Assistance au développement incrémental et prouvé de systèmes enfouis

Cyril Proch

Département de formation doctorale en informatique
UFR STMIA

École doctorale IAEM Lorraine

Assistance au développement incrémental et prouvé de systèmes enfouis

THÈSE

présentée et soutenue publiquement le 22 Mars 2006

pour l'obtention du

Doctorat de l'université Henri Poincaré – Nancy 1
(spécialité informatique)

par

Cyril Proch

Composition du jury

Président :	Monique Grandbastien	Professeur Université Henri Poincaré
Rapporteurs :	Florence Maraninchi	Professeur ENSIMAG-INPG
	Véronique Viguié Donzeau-Gouge	Professeur CNAM
Examinateurs :	Denis Abraham	Responsable de Projets (Docteur) TDF
	Dominique Cansell	Maître de Conférence (HDR) Université Paul Verlaine
	Dominique Méry	Professeur Université Henri Poincaré

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

Mis en page avec la classe thloria.

Remerciements

Bien qu'un seul auteur figure sur la couverture du document, la réalisation et la rédaction d'une thèse est le résultat d'un grand nombre de collaborations et de rencontres. Il est normal de réparer cette injustice en remerciant les personnes qui ont collaboré, directement ou indirectement, à l'aboutissement de ce travail.

En tout premier lieu, je tiens à remercier Dominique Méry et Dominique Cansell pour le soutien qu'ils m'ont apporté durant cette thèse. Ils m'ont accordé leur confiance en acceptant ma candidature pour la présente thèse alors que mon profil n'était pas celui recherché. Par leur rigueur scientifique, ils ont su me guider et m'épauler dans toutes les étapes de mon travail. Je remercie également Florence Maraninchi et Véronique Viguié Donzeau-Gouge de leur intérêt pour mon travail et d'avoir accepté d'être rapporteurs de cette thèse. Leurs différentes remarques ont grandement amélioré la présentation et le contenu de celle-ci. Je tiens à remercier également Monique Grandbastien d'avoir accepté de présider mon jury de thèse. Enfin, je tiens à remercier chaleureusement Denis Abraham pour sa confiance et son intérêt.

Je tiens à signaler ma gratitude au grand nombre de collègues et amis du laboratoire et de l'université qui m'ont fait confiance et qui ont rendu ma vie de thésard agréable. Tout d'abord, l'équipe MOSEL, dans son intégralité, qui a su être accueillante et a permis le bon déroulement de mon travail. Par ailleurs, je tiens à citer les différents thésards (à l'époque mais aujourd'hui docteurs) avec lesquels j'ai partagé de nombreuses pauses-café, toujours génératrices d'une grande quantité d'idées les plus variées : un grand merci à Guillaume Doyen, Mathieu Daquin et Yannick Darcy avec qui j'ai, en plus, partagé mon bureau.

Un thésard ne doit pas seulement être capable de réfléchir sur des questions scientifiques, il doit aussi suivre certaines procédures administratives plus ou moins obscures. Je tiens à exprimer ma reconnaissance à Nadine Beurné et Josianne Reffort pour l'efficacité de leur gestion et pour leur connaissance des méandres de l'administration. Par leur réactivité et leurs compétences, elles ont également contribué au bon déroulement de mon travail et de mes différents déplacements.

Il y a une vie, hors du laboratoire, et les personnes y participant ont eu un réel impact sur mon travail. Je tiens à tous les remercier ici et particulièrement Nelly, qui m'a supporté pendant ces trois années, ainsi que tous mes proch(e)s : mes parents, mes grand-parents, mes beaux-parents, mon frère, Florian, ainsi que Céline. Je tiens également à remercier mes amis (du monde de la musique ou d'ailleurs) qui m'ont aidé à me ressourcer et en particulier Alain, Cédric, Mario, Renaud, Jean-François et mes amis de Bluebird.

Sans l'équipe pédagogique de l'UHP, qui m'a donné l'envie de poursuivre dans l'informatique, je n'aurai certainement pas eu l'idée de faire une thèse, je les en remercie. Je pense tout naturellement à Martine Gautier (qui m'a ouvert les yeux sur les miracles de la compilation), Brigitte Jaray (qui m'a fait faire mes premières preuves), Brigitte Wrobel-Dautcourt (qui m'a appris à jongler avec les scripts) et François Schwaab (qui m'a tout montré d'UNIX et m'a convaincu que l'informatique était en perpétuelle régression depuis MULTICS).

La liste devenant trop longue, je tiens d'avance à présenter mes excuses aux personnes oubliées.

Résumé

Actuellement, les méthodes de développement rigoureuses électroniques ne se sont pas encore imposées et de nombreux développements reposent, essentiellement, sur le savoir-faire. La validation des systèmes (System-on-Chip ou SoC) obtenus se fait, souvent, par des séries de tests ne couvrant pas, en général, toutes les possibilités. L'utilisation de model checker se fait de plus en plus courante mais elle se heurte à la complexité des systèmes décrits et à l'explosion du nombre d'états. Dans ce contexte, nous avons élaboré une méthode de développement de SoC par la preuve en nous appuyant sur une étude de cas, celle du projet RNRT EQUAST, qui visait à produire un outil de mesure pour une norme (ETSI TR 101 290) dans le domaine de la télévision numérique terresmtre. La théorie du raffinement est centrale à nos travaux et l'utilisation intensive de celui-ci permet de conserver la traçabilité par rapport au cahier des charges tout en distribuant, tout au long du développement, la complexité du système (en introduisant plus de détails). L'utilisation du raffinement permet également de distribuer la difficulté de la preuve tout au long de la chaîne de raffinements. Les modèles construits permettent d'aider le concepteur dans ses choix architecturaux, les propriétés invariantes des modèles et le raffinement structurant les différentes tâches du système modélisé. Afin d'aider le concepteur, nous avons proposé un mécanisme de traduction de modèles permettant la production de code SystemC. Le code produit, qui conserve les propriétés d'ordonnancement des modèles sources, sert de support à des expérimentations électroniques. La correction de la traduction définie a été faite par l'intermédiaire de modèles B permettant de décrire la sémantique de simulation de SystemC. Un programme SystemC spécifique peut être représenté par un modèle B, instanciation des modèles génériques de simulation SystemC. Nous avons prouvé, toujours à l'aide du raffinement, que l'exécution des programmes résultats de la traduction conservait les propriétés des modèles abstraits initiaux.

Mots-clés: Méthode B événementielle, méthodes formelles, raffinement, Systèmes Enfouis

Abstract

Design process in electronic engineering is not yet formally defined and developments are based on case studies or empirical knowledge. Systems validation is generally made by tests but these are incomplete and all scenarios or use cases are not tested. The use of model checker becomes more general but is difficult because of systems complexity and state space explosion. We propose a design method built on proof and based on a case study, the RNRT project EQUAST. The aim of this project is to build a measurement tool based on normative requirements (ETSI TR 101 290) in the domain of digital video broadcasting television (DVB-T).

Our works are based on refinement and its use helps tracability by distributing complexity in different steps of refinement. The models help designer in his architectural choices by structuring tasks of modelled system thanks to invariant properties and refinement. To help designer, we propose a translation from formal models to SystemC code; the SystemC language is used by electronic designers to describe different parts of a system. Produced code conserves scheduling properties of source models and permits electronic tests to verify time or consumption constraints. Translation correction is proved by use of B event-based models which explain simulation semantics of SystemC. A specific SystemC program can be represented with a B event-based model. This model is an instanciation of generic models describing SystemC simulation. With the help of refinement, we have shown that simulation of translated programs conserves properties of abstract initial models.

Keywords: B event-based method, refinement, System-on-Chip

Table des matières

Table des figures xi

Introduction 1

1 Validation de programmes et outils . . . 1
2 De nouveaux types de systèmes . . . 2
2.1 Le flot *standard* de conception . . . 3
2.2 Approche modèle . . . 5
2.3 Remarques . . . 6
3 Le projet EQUAST . . . 7
3.1 La métrologie et les SoC . . . 7
3.2 Un processus de normalisation . . . 7
3.3 Objectifs du projet . . . 7
4 Objectifs et contributions . . . 7
4.1 Objectifs . . . 8
4.2 Contributions . . . 8
5 Plan du document . . . 10
6 Récapitulatif . . . 11

1 Etat de l'art 13

1.1 Outils de conception de circuits électroniques . . . 13
1.1.1 les langages HDL . . . 13
1.1.2 Le langage Verilog HDL . . . 15
1.1.3 Bilan . . . 15
1.2 Les langages synchrones . . . 16
1.2.1 Généralités sur les langages synchrones . . . 16
1.2.2 Lustre : langage synchrone flot de données . . . 17
1.2.3 Esterel . . . 18
1.2.4 Signal . . . 19
1.2.5 Vérification et langages synchrones . . . 19
1.3 Langages "système" et outils . . . 20
1.3.1 Extensions du langage C et des langages HDL . . . 20
1.3.2 Autres langages textuels . . . 21
1.3.3 Les langages graphiques de spécification . . . 21

1.3.4 Autres outils de vérification et de validation 23
1.4 Conception de SoC et raffinement formel 25
1.5 Bilan 26

2 La méthode B événementielle **29**
2.1 La théorie du raffinement 29
2.1.1 Raffinement et correction de programmes 30
2.2 Méthode B événementielle : modèle abstrait 33
2.2.1 Structure d'un modèle 33
2.2.2 Les substitutions 35
2.2.3 Forme des événements 36
2.2.4 Obligations de preuves 38
2.2.5 Exemple 39
2.3 Raffinement de modèles 41
2.3.1 Structure d'un raffinement 41
2.3.2 Raffinement et preuves en B 42
2.3.3 Exemple de raffinement 43
2.3.4 Détection d'erreurs de spécification par la preuve 48
2.4 Instanciation et paramétrisation 50
2.4.1 Projet : paramètres formels 50
2.4.2 Instanciation d'un projet 50
2.5 Dérivation d'algorithmes 51
2.5.1 Règles de recomposition 51
2.5.2 Exemple : raffinement vers une implantation 53
2.6 Conclusions 56

3 Etude de cas : premières approches **57**
3.1 Etude de cas : La DVB-T 57
3.1.1 L'information encapsulée 57
3.1.2 La gestion du système 58
3.1.3 La synchronisation 59
3.1.4 La synchronisation des horloges locales 59
3.1.5 T-STD : un modèle de décodeur 60
3.1.6 Les paramètres de QdS 61
3.1.7 Conclusions 62
3.2 Réalisations 62
3.3 L'acquisition de la synchronisation 63
3.3.1 Modèle abstrait : caractériser le flux 63
3.3.2 Premier raffinement 64
3.3.3 Raffinements de l'algorithme d'acquisition de la synchronisation 66
3.3.4 Synchronisation : conclusions 71
3.4 Le calcul du débit 72

3.4.1 Objectifs et principes de modélisation 72
3.4.2 L'algorithme de la mesure du débit 72
3.4.3 Architecture du circuit pour la mesure du débit 73
3.4.4 Développement incrémental de l'architecture pour la mesure du débit 74
3.4.5 Mesure de débit : conclusions 79
3.5 Modélisation d'un buffer de réception 80
3.5.1 T-STD : Abstraction 80
3.5.2 Raffinement : ajout des différents buffers. 82
3.5.3 T-STD : conclusions 86
3.6 Bilan général des premières approches 87
3.6.1 Récapitulatif 87
3.6.2 Critique des approches proposées 87

4 Etude de cas : globalisation **89**
4.1 Développement d'une hiérarchie incrémentale 89
4.1.1 Abstraction initiale : le coeur du système 90
4.1.2 Premier raffinement : **SYS1** 93
4.1.3 Second raffinement : **SYS2** 100
4.1.4 Autres raffinements 106
4.1.5 Raffinement **SYS6** : paramètres QdS 108
4.2 Eléments de conception architecturale 117
4.2.1 Ordonnancement de tâches 117
4.2.2 Aspect temporel 118
4.2.3 Génération de code 118
4.3 Bilan du projet EQUAST 119

5 SystemC : modélisation **121**
5.1 SystemC : un langage haut-niveau de description Hardware? 121
5.1.1 SystemC : une bibliothèque C++ 121
5.1.2 Syntaxe de base 122
5.1.3 Exécution et ordonnancement en deux phases 124
5.1.4 Premier modèle abstrait : manipulation de processus 124
5.2 SystemC : Ordonnancement à base d'événements 126
5.2.1 Ordonnanceur à base d'événements 126
5.2.2 Priorité de notification 127
5.2.3 Premier raffinement : introduction d'événements SystemC 127
5.3 SystemC : communications entre modules 133
5.3.1 Signaux : le média de communication 134
5.3.2 Sensibilité des processus 134
5.3.3 Remarques 136
5.3.4 Algorithme complet de l'ordonnanceur 136
5.3.5 Dernier modèle : prise en compte des signaux 136

5.4 Raisonner sur les programmes SystemC 139
5.4.1 Instanciation de modèles 139
5.4.2 Exemple jouet 140
5.5 Conclusions 142

6 Modèles généraux **145**
6.1 Hiérarchie de dépendance 145
6.1.1 Un graphe valué 145
6.1.2 Propriétés de la relation de dépendance 146
6.2 Modèles B de hiérarchie de dépendance 148
6.2.1 Modèle abstrait 149
6.2.2 Implantation à l'aide du raffinement 151
6.3 Hiérarchie : raffinement vers le matériel 153
6.3.1 Modèle abstrait et premier raffinement 154
6.3.2 Second raffinement : introduction de processus 156
6.3.3 Raffinements suivants : introduction de signaux 158
6.3.4 Hiérarchie : implantation proposée 162
6.4 Abstraction : conclusions 164

7 Conception de SoC à partir de modèles B **167**
7.1 Principes généraux 167
7.1.1 Classe de modèle B 167
7.1.2 Résultat de traduction 170
7.2 Algorithme de traduction 170
7.2.1 Construction de l'interface du module 171
7.2.2 Construction des processus 172
7.2.3 Remarques et limitations 174
7.3 Preuve de traduction 175
7.3.1 Construction d'un modèle d'exécution *ScheduledArchi* 175
7.3.2 Montrer le raffinement 179
7.4 Validité de la traduction 184

Conclusions **185**

Bibliographie **189**

Glossaire **195**

Annexes **197**

A Algorithme de synchronisation (pseudo-code C) **197**

B Exemples de code SystemC **199**

B.1 Un module SystemC type 199

B.2 Un programme SystemC complet 200

Table des figures

1 Flot de conception standard d'un SoC 3
2 Flot de conception "*modèle*" d'un SoC 5
3 Flot de conception initial du projet EQUAST 9
4 Flot de conception orienté raffinement 10
5 Flot de conception orienté raffinement et traduction 11

1.1 Différentes natures de descriptions VHDL 14
1.2 Implantation synchrone d'une machine d'états finis 16
1.3 Vérification du programme P par l'observeur B 19
1.4 Exemple de StateCharts décrivant un logiciel embarqué 22
1.5 Nature du circuit résultat de traduction 23
1.6 Cycle de vérification FormalCheck 25

2.1 Raffinement : conservation des comportements 32
2.2 Raffinement : transition muette 33
2.3 Principales notations ensemblistes du B événementiel 36
2.4 Exemple de projet B 50
2.5 Un autre projet B 51
2.6 Réutilisation de projet 52

3.1 Détails de l'en-tête d'un paquet TS 58
3.2 La structuration des données 59
3.3 Le décodeur T-STD 60
3.4 Comportement de la synchronisation 68
3.5 Comportements des variables *pointer* et *POINTER* 70
3.6 Schéma de conception basée sur la preuve 72
3.7 Illustration de la mesure du débit binaire spécifié par le standard TR 101 290 [55] 73
3.8 Architecture de la mesure du débit B_t 74
3.9 Première architecture 75
3.10 Une architecture plus précise (fonctionnelle) 76
3.11 Architecture du deuxième raffinement 77
3.12 Architecture du dernier raffinement 78
3.13 Raffinement du décodeur abstrait 83

4.1 Relation entre types de paquets et variables abstraites 92
4.2 Séquencement du temps 94
4.3 Hiérarchie incrémentale : système **SYS1** 97
4.4 Hiérarchie **SYS2** : un graphe acyclique 102
4.5 Hiérarchie issue du modèle **SYS3** 107
4.6 Hiérarchie issue du modèle **SYS4** 108
4.7 Hiérarchie issue du modèle **SYS5** 109
4.8 Hiérarchie finale 110

4.9 algorithme de calcul . . . 117
4.10 Durée d'activation des tâches . . . 118
4.11 D'un partitionnement à l'autre . . . 119
4.12 Partition de notre étude . . . 119
4.13 Vers une répartition sur FPGA . . . 120

5.1 Création d'un programme SystemC . . . 122
5.2 Exemple de code SystemC (incomplet) . . . 123
5.3 Automate et en-tête du modèle abstrait . . . 125
5.4 Différentes exécutions possibles d'un processus . . . 125
5.5 Code complet du module `my_module` . . . 128
5.6 Automate concret de l'ordonnanceur . . . 129
5.7 Construction de *newTimed* . . . 131
5.8 Modélisation de l'exécution . . . 134
5.9 Modification de valeur d'un signal . . . 135
5.10 Exemple de code SystemC avec signaux . . . 135
5.11 Génération d'événements B depuis du code SystemC . . . 141

6.1 Etapes de parcours d'un graphe . . . 146
6.2 Hiérarchie incrémentale obtenue par raffinement . . . 147
6.3 Illustration de l'acyclicité d'une relation *dag* . . . 149
6.4 Illustration du raffinement de hiérarchie . . . 165

7.1 Forme générale de la traduction . . . 171
7.2 Démarche générale de validation de traduction . . . 175
7.3 Analyse de flot d'un processus SystemC . . . 176
7.4 Reconstruction d'événement . . . 177

Introduction

L'informatique est devenue centrale à notre mode de vie. Alors que les premiers ordinateurs occupaient une pièce entière, ils sont aujourd'hui devenus un bien de consommation courant. Il est habituel de trouver dans un bureau ou chez un particulier un, voire des ordinateurs, remplissant un grand nombre de tâches, du ludique à la gestion, en passant par la communication et, bien entendu, par la programmation.

Les progrès technologiques et la miniaturisation ont permis l'utilisation de processeurs dans des systèmes embarqués. Ainsi, les téléphones portables et autres agendas électroniques contiennent un processeur et un noyau de système d'exploitation permettant à leurs utilisateurs d'envoyer des messages, de prendre des photos et accessoirement de téléphoner.

De la même façon, l'informatique et l'électronique embarquées sont de plus en plus présentes dans les transports. De la rame de métro parisien METEOR (métro sans chauffeur) aux plus récents *ordinateurs de bord* des automobiles, de plus en plus de programmes sont chargés de gérer différentes parties, de plus en plus critiques et complexes, des systèmes (voiture, avion, portable...) que nous utilisons.

