *VER2* ∧ *E2* = ∅ ∧
BooleanCP(*nd*) = *TRUE* ∧ ¬ (*pos*(*nd*) = *MAX_BIT_CHAR*) ∧
¬ (*out*(*nd*) = *bbus*) ∧
¬ (*IsInDomAT*(*nd*) = *TRUE*)
THEN
BooleanCP(*nd*) := *FALSE* ||
VER2 := *VER2* - {*nd*} || *E4* := *E4* ∪ {*nd*} ||
pos(*nd*) :=0 ||
fc(*nd*) := (1-*bbus*) × *COUNT_FREE* + *bbus* × *max*({*fc*(*nd*) - 1,0})
END ;

take = ANY *nd* WHERE
nd ∈ *VER2* ∧ *E2* = ∅ ∧
BooleanCP(*nd*) = *TRUE* ∧ *pos*(*nd*)=*MAX_BIT_CHAR* ∧
¬ (*IsInDomAT*(*nd*) = *TRUE*)
THEN
Booleanwr(*nd*) := *TRUE* ||
BooleanCP(*nd*) := *FALSE* ||
pos(*nd*) :=0 ||
Totalpos_message(*nd*) := 0 ||
Totalpos_msg_char(*nd*) := 0 ||
TotalLONGMESSAGE := *in_lmsg* ||
TotalMESSAGE := *in_msg* ||
VER2 := *VER2* - {*nd*} || *E4* := *E4* ∪ {*nd*} ||
fc(*nd*) := (1-*bbus*) × *COUNT_FREE* + *bbus* × *max*({*fc*(*nd*) - 1,0})
END ;

sender = ANY *nd* WHERE
nd ∈ *VER2* ∧ *E2* = ∅ ∧
Booleanwr(*nd*) = *TRUE* ∧ ¬ (*Totalpos_message*(*nd*) = *TotalLONGMESSAGE*(*nd*)) ∧
¬ (*BooleanCP*(*nd*) = *TRUE*) ∧ ¬ (*IsInDomAT*(*nd*) = *TRUE*)
THEN
VER2 := *VER2* - {*nd*} || *E4* := *E4* ∪ {*nd*} ||
fc(*nd*) := (1-*bbus*) × *COUNT_FREE* + *bbus* × *max*({*fc*(*nd*) - 1,0})

END ;

sender_w = ANY *nd* WHERE
nd ∈ *VER2* ∧ *E2* = ∅ ∧
Booleanwr(*nd*) = *TRUE* ∧ *Totalpos_message*(*nd*) = *TotalLONGMESSAGE*(*nd*) ∧
¬ (*fc*(*nd*)=1) ∧
¬ (*BooleanCP*(*nd*) = *TRUE*) ∧ ¬ (*IsInDomAT*(*nd*) = *TRUE*)
THEN
VER2 := *VER2* - {*nd*} ‖ *E4* := *E4* ∪ {*nd*} ‖
fc(*nd*) := (1-*bbus*) × *COUNT_FREE* + *bbus* × *max*({*fc*(*nd*) - 1,0})
END ;

free = ANY *nd* WHERE
nd ∈ *VER2* ∧ *E2* = ∅ ∧
Booleanwr(*nd*) = *TRUE* ∧ *Totalpos_message*(*nd*) = *TotalLONGMESSAGE*(*nd*) ∧
fc(*nd*)=1 ∧
¬ (*BooleanCP*(*nd*) = *TRUE*) ∧
¬ (*IsInDomAT*(*nd*) = *TRUE*)
THEN
Booleanwr(*nd*) := *FALSE* ‖
VER2 := *VER2* - {*nd*} ‖
E4 := *E4* ∪ {*nd*} ‖
fc(*nd*) := (1-*bbus*) × *COUNT_FREE* + *bbus* × *max*({*fc*(*nd*) - 1,0})
END ;