Si la taille et les performances physiques (vitesse d'horloge, nombre de transistors) ont progressé de manière très importante, les problèmes de conception inhérents à la programmation informatique restent encore présents. L'utilisation de langages de programmation ne permet pas de garantir l'écriture de programmes fiables et corrects. Le passage d'un cahier des charges à une implantation réalisant correctement les fonctionnalités souhaitées est un exercice difficile [1].

1 Validation de programmes et outils

Le premier outil permettant de valider la conformité d'un programme vis-à-vis de ses spécifications ou de son cahier des charges est le test. Le principe très simple du test est de vérifier que le programme a les comportements attendus sur un certain nombre de scenarii de tests. L'inconvénient du test repose sur le problème de la couverture du jeu de test : il est impossible de savoir si les éventuelles erreurs du programme ou encore les non-conformités ont toutes été détectées. La construction d'un jeu de test exhaustif est généralement impossible et la vérification par test systématique est longue et fastidieuse (elle prend près de la moitié de la phase de développement d'un logiciel). De ce fait, la construction de jeux de tests efficaces et représentatifs est une opération délicate.

La prolongation de ces méthodes de tests est le *model checking* [46]. Cette technique formelle, basée sur une définition claire et rigoureuse, consiste à vérifier les propriétés d'un programme par la construction d'un automate spécifique représentant l'exécution du programme analysé. Le concepteur peut alors vérifier que l'automate respecte des propriétés de la spécification ou plutôt que des propriétés indésirables du programme ne sont acceptées par aucune trace de l'automate. Le model checking permet de vérifier des propriétés de sûreté, de vivacité ou encore d'équité [12] par l'utilisation de logiques temporelles. Cependant, cette technique est confrontée à un problème d'explosion combinatoire du nombre des états qui la rend peu applicable sur des développements de taille réelle. La communauté du model checking s'est alors tourné vers des techniques de model checking symbolique ou encore d'autres approches compositionnelles permettant de retarder ou limiter le phénomène d'explosion combinatoire. Le principal attrait de ces approches reste la vérification automatique de propriétés temporelles sur les systèmes décrits.

Egalement en réaction à ces problèmes de conception un certain nombre de méthodes formelles basées sur la preuve ont été proposées. Ces méthodes de preuves de programmes ont pour but la vérification

[1] Un grand nombre de développements industriels ne terminent pas à cause du non-respect du cahier des charges.

et la validation de la conformité des programmes vis-à-vis de leurs spécifications. Dans ce genre de méthode, le programme n'est généralement pas le point de départ mais plutôt le résultat d'un processus de réflexion et la production d'un programme exécutable est issue d'un raisonnement mathématique sur les spécifications du système. Le principal reproche fait aux méthodes dites formelles est souvent la complexité de celles-ci et le haut niveau d'expertise nécessaire. En effet, les méthodes basées sur la preuve utilisent un support mathématique et demandent la maîtrise d'un *langage de spécification* différent du langage de programmation cible. Elles ne sont pas aussi automatiques que l'approche model checking et demandent un travail de réflexion important.

D'autres méthodes plus ou moins formelles de conception cherchent à garantir, par le respect de règles, le résultat de la programmation et sa conformité avec le résultat attendu. Ainsi les méthodes comme UML (*Unified Modeling Language*) essaient d'imposer des règles de conception et de suivi de développement d'une application. La méthode UML définit un certain nombre de schémas permettant d'étudier un système sous différents *point de vues*. La modularité des programmes et la variété de domaines informatiques concernées (interface graphique, base de données, ...) motivent l'idée d'étudier un système sous ses différents aspects. Cependant, malgré son important succès, le manque d'une sémantique claire et précise empêche UML de s'imposer vraiment comme une méthode rigoureuse et efficace.

2 De nouveaux types de systèmes

Les systèmes enfouis sont des systèmes complexes mélangeant des aspects purement électroniques et des questions plus informatiques. L'informatique embarquée est confrontée à toutes sortes de problèmes : la modularité des programmes et les aspects hétérogènes des systèmes enfouis ajoutent une difficulté supplémentaire aux problèmes classiques de conception. Les programmes doivent réagir à des événements de l'environnement, communiquer ou déléguer des traitements avec d'autres parties (éventuellement électroniques) du système. Des contraintes fortes comme une consommation électrique réduite ou encore le respect d'impératifs temporels stricts sont des problématiques d'un nouveau genre. La conception de tels types de systèmes est donc extrêmement complexe du fait des nombreuses possibilités, informatiques ou électroniques, et des contraintes importantes. Ces difficultés de conception sont encore renforcées par de récents progrès technologiques permettant l'utilisation d'un mécanisme de *reconfiguration matérielle*. Ces nouveaux circuits sont un compromis intéressant entre performances optimales et temps de développement réduit. Les développements réalisés pour d'autres circuits peuvent être facilement réutilisables avec ce type de composants et une architecture spécifique implantée sur ce type de technologie est facilement extensible.

Ces circuits "programmables" peuvent être *reconfigurés* durant l'exécution d'une application, afin d'utiliser la même surface de silicium pour la réalisation de différents traitements. L'utilisation d'un tel mécanisme demande une compréhension fine de l'application réalisée et la connaissance de la durée d'activation des différentes tâches et processus la constituant. Le concepteur, s'il veut utiliser ce type de mécanisme, doit déterminer les tâches en exclusion mutuelle (jamais actives en même temps durant l'exécution de l'application) et réfléchir aux moyens éventuels d'implanter ces tâches sur la même surface de silicium. Une difficulté supplémentaire vient du fait que le mécanisme de reconfiguration dynamique est réalisé en un temps non nul et les circuits le permettant sont moins spécialisés donc moins rapides qu'un circuit spécialisé dans la réalisation d'une tâche unique. De ce fait, si ce type de technologie semble pouvoir limiter l'explosion en taille des réalisations électroniques elle n'est pas toujours applicable en raison de la complexité des traitements à réaliser ou encore à cause de la nature de l'application étudiée.

La conception d'un System-on-Chip (SoC) [2] est une tâche complexe, qui rencontre les problèmes classiques de conception de programmes, enrichis de problèmes physiques liés aux technologies matérielles employées. Le concepteur d'un SoC doit, en fonction d'un grand nombre de paramètres (temps de traitements, surface silicium, consommation, réutilisabilité, coût,...), réaliser un compromis et *partitionner* l'application. La *partition* d'une application consiste à distribuer les différentes tâches du systèmes aux différentes ressources et à déterminer les parties matérielles (utilisant des composants matériels) et logiciels (utilisant un processeur) du système. La difficulté de conception est encore renforcée par l'utilisation

[2] Un glossaire situé à la page 195 référence les différents sigles et abréviations techniques utilisées dans ce document.

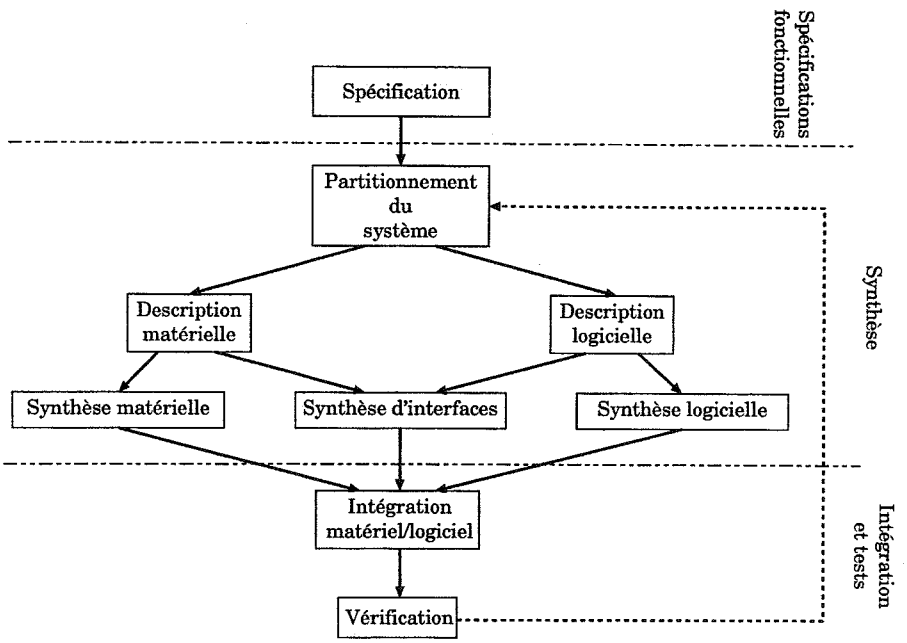


FIG. 1 – Flot de conception standard d'un SoC

de la reconfiguration dynamique matérielle. De ce fait, la conception d'un tel système est une tâche extrêmement délicate et complexe. Nous allons présenter rapidement deux principaux flots de conception de SoC, afin d'illustrer les problèmes inhérents à la construction de tels systèmes.

2.1 Le flot *standard* de conception

Cette partie décrit le flot de conception *standard* d'un SoC qui synthétise un système à partir de sa spécification haut niveau et qui requiert l'utilisation d'outils de co-design, de compilation, de synthèse architecturale et de langages de description adaptés. L'architecture du système est générée à partir de la description fonctionnelle initiale de l'application qui est "décomposée" en blocs fonctionnels. La figure 1 résume les différentes étapes de la conception d'un SoC.

Spécifications du système

Le processus de conception débute par la spécification du système. Cette spécification, généralement en langue naturelle, est issue du cahier des charges. La spécification du système incorpore des contraintes de temps, d'aire de silicium ou encore de consommation pour la future application. Quelques tentatives de formalisation basées sur les machines d'états finis [45, 101] (voir figure 2), flot de conception *modèle*) sont apparues mais dans cette première méthodologie de conception, les concepteurs se contentent généralement d'une description informelle de l'application.

Partitionnement du système

Le concepteur du système utilise la spécification et son expérience pour découper le système en blocs fonctionnels. Cette première étape achevée, le concepteur détermine les parties matérielles et logicielles du système. Ces deux parties sont ré-écrites séparément (par des équipes séparées) dans un langage "haut niveau" pour la partie logicielle (souvent C) et dans un langage permettant la synthèse de circuit pour la partie matérielle (souvent VHDL). Les nouvelles descriptions de l'application ne sont pas reliées avec la description fonctionnelle de celle-ci et rien ne garantit le respect du cahier des charges initial. Il faut donc à nouveau tester les deux parties du système, pour vérifier le respect des fonctionnalités.

Synthèse

La synthèse est le procédé qui permet de passer d'une description "langage" à un modèle physique. Cette traduction est généralement réalisée à l'aide de différents outils de synthèse. La synthèse de chaque bloc par son outil dédié n'est pas suffisante et il convient également de synthétiser les interfaces et les moyens de communication entre blocs.

Pour les parties logicielles, cette synthèse consiste à "raffiner" la description de haut niveau d'une tâche en une description de bas niveau exécutable par un processeur. Cette notion de raffinement n'est pas formellement définie et consiste essentiellement en la réécriture des descriptions initiales en des descriptions plus détaillées. Le code binaire est généré par un compilateur dédié au processeur cible. Lorsque plusieurs tâches sont assignées à une même ressource matérielle, un système d'exploitation est alors nécessaire.

La synthèse matérielle transforme un modèle décrit à l'aide d'un langage (souvent VHDL ou Verilog) en une description bas niveau. Une première étape est réalisée à l'aide d'outils de synthèse qui, à partir d'une description algorithmique, produisent une description au niveau transfert de registres (*Register Transfer Level*, RTL). Cette spécification RTL est ensuite synthétisée à l'aide d'outil de synthèse logique (ou synthèse RTL) pour obtenir un ensemble d'interconnexions de transistors (*netlist*) [110].

Intégration et co-simulation

La phase de synthèse terminée, les différents blocs sont intégrés sur une même plate-forme et débute alors la difficile phase de co-simulation. Les différents blocs étant décrits dans différents langages et représentations intermédiaires, les concepteurs de l'application interprètent les résultats de chaque partie séparément et "rejouent" le comportement d'une partie simulée, afin de voir les réactions de l'autre. Il existe cependant quelques environnements de co-simulation intégrées comme CoCentric [108] de Synopsis permettant la co-simulation de modules VHDL et C.

Validation de conception

Les résultats de la co-simulation sont finalement comparés aux spécifications fonctionnelles et aux contraintes générales de coûts. Les performances du système sont également vérifiées à ce moment. Si le système ne satisfait pas les spécifications tant du point de vue des contraintes temporelles que du point de vue fonctionnel, l'intégralité du processus est repris depuis le partitionnement de la spécification fonctionnelle, les résultats de la co-simulation étant pris en compte lors de la nouvelle conception.

Limitations

Le flot de conception standard présente un certain nombre de limitations. Tout d'abord, l'absence de vérification globale hors de la co-simulation pose un important problème. En effet, un certain nombre de vérifications internes à chaque bloc sont réalisées mais la vérification globale du bon fonctionnement de l'intégralité du système a lieu très tardivement dans le cycle de développement et les erreurs sont lourdes de conséquence au regard du temps de conception déjà écoulé. De plus, développé et testé du logiciel embarqué basé sur une description RTL est extrêmement long et coûteux.

Par ailleurs, la réunion des parties logicielle et matérielle provoque souvent des ré-implantations pour cause de non-conformité entre les interfaces ou entre les délais de communication. L'utilisation d'IP (*Intellectual Property*, approche composante) ou le développement de modules séparément est une source importante de problèmes.

Enfin, ce flot de conception implique des optimisations locales à chaque bloc (ou une refonte complète de tout le système) et ne permet ni un debogage ni une évaluation facile des performances *a priori*. En effet, le mécanisme de co-simulation gène la localisation rapide des erreurs et ne permet pas de conclure facilement sur les problèmes temps réel ou encore sur la consommation d'énergie.

Ce flot de conception ayant montré assez rapidement ses limites, de nouvelles méthodes de conception de SoC ont été proposées et un flot de conception orienté "*modèle*" est apparu.

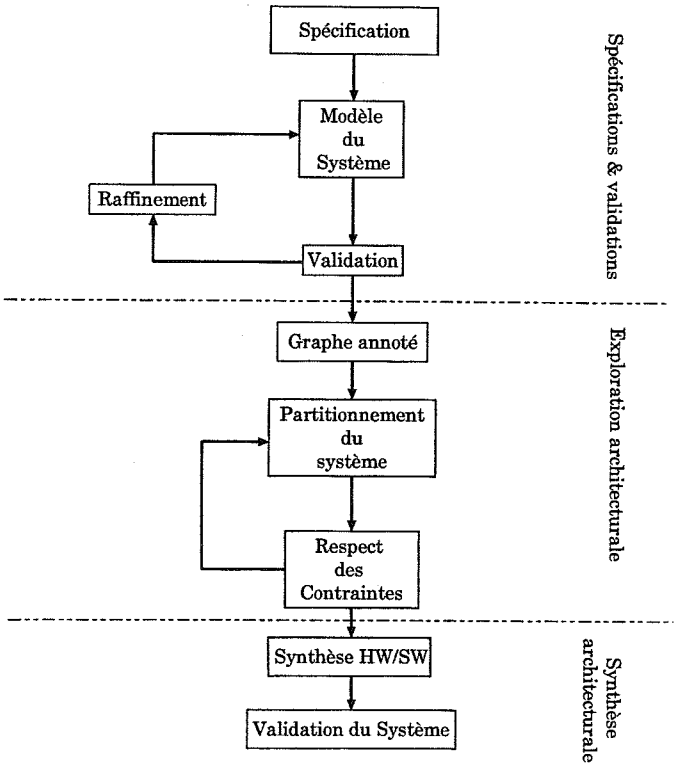


FIG. 2 – Flot de conception "*modèle*" d'un SoC

2.2 Approche modèle

Le flot de conception orienté modèle a pour principal but l'unification des différents langages et méthodes, afin de manipuler un modèle global du système. Dans cette approche, des vérifications (éventuellement formelles) ont lieu lors de la phase de spécification, au début du flot de conception, et permettent une meilleure validation de l'architecture obtenue tout en ayant un coût de conception raisonnable. La figure 2 présente les différentes étapes du flot de conception orienté modèle simplifié.

Spécification du système

Le flot de conception débute généralement par une *spécification fonctionnelle* du système. Cette spécification (exécutable généralement) décrit le comportement de l'application et est accompagnée d'un ensemble de contraintes de conception telles que le coût en surface, la consommation électrique, le débit, les interfaces etc... Ces exigences seront utilisées par la suite dans les phases avales de la conception. La spécification fonctionnelle est très souvent réalisée à l'aide d'un langage de programmation séquentiel et ne fait aucune référence à la structure du matériel ou du logiciel qui permettra l'implantation finale du système.

Cette première spécification est ensuite transformée (raffinée) en un ensemble de blocs fonctionnels communiquant qui exhibent le parallélisme de l'application : *la spécification comportementale*. L'utilisation de la spécification comportementale permet la décomposition d'un système complexe en sous-systèmes.

Le système est décrit par un ensemble de processus (ou blocs fonctionnels) concurrents qui échangent des données au travers de canaux virtuels. Ce nouveau niveau d'abstraction, TLM [34] (*Transaction Level Modelling*), permet de valider et vérifier facilement, par simulation, la cohérence et la communication de ce type de description. TLM est une approche *composant* dans laquelle les blocs matériels sont des modules. Ces modules communiquent par l'intermédiaire de *transactions* (échange de données ou événement).

Ce niveau d'abstraction permet de ne pas considérer les cycles d'horloge du modèle RTL et accélère sensiblement le temps de simulation.

La simulation permet également la validation du modèle avant l'inclusion de nouveaux détails d'implantation. A ce niveau abstrait de description, les vérifications portent essentiellement sur le respect du cahier des charges et la correction interne du modèle. A un niveau d'abstraction donné, une fois l'ensemble des vérifications terminées, le concepteur raffine tout ou partie de son modèle.

Cette opération est répétée plusieurs fois jusqu'à obtenir une description suffisante pour permettre l'étude du partitionnement du système.

Exploration architecturale

Nous présentons dans ce paragraphe les principales étapes de l'exploration architecturale : le choix de l'architecture, le partitionnement matériel/logiciel et l'estimation de performances. Ces opérations sont interdépendantes et l'ordre dans lequel elles sont réalisées peut varier selon le système étudié.

L'exploration architecturale affecte les blocs fonctionnels de la spécification comportementale aux ressources matérielles sélectionnées et ordonne l'exécution des tâches. L'affectation de tâches à des ressources matérielles permet d'associer des temps d'exécution aux blocs composant la description comportementale. Durant cette étape, les communications entre les blocs fonctionnels peuvent être réalisées au travers de canaux plus ou moins abstraits. La communication peut ainsi être instantanée, dans ce cas aucun délai n'est associé aux canaux. Elle peut aussi, en fonction de son niveau d'abstraction, contenir des informations telles que la latence, le débit, la largeur des bus, etc...

A l'aide de ces informations, le concepteur est alors en mesure de vérifier le respect des contraintes de coût et de temps de calcul avant la réalisation définitive de l'application.

Synthèse du système

La dernière étape du flot de conception est la synthèse du système. Cette étape utilise les mêmes outils de synthèse que ceux décrits précédemment. L'avantage est, ici, que les médias de communications ont déjà été pris en compte dans une des descriptions abstraites du système. De ce fait, un choix de type de média de communication (bus, connexion point à point, ...) a déjà été fait.

2.3 Remarques

Les deux flots de conception présentés illustrent bien les difficultés de conception d'un SoC. A la frontière de plusieurs domaines, la conception de ce genre de système est délicate et mélange les problèmes spécifiques de chaque domaine. La motivation de ces flots de conception, et en particulier du flot orienté modèle, est de séparer les différents problèmes afin de les étudier et les résoudre plus efficacement. Les techniques standards de validation du logiciel (tests, model-checking, ...) sont applicables dans le cadre de la conception de systèmes enfouis mais la diversité des problèmes à traiter demande une adaptation de ces techniques ou l'utilisation d'autres outils (simulation).

Le flot de conception basé sur les modèles améliore en de nombreux points le flot de conception standard. L'introduction de descriptions abstraites de l'ensemble de l'application permet d'avoir une vue globale et cohérente du système. La validation du comportement (global mais aussi local) du système au niveau abstrait permet l'élimination d'un certain nombre d'erreurs et sépare les différents problèmes. A partir de la phase d'exploration architecturale, le concepteur n'a plus ou peu à s'inquiéter du respect fonctionnel du cahier des charges, et peut se focaliser sur les aspects techniques de la réalisation.

D'un point de vue qualité de conception, l'approche modèle permet des optimisations locales mais aussi globales du système en proposant des modèles de plus en plus concrets. Dans le même ordre d'idée, les communications et les interfaces ayant été décrites de manière abstraite, l'intégration des différents blocs logiciels et matériels est prise en compte dès le début de la conception et s'en trouve simplifiée.

Dans le cadre de la conception orientée modèle, le terme *raffinement* est de plus en plus employé et l'apparition de niveau de description plus abstrait comme TLM indique un effort d'abstraction. Il est cependant indispensable d'outiller la conception de SoC en proposant des solutions de vérification et de validation adaptées aux différents problèmes de chaque étape.

3 Le projet EQUAST

Nos travaux se sont déroulés au sein du RNRT EQUAST, un projet ministériel, qui visait à produire un outil de mesure dans le cadre de la télévision numérique terrestre (TNT). Cet outil, implanté sous la forme d'un SoC, permet de juger de la qualité de transmission d'un réseau, afin de garantir à l'utilisateur une qualité de diffusion maximale. Le projet est aujourd'hui terminé et deux prototypes, actuellement en phase de pré-commercialisation, en sont issus.

3.1 La métrologie et les SoC

La métrologie est la science de la mesure. La mesure d'une notion qualitative est toujours subjective et se réalise généralement par le contrôle d'un certain nombre de *paramètres* considérés comme représentatifs. Les SoC sont une plate-forme privilégiée pour l'implantation d'outils de mesure. En effet, la partie matérielle des SoC permet de réaliser des traitements lourds alors que la partie logicielle permet l'exploitation et la mise en page des résultats. De plus en plus couramment, des solutions de métrologie sont donc développées sur des SoC permettant la réalisation effective de la mesure mais aussi sa consultation par l'intermédiaire d'interfaces html ou autres.

3.2 Un processus de normalisation

Le télévision numérique terrestre DVB-T (Digital Video Broadcasting Television) a introduit des normes, afin d'évaluer la qualité de transmission des programmes. En DVB-T, tous les programmes sont émis sur le même média, constituant un flux. Le flux est constitué de différents paquets qui contiennent les informations nécessaires à la reconstruction (données, gestion) des programmes. Une première norme TR 101 154 [54] concerne l'évaluation du transit du signal à l'aide d'un certain nombre de paramètres. Ce premier jeu de paramètres d'évaluation se bornait à vérifier la cohérence des informations, le bon respect des structures et la bonne construction de l'en-tête des paquets. L'insuffisance de cette première norme est apparue assez vite et une nouvelle norme a été proposée. Cette nouvelle norme TR 101 290 [55], construite sur la norme précédente, ajoute de nouveaux paramètres, ainsi qu'un certain nombre de paramètres dits de *Qualité de Service* (QdS). Ces paramètres QdS sont des compositions de paramètres déjà présents considérés comme importants.

L'un des principaux inconvénients de ces normes est le haut niveau d'expertise nécessaire au diagnostique des erreurs. En effet, lorsque les paramètres ont des valeurs incorrectes, des alarmes sont déclenchées afin de signaler le dysfonctionnement. Cependant, dans ce domaine, il est rare qu'une unique alarme se déclenche et c'est très souvent une cascade d'alarmes qui apparaissent lors des dysfonctionnement. Déterminer la cause du problème demande alors une bonne connaissance des normes mais aussi des mécanismes de diffusion. Le but des paramètres QdS (dits de *synthèse*) est justement de simplifier l'analyse en proposant des alarmes de "plus haut niveau".

3.3 Objectifs du projet

Le projet RNRT EQUAST, auquel nous avons participé, regroupe 5 partenaires : TDF C2R, Thalès B&M , SODIELEC, LIEN (laboratoire d'électronique de l'UHP) et le LORIA. Son but est la réalisation d'un outil de mesure fondé sur l'intégralité de la nouvelle norme [55]. Du fait du nombre de traitements à réaliser, la conception puis la réalisation d'un tel outil sous la forme d'un système enfoui (System on Chip ou SoC) nécessite une analyse précise des besoins. En effet, malgré le lancement officiel de la TNT, aucun équipement du marché n'implante l'ensemble des paramètres d'évaluation de la qualité. Ceci est aussi bien dû aux coûts de réalisation importants qu'aux contraintes "temps réel" fortes issues de l'analyse "en direct" du flux.

4 Objectifs et contributions

Avec l'apparition des SoCs, le monde de l'électronique a dû s'adapter rapidement à une complexité de conception nouvelle. Le mélange, dans une même plate-forme, du logiciel et du matériel pose de nombreux

problèmes de conception mais aussi de validation et de vérification. Une importante réflexion sur les méthodes de conception a été menée permettant d'apporter des éléments de réponse et une méthodologie générale pour ce type de système.

4.1 Objectifs

Si la production automatique, aussi bien matérielle que logicielle, du code d'implantation semble relativement au point à l'aide de nombreux outils (compilateurs, synthétiseurs), la première partie du flot de conception, la spécification, n'est pas aussi aboutie. Nous avons présenté rapidement un certain nombre de méthodes et de langages ayant pour but la description du SoC en des modèles plus ou moins abstraits mais le nombre important de travaux, ceux présentés dans [64] et d'autres, montre qu'aucune méthode n'est encore vraiment satisfaisante.

En particulier, nous pensons que les méthodes de validation et la vérification du flot de conception proposées sont insuffisantes et ne permettent pas de tirer profit de celui-ci. Par ailleurs, l'absence de sémantique mathématiquement définie provoque des "failles" dans la chaîne de production de code. Ainsi il peut y avoir des différences de comportement entre les simulations de la description d'un circuit et son comportement réel après synthèse. L'utilisation du model checking [46] permet de vérifier un certain nombre de propriétés mais la nature même des systèmes manipulés provoquent rapidement un nombre d'états important qui impose la vérification automatique de propriétés assez simples.

Nous proposons donc un ensemble de suggestions afin d'améliorer la conception de SoC et surtout leur sûreté de fonctionnement en travaillant sur la spécification de ceux-ci à l'aide de la méthode B événementielle. Nous pensons que l'utilisation d'un raffinement formellement défini dès le début du flot de conception permettrait un gain de qualité et de sûreté de fonctionnement important dans la réalisation de SoC. En effet, une notion de raffinement plus ou moins informelle est déjà utilisée dès la spécification du système dans le flot de conception *modèle* de SoC et l'utilisation de niveaux de description plus abstraits (TLM par exemple) montre la nécessité et la réalité de ce type d'approche. Notre objectif principal est de minimiser les allers et retours entre les différents niveaux de description et de minimiser les phases de simulation de "haut-niveau".

La possibilité de décrire des modèles B événementiels très abstraits et de les raffiner en des descriptions implantables nous semble être un outil de codesign intéressant permettant de lier les différents niveaux de spécification (spécifications initiales, niveaux TLM et RTL, ...). Nous pensons qu'un gain de méthodologie important peut être obtenu dans les phases précédent le partitionnement matériel et logiciel du système. Par ailleurs, une utilisation conjointe du raffinement et du savoir-faire des électroniciens doit permettre la réalisation d'un partitionnement judicieux respectant l'ensemble des contraintes fixées.

4.2 Contributions

Dans un premier temps, nous avons étudié les méthodes de conception électronique et nous avons testé quelques possibilités de validation architecturale dans le cadre de l'étude de cas [41]. Plusieurs schémas de conception ont été essayés afin de permettre aux différents partenaires de comprendre les problématiques distinctes et les apports éventuels de chacun.

Par la suite, nous avons repris le schéma de conception orientée *modèle* et utilisé la méthode B événementielle pour concevoir une architecture SoC. Le flot de conception utilisé implique un grand nombre d'allers-et-retours entre les différents niveaux de descriptions du système ce qui est problématique pour la cohérence et la durabilité des développements. Nous avons construit un flot de conception basé sur la preuve et le raffinement limitant les corrections et les allers et retours entre les différents niveaux de description.

Dans le cadre du projet EQUAST, les électroniciens utilisaient le flot de conception, plutôt orienté modèle, présenté à la figure 3. A partir de spécifications relativement informelles, ils construisaient des programmes SystemC permettant surtout des vérifications fonctionnelles. Cette première étape réalisée, les programmes SystemC étaient réécrits dans un langage de description matérielle (VHDL) afin de permettre une synthèse matérielle. Ce premier flot de conception posait de gros problèmes de vérifications. En effet, des erreurs détectées très tard dans le cycle de développement nécessitaient des corrections importantes dans les premières phases du développement (modélisation SystemC voire spécifications) Ces

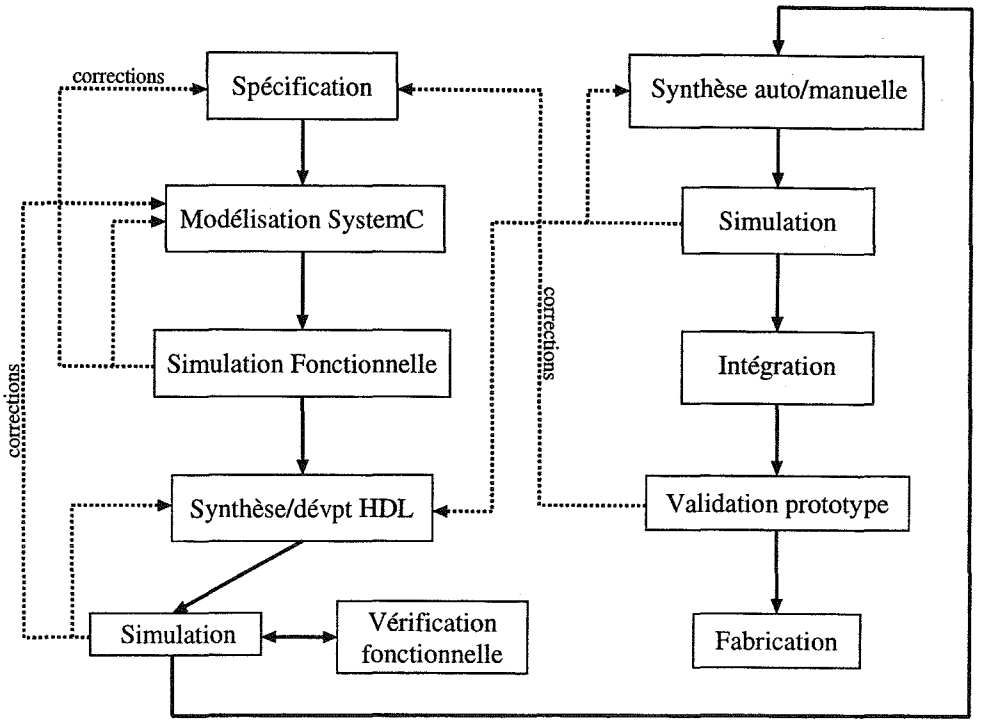


FIG. 3 – Flot de conception initial du projet EQUAST

corrections sont représentées par les flèches pointillées de la figure 3). Au regard du temps de conception et de développement, ce genre d'erreurs peut s'avérer dramatique.

Suite à notre collaboration étroite avec les concepteurs électroniques nous avons proposé un flot de conception différent. Celui-ci, présenté figure 4, utilise la modélisation B et la définition formelle du raffinement B afin d'éviter les corrections a posteriori durant la conception. Ce flot de conception correspond aux premiers travaux que nous avons réalisé dans le cadre du projet EQUAST. Les électroniciens et nous avons modélisé, en parallèle, un système à partir d'une spécification commune. Afin de comprendre la nature de la relation entre les différentes descriptions électroniques nous nous sommes appuyés sur les développements réalisés par les électroniciens pour raffiner formellement nos modèles. Les preuves nous permettent de détecter d'éventuelles erreurs et de les corriger immédiatement. Les concepteurs électroniques se sont également appuyés sur nos modèles qui ont permis l'explication de certains traitements particulièrement complexes. Cette approche permet d'éliminer des problèmes de cohérence et de garantir un comportement similaire entre spécification et description abstraite du système. Une fois l'intégralité des preuves réalisées, les modèles abstraits (SystemC et B) ne sont plus remis en question et servent de support à une description de plus bas niveau. De ce fait, les erreurs détectées au niveau RTL (une fois la traduction en un langage HDL réalisée) n'ont pas d'impact sur les modèles SystemC ni même sur les spécifications du système. En effet, les preuves formelles garantissent la cohérence de ces modèles. L'utilisation d'IP dans ce cadre peut être envisagée : la modélisation du système étant initialement abstraite et s'appuyant sur les modèles des électroniciens, une sous-partie du modèle formel peut parfaitement représenté un composant sans détailler précisément son fonctionnement interne mais seulement ses interactions avec le reste du système.

La collaboration avec les ingénieurs électroniciens et la compréhension de leurs méthodes de conception nous a finalement amené à définir une traduction de nos modèles B vers SystemC permettant une meilleure exploitation de ceux-ci [35]. Le choix du langage SystemC comme langage cible a été naturel car il permet une description plus ou mois abstraite d'un système et était utilisé dans le flot de conception initial comme un "langage de modélisation". La traduction définie produit un code SystemC proche du niveau RTL permettant une synthèse matérielle facile et non des programmes SystemC destinés à des

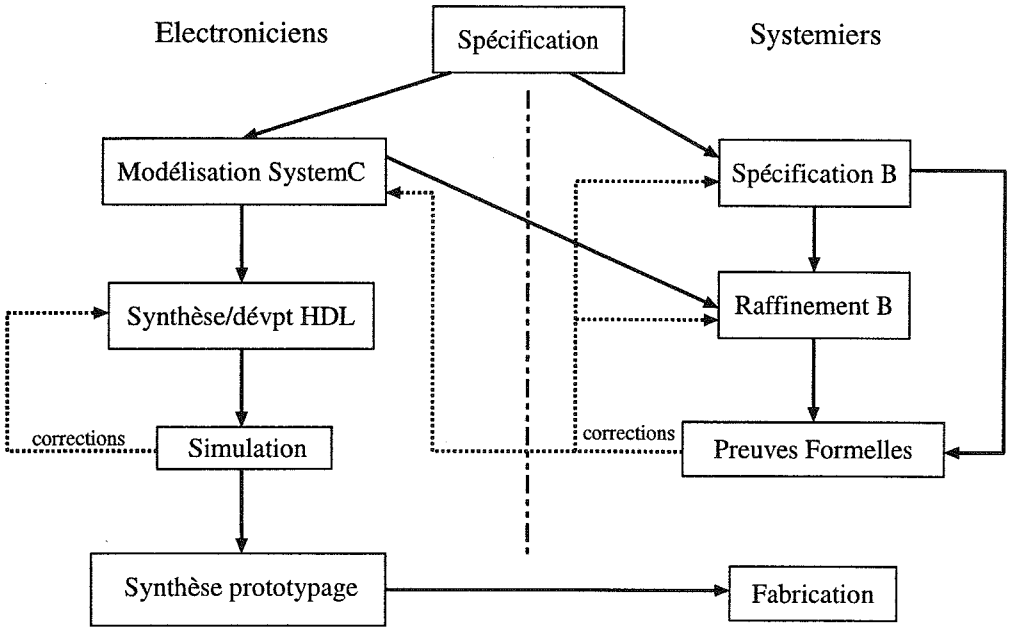


FIG. 4 – Flot de conception orienté raffinement

vérifications fonctionnelles (celles-ci ont été réalisés dans les modèles B abstraits). La correction de cette traduction a été elle-même réalisée à l'aide de modèles B événementiels explicitant le mécanisme de simulation de SystemC [38, 39]. La traduction doit permettre la conservation des preuves réalisées dans le cadre de la modélisation et, en particulier, les preuves de dépendances qui permettent de garantir un ordonnancement correct des tâches. En effet, la définition d'une fonction de traduction de modèles B vers du code SystemC simplifie le flot de conception et minimise encore les allers-retours entre les différents niveaux de description du système. Les corrections ne s'appliquent alors qu'au niveau de description courant et les modèles plus abstraits ne sont pas remis en cause. Un tel flot de conception est présenté à la figure 5.

Ainsi, à partir des documents de normalisation et en relation avec nos partenaires, nous avons extrait et conçu des modèles B événementiels intégrant progressivement, par la relation de raffinement, l'intégralité des paramètres à évaluer [40, 92]. Cette étude a, par ailleurs, permis de justifier l'introduction des nouveaux paramètres QdS déterminés empiriquement [81, 1]. Nous avons réussi, à l'aide de nos modèles, à hiérarchiser les paramètres et les alarmes associées. De ce fait, nous sommes en mesure d'expliquer les cascades d'alarmes déclenchées lors de l'invalidité des paramètres.

5 Plan du document

La suite de ce document est structurée en plusieurs chapitres traitant de points particuliers de nos recherches. Dans le chapitre 1 nous présentons un certain nombre de travaux déjà réalisés sur la spécification de système et en particulier dans les domaines des systèmes enfouis ou SoC. Nous présenterons les langages synchrones comme Lustre, Esterel ou encore Signal qui sont des langages dont la sémantique est clairement définie et qui permettent une description précise de systèmes ainsi que d'intéressantes possibilités de vérification (notamment par *model checking*).