nothing = ANY *nd* WHERE
nd ∈ *VER2* ∧ *E2* = ∅ ∧
¬ (*BooleanCP*(*nd*) = *TRUE*) ∧ ¬ (*IsInDomAT*(*nd*) = *TRUE*) ∧
¬ (*Booleanwr*(*nd*) = *TRUE*) ∧
¬ (*fc*(*nd*)=0 ∧ *bbus*=1 ∧ *in_wri*(*nd*)= *TRUE*)
THEN
VER2 := *VER2* - {*nd*} ‖
E4 := *E4* ∪ {*nd*} ‖
fc(*nd*) := (1-*bbus*) × *COUNT_FREE* + *bbus* × *max*({*fc*(*nd*) - 1,0})
END ;

env = WHEN
E4 ≠ ∅ ∧ *VER2* = ∅
THEN
WRI := *ND* ‖ *E4* := ∅ ‖
in_wri :∈ *ND* → *BOOL* ‖
in_tt :∈ *ND* → ℕ1 ‖
in_msg :∈ *ND* → (0 .. *LONG_MAX_MESSAGE*-1 → *CHAR*) ‖
in_lmsg :∈ *ND* → 1 .. *LONG_MAX_MESSAGE*
END

END

Annexe G

Graphes d'événements

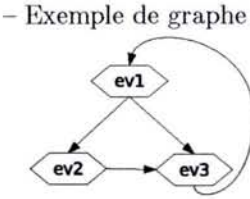
Il existe souvent un ordonnancement plus ou moins implicite entre les événements. Il nous a semblé pratique de représenter cet ordonnancement de manière graphique. Nous utilisons ces graphismes de manière informelle, pour avoir une vision globale du système. Ceci permet également à des électroniciens de voir le comportement du système sans avoir à lire le modèle B.

Nous avons utilisons deux sortes de graphismes, des réseaux de pétri et des graphes que nous appelons *graphes d'événements* qui pourraient se représenter par réseau de Petri mais qui sont moins formels et donc plus rapides à dessiner. Nous ne rappelons pas ici ce qu'est un réseau de Petri, on pourra se référer par exemple à [36]. Nous décrivons ici nos graphes d'événements, ils ne disposent d'aucune sémantique formelle, ils sont utilisés uniquement à titre d'illustration.

G.1 Modèles de base

La figure G.1 montre un modèle composé de trois événements. Chaque événement est représenté par un hexagone. Les transitions entre les événements montrent de quelle façon les événements peuvent se succéder. Dans l'exemple montré sur la figure G.1, l'événement "ev1" peut être succédé par "ev2" ou "ev3" alors que "ev2" est nécessairement succédé par "ev3".

FIG. G.1 – Exemple de graphe d'événement



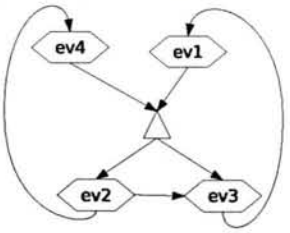
G.2 Triangle

Nous introduisons un nouveau symbole, un triangle, dans les graphes d'événements. Ce triangle permet de "factoriser" des transitions lorsque plusieurs événements ont les mêmes successeurs. Par exemple, sur la figure G.2, les événements "ev1" et "ev4" ont les mêmes successeurs ("ev2" et "ev3"). Pour éviter de faire partir deux flèches de "ev1" et deux flèches de "ev4", nous utilisons un triangle duquel arrive des flèches de "ev4" et "ev1" et deux flèches partant vers "ev2" et "ev3".

G.3 Combinaison avec un réseau de Petri

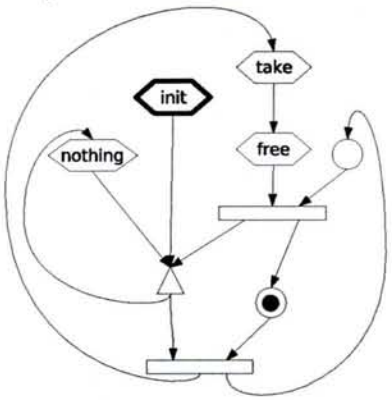
Comme nous considérons que nos graphes d'événements peuvent se récrire sous la forme de réseaux de Petri, nous nous autorisons à les combiner avec des réseaux de Petri. L'exemple montré sur la figure est tiré de l'étude de cas présentée dans le chapitre 7. Dans cet exemple, le passage à l'événement "take" entraîne la transition du jeton de la place se situant en bas, à celle se situant en haut. On voit que ce jeton ne repassera dans la case se