Le chapitre 2 est consacré à la méthode B événementielle de Jean-Raymond Abrial. La méthode B est au coeur de nos travaux de modélisation et nous présenterons donc en détail les principes, en particulier le raffinement, et la syntaxe de celle-ci.

Le chapitre 3 présente les grandes lignes de la norme TR 101 290 [55] et les premiers développements que nous avons réalisés dans le cadre du projet EQUAST. Il illustre notre compréhension progressive des

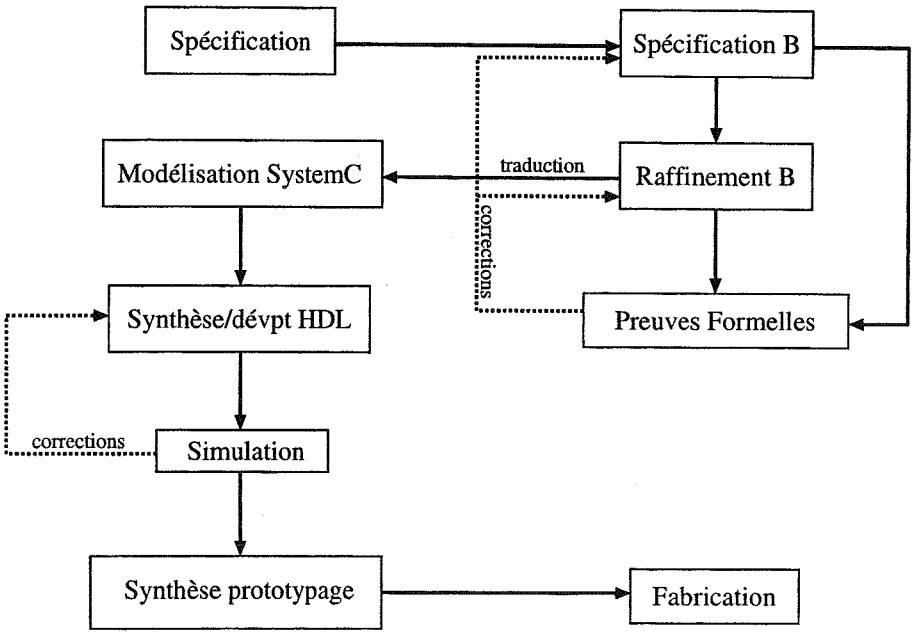


FIG. 5 – Flot de conception orienté raffinement et traduction

problèmes de conception d'un système enfoui en présentant des études de cas spécifiques concernant des points particuliers de l'application.

Le chapitre 4 décrit le travail de modélisation réalisé dans le cadre de l'étude de cas ainsi que les réflexions menées pour tirer profit de l'utilisation d'une méthode formelle. Les développements présentés dans ce chapitre ont pour but la modélisation de l'intégralité d'un système et l'étude de celui-ci par l'intermédiaire de modèles adaptés.

Dans le chapitre 5 nous traiterons du langage SystemC permettant la description de systèmes enfouis. La nature de ce langage et ces mécanismes de simulation en font un outil très utilisé dans la conception électronique. L'utilisation de SystemC par les électroniciens dans le cadre du projet EQUAST nous a naturellement conduit à nous servir de ce langage et à le choisir comme langage cible de traduction. Nous avons donc étudié ce langage à l'aide de la méthode B événementielle et avons proposé des modèles permettant de comprendre et de raisonner sur la simulation de programmes SystemC. Cette étude nous a permis de comprendre le fonctionnement de la simulation SystemC et de définir plus clairement le comportement des programmes SystemC.

Le chapitre 6 est une généralisation de la démarche appliquée dans le cadre du projet EQUAST à d'autres systèmes ou applications de même nature. Nous présentons dans ce chapitre un ensemble de modèles destinés à illustrer et à caractériser la classe des applications pour lesquelles nos travaux ont un intérêt.

Le chapitre 7 présente la fonction de traduction que nous avons définie permettant de produire un module SystemC à partir d'un modèle B événementiel spécifique. La preuve de la correction de cette traduction est également présentée dans ce chapitre. Cette preuve a été réalisée à l'aide de modèles B permettant de montrer la conservation des comportements.

Enfin, nous terminerons ce document par un chapitre de conclusions résumant les différents travaux réalisés et proposant quelques prolongations et travaux futurs dans le domaine.

6 Récapitulatif

Les travaux réalisés dans le cadre de cette thèse sont basés sur une utilisation intensive et systématique de la méthode B et du raffinement. Cette approche permet une certaine cohérence lors de la réflexion

et l'utilisation d'un même formalisme d'un bout à l'autre des raisonnements. Les différents problèmes étudiés sont tous abordés à l'aide de ces puissants outils :

- modélisation B et raffinement sont utilisés pour comprendre et vérifier les développements électroniques. Le raffinement garantit la correction des systèmes décrits et permet de réduire les phases de simulation.
- la modélisation B et le raffinement ont également un grand rôle pédagogique car ils permettent d'expliquer des systèmes parfois complexes. L'utilisation d'un formalisme mathématique permet de décrire avec un vocabulaire précis et commun les systèmes et leurs fonctionnements. Dans le cadre de l'étude de cas, cette approche a provoqué le réel démarrage de débats de fond sur le système. La production de "hiérarchies" de tâches précisant le système a pour origine les différents modèles successifs construits lors le la modélisation.
- nous utilisons également le raffinement, de manière original, pour valider une fonction de traduction de modèles B vers des programmes SystemC, et montrer que celle-ci conserve bien certaines propriétés établies des modèles.

Chapitre 1

Etat de l'art

La conception de systèmes enfouis est un exercice difficile et différentes approches essayent d'aider le concepteur dans sa réflexion. Un nombre important de celles-ci sont des généralisations d'outils ou de langages électroniques et manipulent les modèles classiques de la discipline (FSM (*Finite State Machine*), StateCharts). Inversement, d'autres approches proviennent du monde informatique et essayent d'adapter les modèles courants à la problématique des systèmes enfouis. La vérification et la validation des descriptions sont au coeur du problème et de nombreux outils (simulation, test, model checking ou encore preuve) sont utilisés.

1.1 Outils de conception de circuits électroniques

La construction de circuits électroniques a pendant longtemps reposé sur le savoir-faire et les connaissances empiriques. Le concepteur du circuit décrivait celui-ci à l'aide de notations graphiques (toujours utilisées) appelées *schémas structurels* [107]. Ces premiers modèles étaient extrêmement proches de l'implantation et manipulaient les portes logiques voire les condensateurs, composants principaux des circuits. Avec l'accroissement de la complexité et de la taille des circuits, les schémas structurels comme langage de conception ont été abandonnés ; ils ne permettaient plus la description complète de ces circuits.

Du fait de la complexité des applications et des contraintes technologiques, l'impression d'un circuit électronique est devenue extrêmement coûteuse et les conséquences financières des erreurs sont devenues dramatiques. Les concepteurs de circuits ont donc cherché des outils permettant non seulement de décrire ces circuits mais aussi de vérifier les descriptions réalisées avant impression.

Des langages textuels de description, accompagnés d'outils d'aide à la conception, ont alors fait leur apparition permettant de décrire de plus larges applications et de valider, par simulation, les circuits avant leur réalisation.

1.1.1 les langages HDL

La famille des langages HDL (*Hardware Description Language*) regroupe différents langages permettant la description et la synthèse des réseaux de portes logiques constituant un circuit. Aujourd'hui, les langages HDL sont considérés, dans la communauté électronique, de la même façon que les langages assembleurs dans la communauté informatique.

L'électronicien a toujours utilisé des schémas de description pour représenter des structures analogiques et/ou logiques. Les schémas structurels utilisés ne sont en fait rien de plus qu'un outil de description graphique mais l'ampleur des fonctions numériques à réaliser impose l'utilisation d'un autre outil de description. En effet, la complexité des circuits et des traitements qu'ils réalisent n'a cessé de croître et les schémas de description devenaient trop volumineux et trop complexes. Les langages textuels HDL sont donc une solution pour la description de circuits (ASIC (*Application-Specific Integrated Circuits*), etc) de grosse taille.

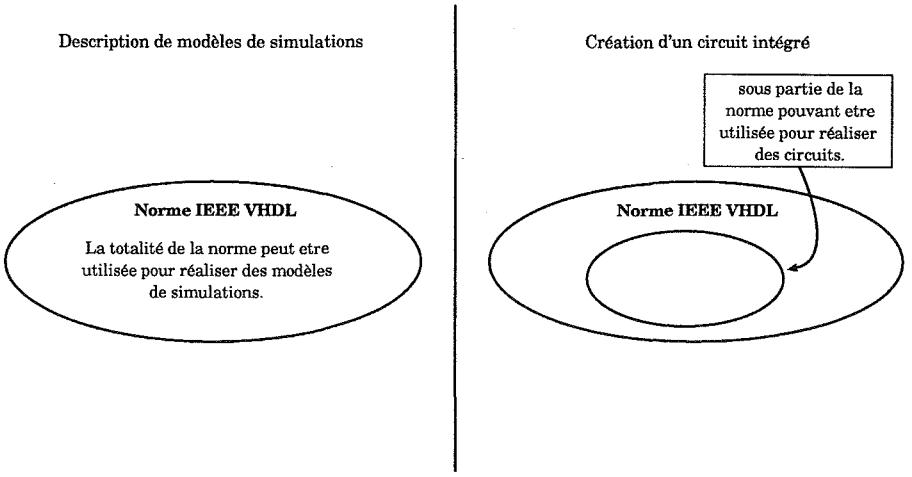


FIG. 1.1 – Différentes natures de descriptions VHDL

VHDL

Développé dans les années 80 aux Etats-Unis, le langage de description VHDL (*Very high speed integrated circuits Hardware Description Language*) est ensuite devenu une norme IEEE en 1987 [105]. Révisée en 1993 pour supprimer quelques ambiguïtés et améliorer la portabilité du langage, cette norme est devenue un standard en matière d'outils de description. La norme définissant la syntaxe et les possibilités du langage VHDL restent très ouvertes :

- Un système peut être décrit très précisément (temps de propagation des signaux, temps de transitions, etc) afin de réaliser un modèle de simulation.
- Seule une sous-partie du langage est *synthétisable* c'est-à-dire peut être traduite par un outil de synthèse en une implantation matérielle avec une technologie donnée [3].

La figure 1.1 résume les deux utilisations du langage VHDL. Le code VHDL d'un additionneur 1 bit est donné, à titre d'exemple, ci-dessous. Le code est constitué de deux parties distinctes :

- l'entité donne les informations concernant les signaux d'entrée et de sortie. Dans l'exemple ci-dessous, nous construisons une entité `architecture` ayant deux ports d'entrée `A` et `B` et deux ports de sortie `Som` et `Ret`.
- l'architecture décrit le comportement de l'entité. Pour une entité donnée, il est possible de créer plusieurs architectures distinctes. Dans l'exemple simple que nous traitons, il s'agit simplement d'utiliser les portes logiques élémentaires afin de calculer le résultat de la somme et de positionner la retenue.

```
entity additionneur is
port (A,B : in bit; Som, Ret : out bit);
end additionneur;

architecture arch_addi of additionneur is
begin
  Som <= A xor B;
  Ret <= A and B;
end arch_addi;
```

Vérification de description HDL

La vérification d'une description HDL se fait essentiellement par la génération de bancs de tests. La liste des entrées de la description HDL permet de produire des *vecteurs de tests* qui sont les différentes

[3] Les temps de propagation dépendent par ailleurs de la technologie choisie

valuations possibles des entrées. On peut donc, en théorie, construire un banc de test exhaustif où toutes les configurations d'entrées seront testées. Il faut ensuite analyser la cohérence des valeurs de sorties de la description HDL en fonction de ses entrées.

Une autre technique couramment employée est l'analyse de couverture de code qui repose sur le principe simple que toute ligne d'un programme a une utilité en soi. Ce type d'analyse se sert souvent de métriques (fonctionnelles, structurelles ou autres) permettant de noter une description à l'aide de certains critères.

De nombreux travaux, comme ceux de Glässer [59, 60] ou Van Tassel [109], donnent une définition formelle de la sémantique de VHDL qui initialement était imprécise. D'autres travaux ont pour but de vérifier les descriptions VHDL à l'aide de model-checker [50, 95]. De telles approches restent cependant limitées à cause du phénomène d'explosion combinatoire du nombre d'états. En effet, la description "assez bas niveau" de circuits implique la prise en compte d'un nombre important de détails qui gènent la vérification simple de propriétés sur des descriptions importantes.

Résultat de synthèse

Un autre problème est celui de la synthèse de circuit depuis une description VHDL correcte du point de vue de sa simulation. En effet, la synthèse d'une *netlist* (réseau de portes logique) depuis une description VHDL ne produit pas toujours des résultats cohérents même lorsque la description source n'utilise que le sous-ensemble du langage VHDL synthétisable.

1.1.2 Le langage Verilog HDL

Le langage Verilog HDL est également un langage de description de circuits logiques, utilisé pour la conception d'ASIC et de FPGA (*Field-Programmable Gate Array*).

A l'origine, il s'agissait d'un langage propriétaire, développé par la société Cadence Design Systems (`http ://www.cadence.com/`), pour être utilisé dans leurs simulateurs logiques, mais le succès grandissant de VHDL a incité ses concepteurs à en faire un standard ouvert. Verilog HDL a maintenant atteint cet objectif : c'est le standard IEEE 1364 [106].

La structure du langage Verilog permet de décrire les entrées et les sorties de modules électroniques, pour définir des portes logiques virtuelles. La combinaison de modules permet de réaliser des schémas électroniques virtuels complexes qu'il est alors possible de tester dans un programme de simulation. De tels tests ont pour objectif de :

- valider le comportement des circuits décrits (le résultat qu'ils délivrent est bien celui attendu) ;
- valider les performances de ces circuits (ils répondent dans un temps donné et les signaux qui parcourent les différents modules sont correctement synchronisés)

Le langage Verilog reste très similaire dans ses concepts au langage VHDL. Plus récent, il a cependant évité de tomber dans les mêmes travers que celui-ci (sémantique plus clairement définie) et utilise une syntaxe assez proche du langage C.

1.1.3 Bilan

La conception de circuits électroniques a progressé avec l'explosion de la complexité des fonctions numériques et avec la taille des circuits à réaliser, en proposant des langages de description appropriés et des outils de synthèse de circuits, sorte de "compilateurs matériels", permettant un plus haut niveau d'abstraction. De ce fait, les architectures électroniques des circuits peuvent être validées avant la réalisation, permettant de détecter préalablement un certain nombre d'erreurs.

Avec l'arrivée des System on Chip (SoC) ou systèmes enfouis qui mélangent dans une même architecture parties matérielles et logicielles, les langages de description HDL sont devenus des langages spécialisés dans la description des parties matérielles. La complexité et les nombreux problèmes de la conception des System on Chip nécessitent un nouveau niveau d'abstraction pour permettre de décrire les applications dans leur ensemble en prenant en compte la communication, l'exécution parallèle et les natures différentes des composants d'un système enfoui.

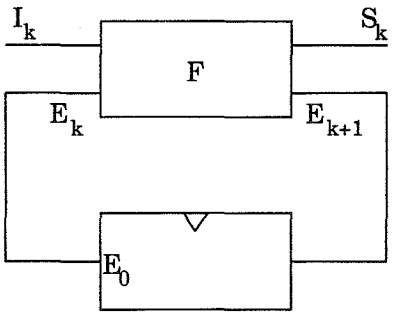


FIG. 1.2 – Implantation synchrone d'une machine d'états finis

1.2 Les langages synchrones

Les langages synchrones comme Esterel [29, 28], Lustre [67] ou encore Signal [78] sont particulièrement bien adaptés à la description de systèmes *réactifs*. Les systèmes réactifs sont des systèmes qui réagissent au rythme des événements extérieurs (émis par l'environnement). Tous les événements extérieurs sont pris en compte et le système choisi ceux qui ne sont pas traités s'il est surchargé. Dans ce cas, on dit que le système fonctionne en mode dégradé.

1.2.1 Généralités sur les langages synchrones

Les trois langages Esterel, Lustre et Signal sont basés sur les mêmes concepts mathématiques et ils ne diffèrent réellement que par la seule la syntaxe [24]. Les langages synchrones modélisent les systèmes réactifs comme des machines d'états finis. Ils reposent sur une sémantique mathématique rigoureuse qui rend possible les validations formelles de propriétés de sûreté et de vivacité en utilisant des techniques de model checking par exemple.

Modèle formel utilisé

L'apparition de ces langages dans les années 80 était motivée par le besoin d'exprimer simplement et conjointement les notions de concurrence et de synchronisme. Un point particulièrement important est la volonté de préserver un modèle formel le plus simple possible, en particulier dans la définition de la composition parallèle de deux processus. La concurrence est un paradigme de base de ces langages et elle s'exprimer simplement.

Cependant, combiner les notions de concurrence et de programmation synchrone tout en conservant un modèle mathématique simple n'est pas trivial. Un principe de base de la programmation synchrone est la discrétisation du temps. On peut considérer alors qu'un programme synchrone est la succession d'un certain nombre de *réactions atomiques*. L'équation $P \equiv R^{\omega}$, avec R l'ensemble des réactions possibles et l'exposant ω indiquant une itération, définit abstraitement le comportement d'un système synchrone.

A chaque instant un système synchrone se trouve dans un état donné, avec des valeurs de sorties et d'entrées spécifiques. Le calcul des valeurs de sorties se fait en fonction de l'état courant et des valeurs des entrées. L'état courant du système est mémorisé et peut être modifier par la fonction de calcul. On peut donc considérer qu'un système réactif synchrone fonctionne suivant l'équation (représentée figure 1.2) :

$$E_{k+1}, S_k = F(I_k, E_k) \tag{1.1}$$

avec I_k les entrées (vecteur) du système, S_k les sorties du systèmes et E_n l'état courant du système. La figure 1.2 illustre l'équation en question qui est caractéristique de l'implantation matérielle d'une machine à états finis.

Par ailleurs, il convient de choisir une définition simple et naturelle pour la composition parallèle. L'équation 1.2 suivante dénote la définition choisie. Cette définition de la composition de processus demande donc la conjonction des réactions de chaque composant (processus). Il est bien connu que la conjonction renforce les *contraintes*. On a donc finalement l'équation générale suivante définissant la composition parallèle dans les langages synchrones :

$$P_1 \| P_2 \equiv (R_1 \wedge R_2)^\omega \tag{1.2}$$

Ces définitions des systèmes réactifs et, en particulier, de la composition peuvent être étudiées et appliquées de différentes manières qui correspondent aux différents langages existants.

1. *microsteps* : on peut voir la réaction d'un système comme une séquence de micro-pas élémentaires. Dans cette approche, les composants primitifs des systèmes sont donc vus comme des séquences de micro-pas et la composition parallèle de ces composants primitifs est réalisée par l'enchevêtrement des évaluations de ces micro-pas selon une manière appropriée. Ce choix n'est pas en accord avec la définition simple 1.2 et ne respecte pas la conjonction. Cependant, c'est cette approche qui est utilisée dans des langages tels que VHDL, Verilog ou encore dans les Statecharts. La sémantique de micro-pas reste confuse [24] et de nombreuses interprétations différentes coexistent [60, 59].
2. *Acyclique* : les ingénieurs en électronique insistent souvent sur le fait que leurs diagrammes de blocs (représentant un circuit) ne contiennent pas de boucle directe [4]. Le langage Lustre adopte ce point de vue et impose syntaxiquement au programmeur l'écriture de programme ne contenant pas de boucle.
3. *Point fixe* : cette approche considère que chaque réaction du système est la solution d'une équation du point fixe. Chaque réaction est une fonction déterministe de la forme :

   $$\{state, input\} \mapsto \{nextstate, output\} \tag{1.3}$$

   Compiler un programme avec cette sémantique est difficile car il est nécessaire de prouver que les relations entre les différentes réactions d'un programme ont toujours une solution. Le langage Esterel a cependant adopté ce point de vue et propose un compilateur efficace permettant la compilation des programmes Esterel.
4. *Contraintes* : cette approche considère que chaque réaction d'un programme peut avoir zéro solution (le programme est bloqué et n'a pas de réaction), une solution de la forme 1.3, ou plusieurs solutions qui sont interprétées comme un comportement non-déterministe. Si l'implantation demande une unique solution à chaque réaction, les autres cas sont intéressants dans le cas de spécification de haut niveau. Le langage Signal a adopté cette approche.

Nous allons rapidement survoler les spécificités de chaque langage synchrone afin de comprendre leurs différences et leurs intérêts.

1.2.2 Lustre : langage synchrone flot de données

Le but de Lustre est de proposer un langage de programmation basé sur un modèle très simple de flot de données [66]. Les concepteurs de circuits électroniques travaillent très souvent avec des systèmes d'équations (différentielles, booléennes, etc...) ou des réseaux de flots de données. Dans ce type de formalisme, chaque variable qui n'est pas une entrée est exprimée comme fonction des autres variables. Ainsi, l'expression $x = y + z$ signifie qu'à chaque instant k (ou à chaque pas) on a $x_k = y_k + z_k$.

En d'autres termes, chaque variable est fonction du temps ce qui suppose une discrétisation du temps : en Lustre, toute variable ou expression est une séquence infinie de valeurs. Ainsi un programme Lustre est une boucle infinie et chaque variable ou expression prend la $k^{\text{ième}}$ valeur de sa séquence au $k^{\text{ième}}$ pas de la boucle.

Lustre propose la notion de *nœud* afin d'aider le programmeur à structurer son programme. Un *nœud* est une fonction de flots : il utilise un certain nombre de flots d'entrée typés et définit un certain nombre de flots de sortie par un système d'équations qui n'utilise que des flots *locaux* (variables locales).

[4] Une boucle directe est l'utilisation d'une sortie d'un bloc comme entrée de ce même bloc dans la même phase de calcul.

La notion de *type structuré* est également disponible et est indispensable pour réaliser des applications de taille réelle. La version commerciale de Lustre, Scade, propose par exemple la possibilité de déclarer et d'utiliser des tableaux.

Enfin pour conclure cette rapide présentation de Lustre, nous allons nous intéresser au *clock calculi.* En effet, il est souvent intéressant d'activer différentes parties d'un programme à différentes vitesses. Les langages synchrones de flots de données proposent cette possibilité en utilisant des *horloges.* Ainsi, en Lustre, un programme a une horloge de base qui est la notion de temps la plus fine. Certains flots peuvent avoir des horloges plus lentes, c'est-à-dire n'avoir de valeurs que durant certains cycles de l'horloge de base. Le *clock calculi* consiste donc à vérifier que :

- tout opérateur de plus d'un opérande doit être appliqué à des opérandes partageant la "même" horloge.
- les horloges d'opérandes d'un nœud doivent respecter les conditions définies dans le nœud.

1.2.3 Esterel

Alors que Lustre est un langage déclaratif qui se focalise sur la spécification de flot de données, le langage Esterel [29] est impératif et s'attache à la description du contrôle d'une application. Intuitivement, un programme Esterel consiste en la description, à l'aide d'une syntaxe impérative traditionnelle, d'un ensemble de processus concurrents. Son exécution est synchronisée par une unique horloge, globale à tout le programme. Au début de chaque pas du programme, chaque processus reprend son exécution (généralement après une instruction `pause`), et exécute un code standard (affecter des valeurs à des expressions...) et finalement se termine ou se suspend en préparation du prochain pas.

Les processus communiquent exclusivement par l'usage de signaux : ils représentent des événements globaux. Les signaux sont utilisés afin de représenter des événements et, par conséquent, ils ne se comportent pas comme les variables standards d'un langage impératif mais plutôt comme des fils dans un circuit électronique.

Comme mentionné précédemment, Esterel considère un pas d'exécution (ou une réaction) comme une équation du point fixe. Par conséquent, Esterel autorise l'écriture de programme contenant des cycles et pour lesquels une analyse statique du code signale un interblocage alors que, dynamiquement, le programme ne peut jamais atteindre un état dans lequel le cycle est actif. Vérifier qu'un programme Esterel n'est jamais en interblocage s'appelle *l'analyse de causalité* et demande de s'assurer que les contraintes de causalité ne sont jamais contredites dans aucun des états atteignables du programme. Ces contraintes de causalité découlent des structures de contrôle et des dépendances de données et un problème de causalité se traduit par des cycles dans la partie "combinatoire" qui empêchent l'utilisation de ces circuits dans les outils de synthèse. En effet, il est possible d'écrire un programme Esterel syntaxiquement correct mais qui contient des cycles de dépendance entre des signaux ou des variables.

Plus formellement, la sémantique d'Esterel est basée sur la notion de logique *constructive* [27]. La compréhension de ce problème et son traitement mathématique correct sont une des contributions les plus importantes d'Esterel au domaine des langages synchrones.

Esterel a également apporté beaucoup dans les techniques de compilation des langages synchrones. Les compilateurs Esterel V1 et V2 construisaient un automate, partant d'un programme Esterel en utilisant l'algorithme de Brzozowski [32] sur les expressions régulières. Plus tard, les travaux de Gonthier [61] ont permis d'accélérer nettement le processus de construction d'automates (FSM). Cette technique, si elle est très efficace sur de petits programmes (moins de mille lignes de code) ne résistait pas au passage à l'échelle des exemples industriels à cause de phénomènes d'explosion du nombre d'états.

La seconde idée de compilation a donc été de traduire les programmes Esterel en circuit électronique de manière directe [26] (au niveau porte logique). Du fait de la sémantique d'Esterel, cette traduction est naturelle et a l'énorme avantage de produire quelques portes logiques pour chaque instruction de programme. De ce fait, la compilation de grands programmes est largement envisageable. Les exécutables issus de cette technique de compilation simulent simplement un réseau de portes logiques (netlist).

Par la suite, de nouvelles techniques de compilation ont été proposées permettant la traduction d'une classe réduite de programmes en code C [53]. Le code ainsi obtenu est un code de *simulation* qui est rapide et efficace et comparable en taille au code généré pour du matériel. Le problème principal de ces techniques de compilation était qu'un nombre important de programmes Esterel corrects ne pouvaient être

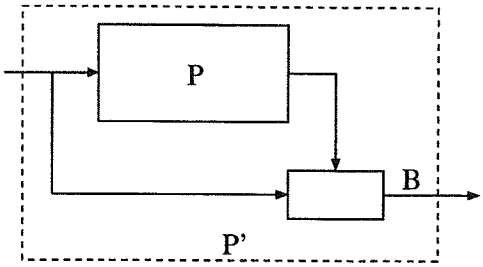


FIG. 1.3 – Vérification du programme P par l'observeur B

compilés. Des travaux d'optimisations, en particulier l'introduction d'un nouveau format intermédiaire [91], tentent de résoudre ce problème.

1.2.4 Signal

Le langage Signal a été essentiellement créé afin de permettre une spécification plus aisée des systèmes en autorisant la description de systèmes ayant des comportements non déterministe (dans le cadre d'une spécification et pas d'une implantation)[78, 79]. Contrairement aux programmes, les systèmes sont généralement considérés comme ouverts : un système peut être mis à jour par ajout, suppression ou remplacement d'un de ses composants.

Signal permet de décrire les systèmes sous la forme de diagrammes de blocs. Un bloc représente un composant ou un sous-système et peut être connecté à d'autres blocs permettant ainsi la description de larges blocs hiérarchiques.

Si P est un système ouvert, il peut évoluer dans un environnement qui est considéré comme actif lorsque P n'y est pas. Les blocs communiquent entre eux par l'intermédiaire de *signaux*. Dans Signal, chaque signal a sa propre horloge et l'horloge générale d'un programme est le supremum des horloges des différents signaux. Les signaux sont donc des séquences typées (booléens, entiers, ...) dont le domaine est enrichi de la valeur qui représente l'absence de signal.

1.2.5 Vérification et langages synchrones

Les langages synchrones permettent la vérification de propriétés temporelles par model checking. Les opérateurs temporels de la logique temporelle peuvent être assez facilement exprimés dans les langages synchrones (en particulier en Lustre qui peut être considérer comme un sous-ensemble d'une logique temporelle). Dans le cas de Lustre, des propriétés de sûreté peuvent être exprimées et vérifiées par l'intermédiaire de noeud prévus à cet effet. L'utilisation d'un même langage pour les programmes et leurs propriétés permet une utilisation facile de ces mécanismes.

La technique des observeurs synchrones [68] est un autre moyen de vérifier les propriétés d'un programme. Le schéma de la figure 1.3 illustre ce procédé. Pour prouver que l'expression B est un invariant du programme P, il suffit de construire un nouveau programme P' constitué du corps de P et des équations définissant B. La seule sortie du programme P' est B. Vérifier que la propriété B est un invariant du programme P revient à vérifier, lors de l'exécution, que la sortie du programme ne prend jamais la valeur $false$. En Lustre, un mécanisme d'assertion est également disponible permettant la description de propriétés.

Une fois encore, un des intérêts d'une telle approche est que les propriétés à vérifier sont écrites dans le même langage que le programme. Cet argument est surtout valable pour convaincre les programmeurs d'écrire des spécifications formelles : ils n'ont pas besoin d'apprendre un nouveau formalisme.

1.3 Langages "système" et outils

Afin d'outiller le flot de conception modèle (cf figure 2, page 5) et d'automatiser certaines étapes, des études ont proposé un grand nombre de langages. Dans [33], Bunker et d'autres référencent différents langages utiles à la spécification de SoC. La diversité et le nombre important de méthodes et langages chargés d'assister la conception d'un SoC montrent que ce domaine est à la frontière de plusieurs autres et que les solutions proposées ne sont pas encore satisfaisantes. De nombreux langages informatiques ont été étendus afin de supporter la description de parties matérielles permettant ainsi l'utilisation d'un unique langage de description lors de la conception.

1.3.1 Extensions du langage C et des langages HDL

Un certain nombre de travaux essayent d'outiller la conception de SoC à l'aide d'extension des langages de description HDL. Les langages SystemVerilog [25] et SuperLog [56] sont des extensions des langages VHDL et Verilog destinés à englober l'intégralité du flot de conception.

Le langage SpecC [52] est une extension du langage C permettant la conception de systèmes enfouis. Le langage SpecC supporte l'essentiel des concepts des systèmes enfouis comme la concurrence, la synchronisation et différents modes de communications. SpecC a pour cible la spécification et la conception de SoC [51] incluant des parties logicielles et matérielles ou bien des blocs synthétisés provenant de langages de description. Ce langage basé sur C est une première tentative de plate-forme générale pour la conception de systèmes enfouis.

Basé sur le même genre d'approche, le langage HardwareC [77] propose également d'utiliser la syntaxe du langage C comme langage commun de développement de SoC. La sémantique des descriptions matérielles de HardwareC est très différente de la sémantique classique C ANSI et de nombreux ajouts ont été réalisés afin d'augmenter l'expressivité du langage. Une limitation importante de ces propositions est le fait qu'elles reposent sur le langage C qui ne permet pas toujours de décrire facilement un grand nombre d'entités et d'objets interagissant ce qui est toujours le cas dans un SoC. Une approche objet semble plus prometteuse car les concepts objets supportent plus naturellement cette diversité et l'interaction des différentes structures. C'est la motivation de langage comme SystemC.

L'Open SystemC Initiative (OSCI) est une association indépendante à but non lucratif ayant pour but la promotion du langage SystemC [88]. Le langage SystemC est en train de devenir un standard de conception de systèmes enfouis. En effet, SystemC commence à combler la distance existant entre les langages HDL et les méthodes de développement de logiciels embarqués (essentiellement C/C++). Le "langage" SystemC comprend une bibliothèque de classes C++ et un noyau de simulation permettant la description d'architectures à différents niveaux (comportemental TLM [89] ou RTL [74]). Un exemple de programme SystemC décrivant une porte AND est donné dans l'annexe B (page 199).

La sémantique de SystemC n'est pas clairement établie et le comportement d'un programme SystemC est défini par l'ordonnanceur fourni. Le langage SystemC suscite de nombreux travaux comme ceux d'Habibi [57, 65] ou d'autres [100, 58], visant à fixer sa sémantique de manière formelle éventuellement par le biais de son ordonnanceur comme Mueller et Ruf [97] qui propose une modélisation formelle du scheduler de SystemC 1.0 à l'aide d'ASM[5] mais SystemC a beaucoup évolué depuis.

De nombreux outils de vérification sont disponibles pour SystemC 2.0. Mueller et d'autres [97, 98] ont proposé et développé un model checker spécifique à SystemC permettant d'observer et de vérifier les valeurs en transit sur les canaux de communication. Basé sur la logique FLTL [99], ce model checker permet d'exprimer des propriétés de sûreté dans un formalisme assez simple. Le fait de ne pouvoir décrire que des propriétés concernant les signaux de communication est cependant assez limitant.

Dans ses travaux [86, 85], Moy propose des solutions plus complètes à l'aide d'une boite à outils pour l'analyse de systèmes décrits en SystemC. Plutot centrés sur les descriptions de niveau TLM, ces outils produisent une représentation intermédiaire HPIOM (*Heterogeneous Parallel Input/Output Machines*) qui peut-être couplée avec des outils de vérifications. Ces travaux définissent une sémantique formelle exécutable de SystemC. Différents types de propriétés peuvent être définies et vérifiés au niveau TLM comme l'absence d'interblocage ou encore l'unicité d'écriture d'un signal au cours d'un cycle de simulation.

[5] Un travail similaire des mêmes auteurs a également été réalisé sur SpecC [87].

1.3.2 Autres langages textuels

Le langage Java est également utilisé comme langage de description commun aux différentes étapes de descriptions du système [23]. Un important avantage de Java est la possibilité de simuler facilement une exécution concurrente de processus par le mécanisme des *threads* et des *rendez-vous* qui sont intégrés dans le langage. Comme dans les autres langages orientés objet utilisés pour la description de systèmes enfouis, Java permet l'identification des processus comme des méthodes (synchronisées) et les modules composant le SoC sont des objets.

Un nouveau langage, Rosetta, défini par Alexander [11, 10] travaille à un niveau d'abstraction plus élevé. Il permet de spécifier l'ensemble des besoins et des contraintes d'une application et de s'abstraire des considérations d'implantation du ou des langages employés.

L'idée de Rosetta est de décrire de vastes systèmes par l'intermédiaire de différentes vues ou *facettes*. Une vue est un modèle d'un composant du système ou du système lui-même qui décrit les informations d'un domaine spécifique (analogique, digital, mécanique...). Afin de permettre une conception simple, chaque vue du système doit utiliser un modèle différent munit d'une sémantique et d'un vocabulaire spécifique. Les différentes vues sont donc écrites afin de définir les différents aspects du système et l'assemblage des différentes vues permet la construction de modèles représentant l'intégralité du système étudié.

L'exemple suivant décrit une facette ayant pour nom `sort-req` et qui représente un composant de tri. Les variables `i` et `o` sont les entrées et sorties de la facette. Le domaine spécifique de la facette est `state-based`, permettant de décrire les transitions entre un état construit à partir des valeurs des sorties et l'état suivant. La notation `x'` est utilisée pour indiquer la valeur de la variable `x` à l'état suivant. Les deux termes `l1`, `l2` définissent les spécifications fonctionnelles de la vue alors que `c1` exprime une contrainte de consommation.

```
facet sort-req(i:: in sequence(T);
               o:: out sequence(T)) is
  power :: real;
begin state-based
  l1: permute(o',i);
  l2: ordered(o');
  c1: power <= 10mW;
end sort-req;
```

Dans le cadre des SoCs, il n'est pas suffisant de considérer indépendamment chaque information spécifique d'un domaine. En effet, les interactions sont une des causes majeures d'erreurs de conception.

1.3.3 Les langages graphiques de spécification

L'utilisation initiale de langages et d'outils graphiques est importante dans le monde de l'électronique et la spécification d'un système est souvent réalisée par l'intermédiaire de tels langages. La spécification fonctionnelle parallèle repose sur un modèle d'exécution qui décrit le comportement des tâches. Les modèles de comportement se différencient par leurs granularités (processus, objet), leurs mécanismes de communication inter-tâches (passage de messages, variables partagées ...), leurs mécanismes de synchronisation, leurs orientations (vérification formelle, synthèse, simulation), ... Nous allons présenter rapidement quelques modèles concernant la spécification fonctionnelle qui s'appuie essentiellement sur le modèle des machines à états finis.

Les *machines à états finis* ou *Finite State Machine (FSM)* font parties des modèles de calcul les plus utilisés pour la description de composants numériques. Elles sont composées de noeuds représentant les différents états du système et d'arcs représentant les transitions. Les transitions sont étiquetées par des conditions de transitions appelées *gardes* et par des *actions* représentant la génération des sorties. Ce modèle reste cependant trop proche de l'architecture matérielle, il est donc assez peu utilisable pour des modélisations faisant abstraction de l'implantation. Par ailleurs, le nombre d'états du modèle croît de façon exponentielle avec la taille du système modélisé.