FIG. G.2 – Exemple de graphe d'événement utilisant un triangle



situant en bas seulement lorsque l'événement "free" se déclenchera pour passer à un autre événement ("take" ou "nothing"). Ceci est une manière de dire que lorsque l'événement "take" s'est produit, il ne peut pas se reproduire tant que l'événement "free" s'est déclenché.

FIG. G.3 – Exemple en combinaison avec un réseau de Petri



Nancy-Université

Université Henri Poincaré

Monsieur **ZIMMERMANN Yann**

DOCTORAT DE L'UNIVERSITE HENRI POINCARE, NANCY 1

en INFORMATIQUE

VU, APPROUVÉ ET PERMIS D'IMPRIMER N° 1241

Nancy, le 11 mai 2007

Le Président de l'Université

LE PRÉSIDENT

J.P. FINANCE

Université Henri Poincaré, Nancy I
24-30 rue Lionnois - B.P. 3069 - 54013 NANCY Cédex
Tél. : 03 83 68 20 00 - Fax : 03 83 68 21 00

Résumé

Les systèmes électroniques sont de plus en plus complexes et les exigences de fiabilité vis-à-vis de ces systèmes sont de plus en plus importantes. Le défi est de continuer à développer des systèmes de plus en plus complexes tout en assurant la correction de ceux-ci. Les méthodes de correction par le test sont aujourd'hui dépassées par la complexité des systèmes. Nous proposons dans cette thèse d'utiliser la preuve et le raffinement pour assurer la correction d'un système. La correction par la preuve à l'avantage de ne pas être limitée par la complexité du système. Nous proposons d'utiliser la méthode B et son concept de raffinement pour simplifier le processus de modélisation et de preuve. À chaque étape du raffinement, des obligations de preuve sont générées par les outils supports pour assurer que le modèle plus concret est bien correct vis-à-vis du modèle abstrait. Cette méthode assure que l'implantation finalement obtenue est correcte vis-à-vis du premier modèle abstrait qui constitue la spécification. Nous avons commencé par une étude de cas réaliste choisie par un industriel (Volvo) consistant à modéliser un contrôleur d'accès à un bus série. Cette étude de cas nous a permis de dégager des règles de modélisation pour le développement de circuits électroniques par la méthode B. Cela nous a conduit à définir le langage BHDL correspondant au niveau synthétisable de B et des traducteurs de BHDL vers VHDL et SystemC ont été développés. Une étude théorique du langage BHDL a été faite en définissant deux sémantiques de ce langage et en prouvant la correction de la traduction de BHDL vers VHDL. Des travaux ont également été faits pour traduire BHDL vers ACL2.

Mots-clés: méthode B, circuits électroniques, raffinement

Abstract

Electronic systems become more and more complex and reliability requirements are more and more important. The challenge is to continue to develop more and more complex systems while ensuring correction of systems. Test-based methods are now overtaken by complexity of systems. We suggest using proof and refinement to ensure correction of systems. Proof-based methods are not limited by the complexity of systems. We suggest using the B method and its concept of refinement to simplify the process of modelling and proving. At each refinement step, proof obligations are automatically generated by tools to ensure that the concrete model is correct with respect to the abstract model. This method ensures that the final implementation is correct with respect to the initial abstract model which is the specification. We started by a realistic case study chosen by an industrialist (Volvo) consisting in modelling an access controller for a serial bus. This case study leaded us to define some modelling rules to develop electronic circuits using the B method. We have defined the BHDL language which is the synthesisable level of B and we have implemented translators from BHDL towards VHDL and SystemC. A theoretic study of BHDL has been done defining two semantics for this language and proving the correction of the translation from BDHL to VHDL. Some work has also been done to translate BHDL to ACL2.

Keywords: B method, electronic circuits, refinement