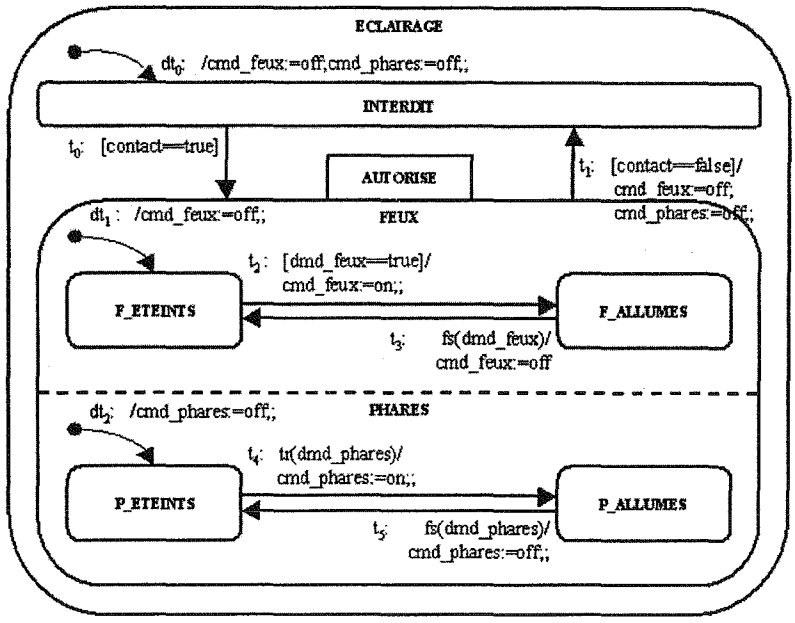


FIG. 1.4 – Exemple de StateCharts décrivant un logiciel embarqué

D. Harel a introduit dans [70] le concept de *Hierarchical and Concurrent Finite State Machine (HCFSM)* qui permet, à l'aide d'un langage de représentation graphique, les *StateCharts*, une réduction exponentielle de la complexité. Les spécifications réalisées à l'aide de StateCharts [20] permettent de visualiser le comportement du système. En effet, les StateCharts constituent une description hiérarchique de systèmes (les descriptions de systèmes complexes sont construites comme des compositions de systèmes plus simples) et décrivent l'exécution concurrente. De plus, ce type de représentation a l'avantage de permettre la génération d'un modèle exécutable et la production d'une partie du code [94]. La figure 1.4 présente un exemple de StateCharts décrivant un logiciel embarqué contrôlant la commande d'éclairage d'un véhicule.

Plus tard, tirant les leçons des langages synchrones comme Esterel, une approche synchrone des StateCharts a été proposée par C. André : les *SyncCharts* [13, 14].

Les *Co-Design Finite State Machine (CFSM)* [45] utilisent des FSMs auxquelles a été ajouté un mode de communication asynchrone réalisé par des "buffers" à une place. Les événements émis (soit par une CFSM ou par son environnement) sont détectés par une ou plusieurs CFSMs. L'émission d'un événement n'est pas bloquante et le tampon (buffer) d'une place existant entre l'émetteur et les récepteurs mémorise l'événement jusqu'à ce qu'il soit détecté ou écrasé.Un modèle de temps réel discret est utilisé et les CFSMs réalisent des tâches en un temps non nul et non borné. Le trop bas niveau de description des CFSMs a motivé la création des *Abstract Co-Design Finite State Machine (ACFSM).*

Une ACFSM [101] est un réseau de FSMs communiquant par événements à travers des signaux sans perte (files FIFO infinies). Cette première spécification sous forme de réseaux d'ACFSM peut être raffinée afin d'obtenir des *Extended Co-Design Finite State Machine (ECFSM)* qui sont des réseaux de FSMs communiquant par l'intermédiaire de files FIFO bornées. Les ECFSMs permettent la modélisation de communication avec perte (files FIFO bornées) et les CFSMs ne sont qu'un cas particulier de ECFSM communiquant par des files à une place.

D'autres modèles sémantiques sont utilisés comme le modèle CSP (*Communicating Sequential Processes*) de Hoare [72] ou encore les réseaux de processus (*Kahn Process Network*) [75, 43] qui sont très utilisés et assez bien adaptés aux applications orientées traitement du signal[6].

[6]Ces applications traitent en général des signaux continus et réalisent des calculs assez coûteux comme des FFT (Fast

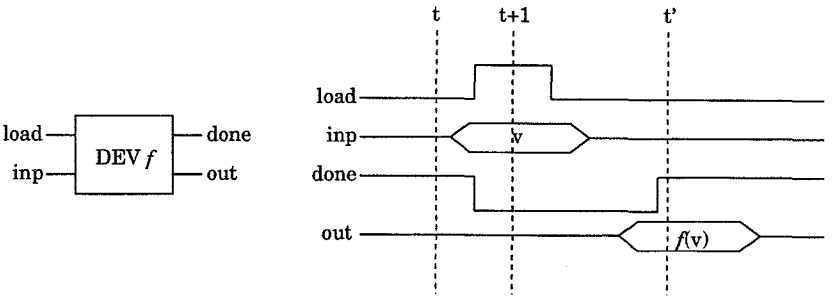


FIG. 1.5 – Nature du circuit résultat de traduction

1.3.4 Autres outils de vérification et de validation

Preuves de microcode en ACL2

L'utilisation de méthodes de validation basées sur la preuve reste relativement marginale dans le cadre des systèmes enfouis. Cependant les travaux dans ce domaine ont fourni des résultats intéressants. Dans [82], Moore et d'autres ont validé le microcode de la division en virgule flottante du processeur AMD5$_K$86 a été validé par l'utilisation du prouveur automatique d'ACL2. Le langage ACL2 est un sous ensemble de Common Lisp ce qui permet de considérer des descriptions ACL2 comme des programmes Lisp exécutables ou comme une description formelle utilisable par le prouveur automatique. Le microcode de la division a été traduit en une représentation intermédiaire puis découpée en une série d'assertions formelles en ACL2. Les assertions produites sont toutes, après quelques transformations, validées par le prouveur automatique ACL2 [31].

Génération de circuits à l'aide de HOL

Dans une autre approche [62], basée sur HOL (*Higher Order Logic*) [12], Gordon définit la traduction de fonctions récursives en Verilog HDL . Un compilateur, implanté en HOL4, est disponible et la compilation produit un théorème de correction (qui doit être prouvé) pour chaque fonction compilée. Des formules logiques représentant le circuit sont également générées et la traduction en Verilog HDL est quasi immédiate. Des composants primitifs sont disponibles et sont utilisés pour la construction (à partir de règles) de circuits plus complexes. Les circuits ainsi produits sont corrects par construction, les règles étant prouvées. Les portes AND et NOT, composants de base des circuits, sont décrites par les théorèmes (prouvés) suivants :

$$\vdash \texttt{AND}(in_1, in_2, out) = \forall t.\ out\ t = (in_1\ t\ \wedge\ in_2\ t)$$

$$\vdash \texttt{NOT}(inp, out) = \forall t.\ out\ t = \neg(inp\ t)$$

Le résultat de la traduction d'une fonction f définie en HOL est un circuit qui calcule f comme présenté figure 1.5. Une valeur v est présente sur l'entrée `inp` quand une requête (front montant) a lieu sur l'entrée `load`. Une valeur $f(v)$ est calculée et devient la valeur de sortie `out`, la sortie `done` signalant la fin du calcul.

L'utilisation d'un prouveur automatique permet d'accélérer la validation des descriptions ainsi produites et remplace la simulation et les tests de la description Verilog HDL. Seul un sous-ensemble de HOL est "synthétisable", l'ensemble des fonctions "tail-recursives", mais de futurs travaux visent à généraliser l'approche et à enrichir ce premier sous-ensemble.

Fourrier Transformations).

Langage et outils BlueSpec

Un langage comme BlueSpec [30], basé sur le standard SystemVerilog, permet de décrire, de simuler de générer un système. Basé sur des règles de réécriture [15], BlueSpec SystemVerilog (BSV) propose un modèle d'exécution formel différent des langages VHDL ou SystemC. L'exemple de code suivant décrit un compteur 8 bits à l'aide d'une interface (`Counter`) et de son implantation (`mkCounter`). L'interface permet de définir un ensemble de comportement à l'aide de méthodes et de leurs profils et l'implantation réalise les méthodes définies.

```
// interface
interface Counter;
   method Bit#(8) read();
   method Action Load(Bit(#8) newval);
   method Action increment();
endinterface

//implantation
module mkCounter(Counter) ;
   Reg#(Bit#(8)) value <- mkReg(0);

   method Bit#(8) read();
     return value;
   endmethod

   method Action Load(Bit(#8) newval);
     value <= newval;
   endmethod

   method Action increment();
     value <= value+1;
   endmethod
endmodule
```

Un mécanisme d'assertions permet de vérifier des propriétés du modèle décrit, et l'utilisation d'un système de règles de réécriture permet des vérifications de propriété par model checker et par prouveur automatique. Par ailleurs, BlueSpec permet la génération de codes sous différentes formes (langage C ou description Verilog au niveau RTL synthétisable) ce qui le rend également attractif pour la description de SoC.

L'environnement RAVEN

Des outils de vérification (model checker) et d'aide au développement sont également apparus comme l'environnement RAVEN [96] qui permet de développer des systèmes à différents niveaux d'abstraction et de vérifier des propriétés temporelles. Les systèmes étudiés sont décrits dans RAVEN comme un ensemble de modules interagissant. Les différents modules sont connectés par des canaux de communication. RAVEN propose quelques algorithmes permettant l'analyse du système (absence d'interblocages ou encore propriétés de vivacité). Le model checker proposé par l'outil utilise la logique temporelle CCTL (*Clocked Computation Tree Logic*) et produit un contre-exemple qui explicite une exécution provoquant la violation des propriétés spécifiées.

L'environnement Forte

Les laboratoires d'Intel proposent un environnement de vérification formelle, Forte, qui permet de vérifier et de valider de larges développements industriels [76]. L'environnement Forte permet de décrire des circuits et de les valider en combinant des méthodes de model checking avec un prouveur automatique

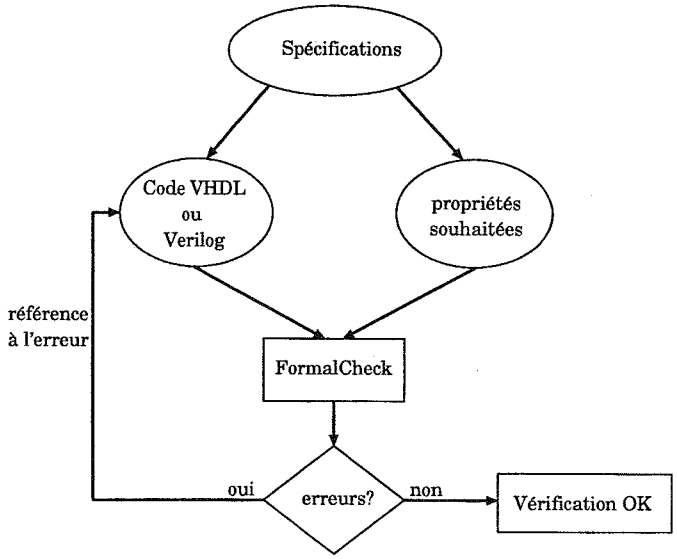


FIG. 1.6 – Cycle de vérification FormalCheck

[21]. Cette combinaison est réalisée à l'aide d'un langage fonctionnel, FL, qui propose des primitives temporelles. Deux model checker sont utilisés dans l'environnement Forte. L'un utilise la STE (*Symbolic Trajectory Evaluation*), une technique basée sur le model checking symbolique et la simulation symbolique [102] : il s'agit de donner en entrée des symboles plutôt que des valeurs, des techniques de preuve sont ensuite utilisées pour vérifier que le système possède les propriétés attendues. Le second model checker utilise la logique LTL (*Linear Time Logic*) et est dédié aux vérifications de circuits arithmétiques.

FormalCheck, outil de vérification

Un outil du même genre, FormalCheck [63], est proposé par la société Cadence. L'outil FormalCheck est un model checker qui supporte les sous-ensembles synthétisables des deux langages Verilog et VHDL. Les propriétés devant être vérifiées sont décrites séparemment en utilisant des brouillons (*templates*). La conception du SoC est validé par la vérification de propriétés temporelles attendues sur une description relativement terminale du système. FormalCheck permet la vérification de propriétés de sûreté mais aussi de vivacité. La démarche appliquée est assez similaire à celles des autres outils déjà décrits et est présentée à la figure 1.6.

1.4 Conception de SoC et raffinement formel

Le raffinement apparaît comme une solution intéressante pour distribuer la complexité des systèmes dans leurs différents niveaux de description (cf chapitre suivant).

Le formalisme des *Actions Systems* de Back [19] est inspiré des commandes gardées de Dijkstra [49], supporte le raffinement et permet de décrire des systèmes parallèles distribués. Les Actions Systems sont des commandes gardées de la forme $g \rightarrow S$ avec g une condition booléenne, la *garde* et S, le *corps*, une opération du langage de commande. Le comportement d'un système est décrit par des actions qui peuvent être réalisées en parallèle si elles n'ont pas de variables communes (ou qui respectent certaines contraintes). Les actions restent également atomiques : si une action est choisie pour être exécutée, son exécution se déroule sans interférence en provenance d'autres actions du système.

Dans ses travaux [103], Kaisa Sere modélise des systèmes électroniques à l'aide des Actions Systems et utilise le raffinement pour déduire les comportements des circuits. Dans [104], Sere décrit la possibilité d'une instruction de retour d'appel de procédure. Ce mécanisme augmente les possibilités de parallélisme dans les modélisations d'algorithmes distribués.

Dans [18], Back et Sere définissent une série de principes permettant de déduire, d'une spécification sous la forme d'un Actions System, un système modulaire. Ils partent du constat que les approches à la UNITY [44], manquent d'outils de décomposition aidant à la modularisation. Les auteurs définissent un opérateur de composition parallèle d'Actions Systems qui permet de construire une nouvelle spécification. Afin de proposer une décomposition modulaire correcte, les auteurs définissent également des règles de partage des variables entre Actions Systems.

Dans les travaux de Plosila [90], l'utilisation des Actions Systems et du raffinement permet la définition d'un cadre de conception formelle pour microprocesseurs pipelinés : la spécification initiale du processeur est une unique action qui est raffinée, pas à pas, en des actions atomiques simples. L'unique système initial est décomposé en des composants réactifs, communiquant par l'intermédiaire de variables partagées, qui modélise les différentes parties fonctionnelles du microprocesseur. Les preuves de raffinement permettent de garantir un fonctionnement cohérent du circuit vis-à-vis du cahier des charges.

Une autre étude [73], concerne les circuits reconfigurables (FPGA) et étudie l'applicabilité des Actions Systems dans le partitionnement logiciel et matériel. Dans ces travaux, l'accent est mis sur la qualité de la spécification initiale (réalisée à l'aide des Actions Systems). Le partitionnement et les développements réalisés à partir de cette première spécification sont écrits dans le langage HandelC et restent assez informels.

Dans le cadre de la conception de SoC, des travaux ont été réalisés sur la modélisation de circuits électroniques à l'aide de la méthode B. Abrial, dans [5], pose un ensemble de principes de modélisation de base pour des circuits mais plus généralement pour des systèmes réactifs : le système et son environnement "prennent la main" à tour de rôle. D'autres travaux, [37], ont consisté à modéliser (toujours à l'aide de la méthode B) des composants logiques de base comme un additionneur et un multiplieur de n bits. Partant de la définition mathématique (et exacte) de ces opérations, le processus de raffinement permet d'obtenir un système utilisant des séquences bornées de bits et indiquant un dépassement. Une fois encore, les preuves de raffinement et d'invariance permettent de montrer que le système final est une implantation correcte du système initial.

Plus récemment, Zimmermann et d'autres [36] définissent des modèles B permettant de décrire le comportement de circuit. Plusieurs raffinements successifs produisent une description B très proche du langage VHDL : ce niveau de raffinement est appelé BHDL. A partir de ce niveau de raffinement, la production de circuits devient automatique. Les modèles proposés dans [69] décrivent un système de traitement discret de signaux normalement spécifié à l'aide de systèmes d'équations mathématiques. L'implantation traditionnelle de ce genre de système est une machines à états finis. Les différents modèles B permettent d'établir formellement le lien entre le niveau abstrait, les systèmes d'équations mathématiques, et le niveau concret (l'implantation).

Ces différentes approches basées sur le raffinement formel permettent également l'utilisation d'IP. Dans [111], Zimmermann présente une méthodologie permettant de réutiliser des composants matériels non développés dans le cadre de la méthode B. L'idée principal de cette approche est d'écrire une spécification B du composant et de prouver, à l'aide d'ACL2, que la description VHDL de celui-ci implante bien la spécification. Par ailleurs, le développement d'une bibliothèque de composants prouvés peut également permettre le maintien d'une cohérence globale.

1.5 Bilan

Des méthodes formelles adaptées à la conception de circuits et plus généralement de systèmes complexes existent. Les langages synchrones, grâce à une sémantique précise, proposent un cadre rigoureux et clair pour la description de circuits. A l'aide du mécanisme d'observeur, le concepteur d'un circuit peut même vérifier l'invariance de propriétés décrites dans le même langage que le programme. Un mécanisme d'assertions est également proposé et permet un certain nombre de vérifications et la formulation d'hypothèses sur le comportement de l'environnement du système. Cependant, le mécanisme d'observeur, très proche des techniques de model checking se heurte au même problème combinatoire d'explosion du nombre d'états. De plus la complexité intrinsèque des systèmes décrits (circuits électroniques de contrôle, ...) ne permet pas de vérifier facilement des propriétés intéressantes sur ces systèmes très complexes.

Par ailleurs, des environnements de conception et des outils permettent de vérifier, souvent par si-

mulation, la cohérence des descriptions et le respect du cahier des charges. Ici encore, la complexité des systèmes étudiés est un problème. Le niveau d'abstraction TLM tente de régler ce problème en se déchargeant des considérations purement électroniques (cycle d'horloge, temps de propagation des signaux, ...) et en ne considérant que les communications entre blocs. La co-existence de plusieurs niveaux d'abstractions pour décrire le même système demande de vérifier la cohérence des descriptions entre-elles. Dans la plupart des approches, la relation de raffinement employée pour réaliser cette vérification n'est pas formellement définie ; des scenarii de tests permettent de vérifier que le système conçu a un comportement similaire dans les différentes descriptions.

Les travaux visant à assister la conception d'un SoC en s'appuyant sur une définition formelle du raffinement ne sont pas assez avancés pour permettre un développement réel. Une utilisation systématique du raffinement permet une conception unifiée et améliore la qualité des SoC produits en garantissant une tracabilité depuis le cahier des charges.

Chapitre 2

La méthode B événementielle

La méthode B événementielle est une méthode formelle proposée par Jean-Raymond Abrial [6] qui permet de modéliser des systèmes fermés réagissant avec l'extérieur. Après Z et B [2], Abrial propose cette nouvelle méthode, évolution de la méthode B [3], basée sur l'hypothèse du monde clos ou système fermé : le système est représenté avec son environnement et l'ensemble des interactions est présent dans le modèle. La méthode B événementielle utilise la théorie des ensembles et le langage de substitutions généralisées de B. La preuve et le raffinement sont au coeur de ce formalisme qui permet de développer progressivement des modèles de systèmes (informatiques ou non) et de prouver des propriétés sur ces systèmes à l'aide d'un invariant.

Utilisée dans de nombreux contextes et projets industriels de grande échelle [22, 80], la méthode B a fait preuve de sa stabilité et de son applicabilité. Le fait que cette méthode soit outillée la rend plus abordable et permet de mécaniser le processus de preuve. La vérification de modèles est par conséquent accélérée et rend la validation de système par méthode formelle envisageable dans des développements réels.

L'utilisation d'un formalisme mathématique comme la théorie des ensembles et le raffinement des systèmes permet de raisonner sur des systèmes purement informatiques [7] mais aussi sur des systèmes n'ayant pas de rapport immédiat avec ce domaine comme la description d'un système de contrôle d'accès [4]. Par ailleurs le raffinement B permet de distribuer la complexité des problèmes et des travaux dans le cadre de l'algorithmique distribuée ont montré son intérêt [9].

Dans un premier temps, nous allons présenter la théorie du raffinement et illustrer à l'aide de systèmes de transitions les propriétés de la relation de raffinement. Nos présenterons par la suite les principes et la syntaxe de la méthode B événementielle.

2.1 La théorie du raffinement

Le raffinement est une approche pour construire des programmes corrects et fiables. L'idée est de dériver par étapes successives une spécification initiale en vérifiant que chaque transformation du modèle préserve sa correction au regard du modèle précédent. Dans les années 70, Wirth puis Dijkstra introduisent la notion de raffinement (en utilisant les commandes gardées [49]) qui est par la suite formalisée par Back [17]. Une série de travaux a ensuite détaillé cette notion [19, 16]. Caroll Morgan a également poursuivi des travaux sur le raffinement [83] et a étudié, par la suite, l'apport d'éléments probabilistiques à la description de modèles et leurs éventuels impacts dans le processus de raffinement [71].

Selon le modèle utilisé, il existe différentes notions de raffinement qui ne sont pas toujours équivalentes. En effet, la notion de raffinement est définie par le choix de différentes propriétés et l'on peut considérer des notions de raffinement plus ou moins "strictes" selon les propriétés le définissant. Cependant, la correction du raffinement est montrée, la plupart du temps, par la vérification de conditions suffisantes sur la relation le définissant. Le raffinement est un moyen de construire progressivement des programmes corrects ou plus généralement de décrire progressivement des systèmes complexes en préservant les propriétés des descriptions abstraites. L'objectif est de garantir, par la preuve, que les comportements des différentes

descriptions d'un même système sont cohérents.

2.1.1 Raffinement et correction de programmes

L'intérêt des méthodes formelles est la vérification ou la validation de manière rigoureuse des propriétés d'un modèle formel. Un modèle formel permet de raisonner mathématiquement sur un système. Dans la logique de Hoare, un programme S est associé à une précondition P et une postcondition Q. La précondition P caractérise les états avant l'exécution de S, tandis que la postcondition Q est vérifiée après l'exécution de S.

Un programme S est totalement correct vis-à-vis de sa spécification (P, Q) si, à partir de tout état initial vérifiant la précondition P, le programme termine et fait passer le système dans un état satisfaisant la postcondition Q. Si la terminaison n'est pas assurée, la correction est dite partielle. Un programme S totalement correct par rapport à sa spécification (P, Q) est dénoté dans la suite par $[P]\ S\ [Q]$.

Le problème de la correction de programmes suppose que le programme S existe déjà. Si le programme S n'existe pas ou n'est pas connu, le problème consiste à dériver un programme correct simple. On cherche donc un programme S' tel que :

$$\forall\ P, Q\ .\ ([P]\ S'\ [Q]\ \Rightarrow\ [P]\ S\ [Q])$$

Cette propriété exprime le fait que le programme S' satisfait toute spécification satisfaite par S. On peut donc substituer S' à S en continuant à satisfaire la spécification initiale. S' est un raffinement de S noté :

$$S\ \sqsubseteq\ S'$$

La relation de raffinement $\sqsubseteq$ vérifie deux propriétés importantes :
- réflexivité, un programme est son propre raffinement.
- transitivité, un programme peut être raffiné par plusieurs étapes successives.

De ce fait, la dérivation peut être réalisée de manière extrêmement progressive ce qui permet de distribuer la complexité du programme dans les différentes descriptions de celui-ci et de ce fait de raisonner plus simplement sur les différents programmes ainsi construits. Il est cependant difficile de comparer des expressions de la forme $[P]\ S\ [Q]$ qui sont constituées de trois paramètres P, S et Q. Plusieurs sémantiques ont permis de simplifier les vérifications liées à la notion de raffinement de modèle en fixant ou contraignant certains paramètres. La correction et le raffinement dépendent de la sémantique utilisée pour interpréter la relation $[P]\ S\ [Q]$. Back détaille un certain nombre de sémantiques que nous allons reprendre afin de montrer la richesse du raffinement et son intérêt dans la description de programmes et de systèmes.

Sémantique de transformateurs de prédicats

Dans le but de vérifier la correction des programmes, Dijkstra [49] a introduit la notion de *plus faible précondition* (notée wp pour *weakest precondition*). Cet opérateur wp permet de calculer la plus faible précondition garantissant la terminaison d'un programme S et la satisfaction de la postcondition Q associée à ce programme.

Pour un programme S et sa postcondition Q, $wp(S, Q)$ est la plus faible précondition P telle que $[P]\ S\ [Q]$. La correction de programme peut être exprimée dans cette sémantique. Un programme S de précondition P et de postcondition Q est correct si sa précondition P permet de déduire $wp(S, Q)$.

$$P\ \Rightarrow\ wp(S, Q)$$

Une interprétation naturelle de ceci est que la précondition P du programme est plus forte (ou contraigante) que $wp(S, Q)$. La correction de S vis à vis de sa spécification (P, Q) est donc assurée. On peut maintenant exprimer le raffinement à l'aide de l'opérateur wp de la manière suivante :

$$S\ \sqsubseteq\ S'\ \Leftrightarrow\ (\forall\ Q.wp(S, Q)\ \Rightarrow\ wp(S', Q))$$

La sémantique de l'opérateur *wp* est dite *backward* dans le sens où elle permet de revenir sur les préconditions possibles pour un programme S et une postcondition Q. Le raffinement ainsi défini permet de construire des programmes concrets corrects : si l'on peut déduire de la plus faible précondition $wp(S, Q)$ de la spécification S la plus faible précondition $wp(S', Q)$ du programme S', celui-ci respecte les préconditions et les postconditions de la spécification S. La méthode des Actions Systems de Back [19] est basée sur cette définition du raffinement.

Sémantique du choix

Si la sémantique *wp* est *backward* et permet de reconstruire des préconditions depuis les postconditions, la sémantique du choix est une sémantique *forward*. Elle permet de considérer les postconditions possibles d'un programme S en fonction de la précondition P. Si l'on considère un programme S de précondition P, de postcondition Q et l'ensemble des postconditions noté $sp(S, P)$ (*strongest postcondition*), le programme S est correct si :

$$Q \Rightarrow sp(S, P)$$

On peut, intuitivement, dire que cette sémantique associe à tout état initial un ensemble de postconditions possibles pour l'état final. On exprime la notion de raffinement entre deux programmes S et S' dans cette sémantique par :

$$S \sqsubseteq S' \Leftrightarrow (\forall P.\ sp(S, P) \subseteq sp(S', P))$$

En définissant le raffinement de cette manière, chaque étape de raffinement permet de construire une description S' dont l'ensemble des postconditions $sp(S', P)$ contient l'ensemble des postconditions de la spécification $sp(S, P)$. Enfin, la sémantique du choix est duale à la sémantique *wp*, pour tout couple (P, Q) pre et postconditions d'un programme S, on a :

$$(P \Rightarrow wp(S, Q)) \Leftrightarrow (Q \Rightarrow sp(S, P))$$

Ces deux définitions du raffinement reposent sur des choix arbitraires qui permettent de garantir la conservation de propriétés désirées et d'une certaine correction.

Système de transitions étiquetées

Il est également possible de considérer le raffinement dans le cadre des systèmes de transitions. Un système de transitions étiquetées (LTS) est un quintuplet (E, I, A, t, v) avec :

- un ensemble d'états E.
- un sous-ensemble d'états initiaux I, tel que $I \subseteq E$.
- un ensemble d'actions (ou étiquettes de transitions) A.
- une relation de transition t telle que $t \subseteq E \times A \times E$.
- une fonction v qui associe à chaque état s du système des valeurs aux variables du modèle.

Un chemin entre deux états s_a et s_b de E est une séquence finie de transitions $a_0, ..., a_n$ de t telle qu'il existe une séquence d'états $s_1, ...s_n$ de E de la forme :

$$s_a \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \ldots \xrightarrow{a_{n-1}} s_n \xrightarrow{a_n} s_b$$

Une définition du raffinement dans le cadre des systèmes de transition peut être également choisie. Ici encore les choix réalisés définissent une relation de raffinement spécifique. Soient deux systèmes de transitions, $S = (E, I, A, t, v)$ et $S' = (E', I', A', t', v')$, le lien entre les états abstraits de S et les états concrets de S' se fait par un invariant de collage Inv et une relation $\alpha \subseteq E' \times E$ entre les états telle que :

$$s'\ \alpha\ s \Leftrightarrow ((v'(s') \wedge Inv) \Rightarrow v(s))$$

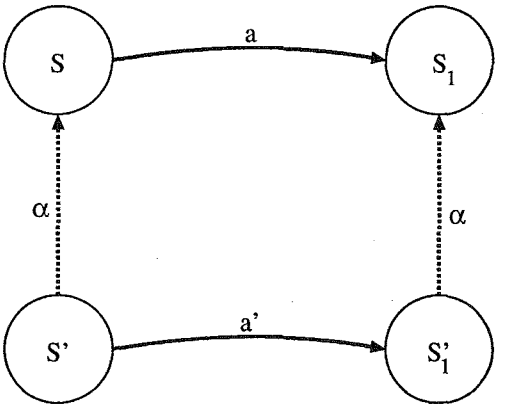


FIG. 2.1 – Raffinement : conservation des comportements

Une telle propriété affirme que deux états (abstrait et concret) s et s' sont liés par la relation de raffinement α uniquement si l'on peut déduire les valeurs des variables abstraites à l'état s des valeurs des variables concrètes à l'état s' et de l'invariant.

Prouver que le système S est raffiné par le système S' revient à montrer un certain nombre de propriétés entre les deux systèmes. Nous allons passer en revue les différentes propriétés permettant de garantir qu'un système raffine un autre.

Comportements Tout d'abord, pour chaque transition du système de transition concret S', il doit exister une transition correspondante dans le système de transition abstrait.

$$(s' \ \alpha \ s \ \wedge \ s' \xrightarrow{a'} s'_1 \in t') \ \Rightarrow \ \exists \ s_1 \ \exists \ a.(s \xrightarrow{a} s_1 \in t \ \wedge \ s'_1 \ \alpha \ s_1)$$

Cette propriété traduit le fait que les comportements du système de transitions concret sont des comportements du système de transitions abstrait. Toutes les transitions entre états sont déjà prévues dans l'abstraction. La figure 2.1 exprime cette relation entre les comportements abstraits et concrets des deux systèmes de transitions étiquetées.

Bégaiement Une transition muette τ est une transition entre deux états concrets correspondant au même état abstrait. L'apparition de ces transitions muettes au niveau concret provoque ce que l'on appelle un bégaiement de transition (*stuttering*) au niveau abstrait.

$$(s' \ \alpha \ s \ \wedge \ s' \xrightarrow{\tau} s'_1 \in t') \ \Rightarrow s'_1 \ \alpha \ s$$

Dans ce cas, le raffinement du système consiste à séparer des états abstraits en sous-états concrets. Les transitions de ces sous-états concrets sont donc masqués dans le système abstrait qui reste dans le même état. La figure 2.2 illustre graphiquement ce principe de transitions muettes ou de bégaiement de transition dans l'abstraction.

Blocage Un état bloquant d'un système de transitions étiquetées est un état n'ayant aucune transition (sortante) avec d'autres états et qui bloque le système de transitions. Le raffinement doit préserver le comportement du système et par conséquent ne doit pas introduire de nouveaux blocages. Si l'on introduit l'ensemble b des états bloquants de S tel que $b \subseteq E$ et b' l'ensemble des états bloquants de S' tel que $b' \subseteq E'$ on a alors :

$$(s' \ \alpha \ s \ \wedge s' \in b') \ \Rightarrow \ s \in b$$

Cette propriété exprime le fait que le système de transition concret n'apporte pas plus de blocage que l'abstraction. Une fois encore, définir une relation de raffinement respectant cette propriété est un choix arbitraire mais qui permet une conservation "stricte" des comportements de l'abstraction.

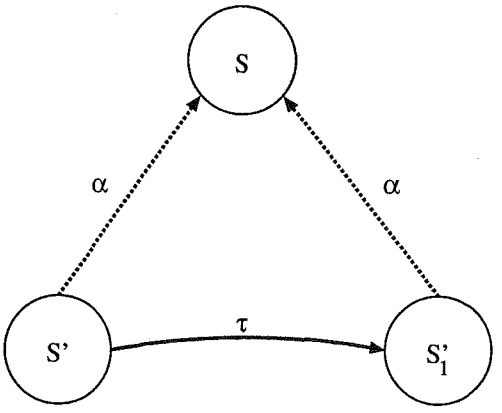


FIG. 2.2 – Raffinement : transition muette

Variant Afin d'éviter une prise de contrôle infinie des nouvelles transitions on doit vérifier la non-divergence. Si l'on suppose qu'un état s de E est associé à plusieurs états concrets par la relation r, les k états concrets sont définis par les variables s'_i avec $i \in \{1, ..., k\}$.

$$\forall\, k.(k \geq 0 \ \wedge\ s'_k\ \alpha\ s \ \Rightarrow\ \exists\, a, k'.(a \in L_1 \ \wedge\ k' \geq k \ \wedge\ s'_{k'-1} \xrightarrow{a} s'_{k'} \in t'))$$

La propriété précédente exprime le fait que les états abstraits restent vivaces, c'est-à-dire qu'ils sont atteints et que les états concrets raffinant un état abstrait ne forment pas une boucle infinie. L'existence d'une telle boucle aurait pour conséquence l'apparition d'un nouveau bloquage dans l'abstraction.

Initialisation A tout état initial concret s' doit correspondre un état initial abstrait s :

$$\forall\, s' \in I' \ \ \exists\, s \in I \ \ s' \alpha\ s$$

Le raffinement doit préserver l'initialisation du système. De ce fait, l'état initial concret doit être cohérent avec l'état initial abstrait.

2.2 Méthode B événementielle : modèle abstrait

Un modèle B événementiel est la description d'un système de transition. Nous avons déjà présenté dans le cadre du raffinement les systèmes de transition. Un modèle B est donc un ensemble de variables, un invariant et un ensemble d'événements qui sont des actions gardées (dans le même sens que les Action Systems de Back [19]).

2.2.1 Structure d'un modèle

Les variables x du modèle doivent satisfaire l'invariant $I(x)$ qui est un prédicat sur ces variables. Les événements du modèle, quant à eux, décrivent la dynamique du système. Un modèle peut également décrire des ensembles abstraits (permettant le typage) et des constantes ainsi que leurs propriétés. Un modèle B aura donc la structure suivante :

MODEL
  m
SETS
  s
CONSTANTS
  c
PROPERTIES
  $P(s,c)$
VARIABLES
  x
INVARIANT
  $I(x)$
ASSERTIONS
  $A(x)$
INITIALISATION
  <une substitution>
EVENTS
  <liste d'événements>
END

- tout modèle a un nom m,
- la clause SETS permet de définir les ensembles s utilisés dans le modèle. Les ensembles définis dans cette clause sont indépendants les uns des autres et toujours non-vides. Les ensembles en question permettent le typage des variables et une certaine généricité.
- la clause CONSTANTS permet de définir une liste de constantes c utilisées dans le modèle. Ces constantes permettent de décrire le contexte, l'environnement et permettent la paramétrisation.
- la clause PROPERTIES contient un prédicat $P(s,c)$. Ce prédicat définit les constantes c déclarées dans la clause CONSTANTS ou encore les ensembles définis dans la clause SETS.
- la clause VARIABLES permet de définir la liste de variables x du modèle.
- la clause INVARIANT introduit le prédicat $I(x)$ qui définit les variables de la liste x et qui doit rester vrai dans chaque état du système. Des obligations de preuves seront générées pour vérifier que l'activation de chaque événement préserve cet invariant.
- la clause ASSERTIONS permet de définir un prédicat $A(x)$. Une assertion doit être prouvée à l'aide de l'invariant et des propriétés du modèle.
- la clause INITIALISATION permet de définir les valeurs initiales des variables x. C'est une substitution.
- la clause EVENTS permet de définir la liste des événements avec leur noms et leurs définitions.

L'exemple d'en-tête suivante est issu de développements réels réalisés dans le cadre du projet RNRT EQUAST :

MODEL
  synchro
SETS
  $VALEUR = \{OK, KO, IND\}$
VARIABLES
  TS_sync_loss
INVARIANT
  $TS\_sync\_loss \in VALEUR$
INITIALISATION
  $TS\_sync\_loss := IND$

L'en-tête précédent définit le modèle *synchro*. Un ensemble $VALEUR$ est défini en listant les valeurs le composant puis la variable TS_sync_loss est déclarée. L'invariant du modèle est un simple prédicat

de typage et l'initialisation consiste en une substitution simple (voir ci-dessous). Des exemples d'en-têtes plus complets et complexes seront présentés par la suite.

La forme générale d'un modèle étant présentée nous allons maintenant étudier plus en détail le langage des substitutions du B événementiel et nous intéresser à la structure des différentes formes d'événements.

2.2.2 Les substitutions

Le langage des substitutions généralisées B est très simple : une substitution est une expression qui précise comment changent les variables du modèle. Il existe plusieurs formes de substitution dont la plus générale est la définition d'un prédicat explicitant les valeurs *avant et après substitution* des variables modifiées. Ce prédicat est nommé le prédicat *avant-après* ou *before-after* en anglais. Dans la suite de cette partie, la notation x_0 désignera la valeur précédente (ou valeur avant) des variables et la notation x désignera la nouvelle valeur (ou valeur après).

Substitution généralisée

La substitution généralisée est la substitution la plus générale. Elle permet d'exprimer les autres substitutions sous *forme normale* et est de la forme $x : P(x_0, x)$ qui signifie que x peut prendre n'importe quelle valeur qui satisfait le prédicat $P(x_0, x)$ avec x_0 la valeur précédente de la variable x.

Par exemple on peut considérer la substitution suivante qui affecte à la variable TS_sync_loss la valeur IND.

$$TS\_sync\_loss : (TS\_sync\_loss = IND)$$

Substitution simple

La substitution simple est une substitution qui ressemble à une affectation. Elle est de la forme $x := E(x)$ où $E(x)$ est une expression fonction de x. Sa forme normale, exprimée sous forme de substitution généralisée, est $x : (x = E(x_0))$.

$$TS\_sync\_loss := IND$$

Substitution ensembliste

Cette dernière substitution permet d'affecter une valeur dans un ensemble donné à une variable et se note $x :\in S(x)$. La variable x peut donc prendre n'importe quelle valeur contenue dans l'ensemble $S(x)$. La forme normale de cette substitution est $x : (x \in S(x_0))$. L'initialisation de la variable TS_sync_loss avec la valeur IND de l'ensemble $VALEUR$ peut donc s'écrire :

$$TS\_sync\_loss :\in \{IND\}$$

Plus généralement, la substitution $TS\_sync\_loss :\in VALEUR$ affecte une valeur de l'ensemble $VALEUR$ à la variable TS_sync_loss.

Substitutions parallèles

L'opérateur $\|$ est la composition parallèle de deux substitutions. Les deux substitutions ont lieu en même temps mais ne doivent pas concerner les mêmes variables, on a ainsi $x : P(x_0, x) \| y : Q(y_0, y)$ avec x et y des listes de variables n'ayant aucune variable en commun. La forme générale associée à la composition parallèle de deux substitutions est $x, y : (P(x_0, x) \wedge Q(y_0, y))$.

Nom	Syntaxe	Définition
Relation binaire	$s \leftrightarrow t$	$\mathcal{P}(s \times t)$
Domaine	$\mathsf{dom}(r)$	$\{a \mid a \in s \ \wedge \ \exists b \cdot (b \in t \ \wedge \ a \mapsto b \in r)\}$
Codomaine	$\mathsf{ran}(r)$	$\mathsf{dom}(r^{-1})$
Identité	$\mathsf{id}(s)$	$\{x, y \mid x \in s \ \wedge \ y \in s \ \wedge \ x = y\}$
Restriction	$s \lhd r$	$\mathsf{id}(s); r$
Co-restriction	$r \rhd t$	$r; \mathsf{id}(s)$
Anti-restriction	$s \ \lhd\!\!\!- r$	$(\mathsf{dom}(r) - s) \lhd r$
Anti-co-restriction	$r \ \rhd\!\!\!- t$	$r \rhd (\mathsf{ran}(r) - t)$
Image	$r[w]$	$\mathsf{ran}(w \lhd r)$
Surcharge	$q \ \lhd\!\!\!+ r$	$(\mathsf{dom}(r) \ \lhd\!\!\!- q) \cup r$
Fonction partielle	$s \nrightarrow t$	$\{r \mid r \in s \leftrightarrow t \ \wedge \ (r^{-1}; r) = \mathsf{id}(t)\}$
Fonction totale	$s \rightarrow t$	$\{f \mid f \in s \nrightarrow t \ \wedge \ \mathsf{dom}(f) = s\}$
Injection partielle	$\rightarrowtail$	$\{f \mid f \in s \nrightarrow t \ \wedge \ f^{-1} \in t \nrightarrow s\}$
Injection totale	$\rightarrowtail$	$s \rightarrowtail t \cap s \rightarrow t$
Surjection partielle	$\twoheadrightarrow$	$\{f \mid f \in s \nrightarrow t \ \wedge \ \mathsf{ran}(f) = t\}$
Surjection totale	$\twoheadrightarrow$	$s \twoheadrightarrow t \cap s \rightarrow t$
Bijection partielle	$\rightarrowtail\!\!\!\!\twoheadrightarrow$	$s \rightarrowtail t \cap s \twoheadrightarrow t$
Bijection totale	$\rightarrowtail\!\!\!\!\twoheadrightarrow$	$s \twoheadrightarrow t \cap s \twoheadrightarrow t$

FIG. 2.3 – Principales notations ensemblistes du B événementiel

Prédicat avant-après des substitutions

Le tableau ci dessous résume les prédicats avant-après des différentes substitutions présentées. Comme dit précédemment, ce prédicat lie les valeurs des variables avant et après substitution.

Substitution	**Prédicat Avant-Après**
$x : P(x_0, x)$	$P(x_0, x)$
$x := E(x)$	$x = E(x_0)$
$x :\in S(x)$	$x \in S(x_0)$
$x : P(x_0, x) \| y : Q(y_0, y)$	$P(x_0, x) \ \wedge \ Q(y_0, y)$

La figure 2.3 résume la plupart des opérations ensemblistes proposées en B événementiel. Dans cette figure r est supposée être une relation de s vers t ($r \in s \leftrightarrow t$).

2.2.3 Forme des événements

Un événement B modélise une transition du système dynamique étudié. Un événement se produit instantanément et représente un changement d'état du système. La méthode B événementielle propose trois différentes sortes d'événements. En réalité, on peut ramener tous les événements à une même et unique écriture, les autres types d'événements définis étant simplement des facilités syntaxiques. De manière générale, un événement est composé d'une garde et d'une substitution : il se déclenche lorsque sa garde est vraie et effectue son action (la substitution, donnée sous sa forme générale ici) qui correspond au changement d'état.

Evénement indéterministe

L'événement indéterministe est l'événement le plus complet proposé dans la méthode B événementielle. Il est composé d'une liste de variable locale l, d'un prédicat sur les variables locales l et sur les variables globales x et d'une substitution. Cet événement ne s'active que si le prédicat $G(x, l)$ est vrai et modifie les valeurs de x à l'aide des variables locales l éventuellement définies.

name $\widehat{=}$
ANY l WHERE
$G(x, l)$
THEN
$x : P(x_0, x, l)$
END

Evénement gardé

L'événement gardé est plus simple que l'événement indéterministe; il ne propose pas de liste de variables locales. Son activation reste cependant conditionné par une garde $G(x)$, prédicat portant sur les variables globales du système.

name $\widehat{=}$
SELECT
$G(x)$
THEN
$x : P(x_0, x)$
END

Evénement simple

L'événement simple est un événement non gardé (ou dont la garde est toujours vraie) et qui est toujours activé. Cet événement est donc uniquement composé d'une substitution.

name $\widehat{=}$
BEGIN
$x : P(x_0, x)$
END

Prédicats Avant-Après liés aux événements

Les différentes sortes d'événements présentés, il convient de définir les relations existantes entre les valeurs des variables avant et après activation d'un événement. En effet, ces relations auront une grande importance lors de la preuve de la préservation de l'invariant par l'activation des événements du modèle. Le tableau ci-dessous résume, pour les différents événements, les prédicats Avant-Après associés. On considère la substitution $S(x)$ et son prédicat avant-après associé P. Les notations x et x' représentent les valeurs de la variable x avant et après le déclenchement de l'événement considéré.

Evénement	Garde	Prédicat Avant-Après
BEGIN $S(x)$ END	$true$	$P(x, x')$
SELECT $G(x)$ THEN $S(x)$ END	$G(x)$	$G(x) \wedge P(x, x')$
ANY t WHERE $G(t, x)$ THEN $S(t, x)$ END	$\exists t \cdot (G(t, x))$	$\exists t \cdot (G(t, x) \wedge P(x, x', t))$

2.2.4 Obligations de preuves

Nous allons détailler ici les obligations de preuves associées à un modèle. Ces obligations de preuves sont des prédicats devant être prouvés et qui assurent la cohérence interne du modèle. Elles sont de différentes natures : preuve d'assertion, faisabilité de substitution et enfin préservation d'invariant.

Preuve d'assertions

Une assertion est un prédicat sur les variables et les constantes qui doit être prouvé à l'aide de l'invariant $I(x)$ et des propriété des constantes et des ensembles abstraits $P(s, c)$.

$$P(s, c) \wedge I(x) \Rightarrow A(x) \quad \text{(ASS1)}$$

Ces preuves d'assertions permettent de prouver un théorème décrit comme assertion. Le théorème peut alors être utilisé pour aider à la preuve des autres obligations de preuve.

Faisabilité de substitution liée à un événement

La méthode B événementielle considère que la substitution d'un événement est toujours faisable sous couvert de la garde de l'événement et l'invariant du modèle. On considère que des substitutions non faisables sont interdites (comme $x :\in \emptyset$), la cohérence des modèles va donc être assurée par des obligations de preuves spécifiques à la faisabilité des substitutions.

Lors de l'initialisation, $x : Init(x)$, il faut donc prouver qu'il existe bien une valeur remplissant les conditions d'initialisation des variables. On a donc l'obligation de preuves suivante :

$$\exists x \cdot Init(x) \quad \text{(FIS1)}$$

Pour qu'un événement soit faisable, il faut qu'une transition du système de transition soit possible. La garde de l'événement considéré et l'invariant du système permettent de caractériser cette transition. On doit donc montrer qu'il existe une valeur x qui satisfait le prédicat avant-après de la substitution sous couvert de la garde de l'événement et de l'invariant. Le tableau ci-dessous présente l'obligation de preuve générée pour chaque type d'événement :

Evénement	Faisabilité
BEGIN $x : P(x_0, x)$ END	$I(x) \Rightarrow \exists x' \cdot P(x, x')$
SELECT $G(x)$ THEN $x : P(x_0, x)$ END	$I(x) \wedge G(x) \Rightarrow \exists x' \cdot P(x, x')$
ANY l WHERE $G(l, x)$ THEN $x : P(x_0, x, l)$ END	$I(x) \wedge G(l, x) \Rightarrow \exists x' \cdot P(x, x', l)$

Preuve de préservation de l'invariant

Enfin les dernières obligations de preuves visent à assurer la préservation de l'invariant. Ces obligations de preuves permettent de vérifier que les variables x du système ne violent pas l'invariant $I(x)$: nous dirons que l'invariant doit être préservé par les différentes activations des événements du modèle.

Pour que l'invariant soit préservé, il doit déjà être établi lors de l'initialisation du système. Une première obligation de preuve est de vérifier que l'initialisation établit bien l'invariant.

$$\boxed{Init(x) \Rightarrow I(x)} \quad \text{(INV1)}$$

Une fois l'invariant prouvé, l'activation d'un événement doit préserver l'invariant. Pour chaque événement du modèle B, on va générer une obligation de preuve de la forme :

$$\boxed{I(x) \wedge BA(x, x') \Rightarrow I(x')} \quad \text{(INV2)}$$

avec $BA(x, x')$ le prédicat Avant-Après associé à l'événement considéré. Dans cette obligation de preuve, on suppose que le système est dans un état satisfaisant (l'invariant $I(x)$ est supposé vrai) et l'on vérifie, à l'aide des transformations définies par la transition (le prédicat Avant-Après $BA(x, x')$), que les nouvelles valeurs des variables respectent l'invariant. Les transformations définies par la transition sont simplement les modifications des variables définies par les substitutions réalisées dans la partie action de l'événement considéré.

2.2.5 Exemple

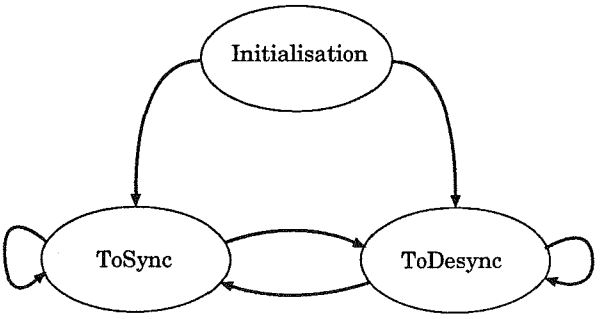
Nous présentons ici l'intégralité du modèle B qui nous a servi à valider un algorithme de synchronisation entre un récepteur et un émetteur dans le cadre du projet EQUAST. Nous présenterons plus en détail le système étudié au chapitre 3, mais nous utilisons ici, à titre d'exemple, un modèle issu de nos travaux.

Dans cette étude de cas, l'émetteur émet des paquets constituant un flux sur un média quelconque. Afin de traiter les paquets, le récepteur doit détecter un octet spécifique de synchronisation au début du paquet. Un récepteur est considéré comme synchronisé lorsqu'il a reçu un certain nombre de paquets débutant par l'octet de synchronisation. La perte de synchronisation a lieu quand l'émetteur est synchronisé et qu'il reçoit plusieurs paquets successifs ayant un octet de synchronisation défaillant.

Le modèle abstrait présenté ici reste très général et se contente de fixer les *objectifs* de l'algorithme : les raffinements successifs auront la charge de détailler comment ces objectifs seront réalisés. Le système de transition présenté à la droite de l'entête illustre le modèle B événementiel. Ce système très simple est composé de trois états distincts représentant l'initialisation du modèle ainsi que le passage d'un état

synchronisé à un état désynchronisé et vice-versa. Les transitions d'un état à un autre représentent le déclenchement d'un événement du modèle.

MODEL
synchro
SETS
$VALEUR = \{OK, KO, IND\}$
VARIABLES
TS_sync_loss
INVARIANT
$TS\_sync\_loss \in VALEUR$
INITIALISATION
$TS\_sync\_loss := IND$
EVENTS
ToDesync $\widehat{=}$
BEGIN
$TS\_sync\_loss :\in \{KO, IND\}$
END ;
ToSync $\widehat{=}$
BEGIN
$TS\_sync\_loss := OK$
END
END



Le modèle présenté ci-dessus est très simple. Les deux événements ToSync et ToDesync sont non gardés et modifient la variable TS_sync_loss. La variable TS_sync_loss représentent l'état de synchronisation du recépteur :

- à l'initialisation, le récepteur n'a encore reçu aucun paquet il n'est ni synchronisé ni en état d'erreur. La variable TS_sync_loss est donc initialisée à IND.
- le récepteur est synchronisé, la variable TS_sync_loss est égale à OK.
- le récepteur n'est pas synchronisé, la variable TS_sync_loss est différente de OK.

Le modèle abstrait se contente de décrire un système qui passe d'un état synchronisé à désynchronisé et inversement. Cependant, du fait de l'absence de garde on peut également imaginer qu'un même événement s'exécute toujours sans pouvoir laisser l'autre prendre la main. Ce comportement n'est pas du tout contradictoire avec l'étude de cas étudiée : selon les données, le système pourrait ne pas réussir à obtenir la synchronisation et l'événement ToSync ne pourrait jamais s'exécuter. Le présent modèle est une abstraction et est par conséquent relativement indéterministe (pas de gardes, substitution indéterministe). Il répond cependant à son rôle qui est de souligner les deux états distincts du système. Les obligations de preuves associées à ce modèle sont triviales et il est extrêmement facile de vérifier l'invariant qui n'est constitué que d'une simple propriété de typage.

L'obligation suivante est générée de manière automatique et permet de vérifier la conservation de l'invariant suite à l'exécution de l'événement ToDesync, et est facilement prouvée par un prouveur automatique. Les différentes variables sont renommées afin de pouvoir facilement identifier la valeur de la variable avant et après déclenchement d'un événement.

$$TS\_sync\_loss \in VALEUR \ \wedge \ (\textit{invariant etabli})$$
$$TS\_sync\_loss' \in \{KO, IND\} \ (\textit{predicat avant apres})$$
$$\Rightarrow$$
$$TS\_sync\_loss' \in VALEUR$$

L'obligation de preuve est facilement vérifiée en substituant TS_sync_loss' par sa valeur, KO ou

IND. Il est facile de montrer que $\{KO, IND\} \subseteq VALEUR$. Ce modèle très simple ne génère que très peu d'obligations de preuve (pour la plupart de simple vérification de cohérence du typage) qui sont toutes prouvées de manière automatique par le prouveur de la balbulette lors de leur génération.

2.3 Raffinement de modèles

La notion de raffinement est au coeur de la méthode B événementielle et un développement B est composé d'un modèle abstrait et d'un ou plusieurs raffinements successifs. Les différents pas de raffinement permettent une meilleure compréhension du système ou de déduire une implantation dans le cas d'un système logiciel.

2.3.1 Structure d'un raffinement

Un raffinement est un modèle B un peu particulier qui a la forme suivante :

REFINEMENT
  r
REFINES
  m
SETS
  t
CONSTANTS
  d
PROPERTIES
  $Q(t, d)$
VARIABLES
  y
INVARIANT
  $J(x, y)$
VARIANT
  $V(y)$
ASSERTIONS
  $B(y)$
INITIALISATION
  $y : INIT(y)$
EVENTS
  <liste d'événement>
END

- la clause REFINEMENT contient le nom r du raffinement.
- la clause REFINES contient le nom m du modèle abstrait raffiné.
- les clauses SETS, CONSTANTS, PROPERTIES ont les mêmes fonctions que dans le modèle déjà présenté. Cependant, les ensembles et les constantes définis dans ce modèle concret sont différents de ceux définis dans l'abstraction. Le prédicat $Q(t, d)$ peut cependant lier constantes et ensembles abstraits avec les nouveaux ensembles et constantes définis.
- la clause VARIABLES contient la liste y des variables du modèle.
- la clause INVARIANT contient l'invariant $J(x, y)$ du modèle on appelle cet invariant, *l'invariant de collage* car une partie de ce prédicat lie les variables abstraites x aux variables concrètes y. L'autre partie de l'invariant est constitué de propriétés sur la liste de variables y. Pour des raisons de commodité, nous supposerons que x et y sont des listes distinctes (un mécanisme de renommage permet de traiter les variables communes).
- la clause INITIALISATION est une substitution qui initialise les variables y.

- la clause EVENTS contient les événements du modèle raffiné. Les événements peuvent être de deux natures différentes : les nouveaux événements, qui n'existaient pas dans l'abstraction, et les événements raffinés qui existaient dans l'abstraction (même nom) et dont une version concrète est donnée dans ce nouveau modèle.

2.3.2 Raffinement et preuves en B

Comme défini dans la théorie du raffinement, le modèle B concret est un modèle qui conserve les comportements de son modèle abstrait. L'invariant abstrait défini dans le premier modèle m caractérise également le modèle concret r. Par le raffinement, le modèle concret "hérite" donc des propriétés invariantes de son abstraction (sans refaire de preuves). Cet invariant va permettre de faire des preuves sur le nouveau modèle.

En contrepartie, un certain nombre d'obligations de preuve seront générées afin de prouver la correction du raffinement par rapport à son abstraction. Le raffinement B est un raffinement basé sur les événements : les événements raffinés peuvent changer de forme afin d'enlever de l'indéterminisme aux substitutions et renforcer la garde. Comme le système concret doit avoir un comportement cohérent avec celui de l'abstraction, nous allons devoir montrer qu'une transformation concrète des variables correspond à une transformation abstraite.

Un modèle B événementiel décrivant un système de transitions, la définition du raffinement B est très proche de la définition de raffinement d'un système de transition (présentée page 31). Les obligations de preuves présentées ont pour but de s'assurer que le système concret et le système abstrait (ou plutôt leurs états) sont reliés par une relation de raffinement qui a bien les propriétés souhaitées.

Ainsi, en contraignant plus les versions concrètes des événements, des blocages du système de transition peuvent apparaître. Si ces blocages n'existent pas dans l'abstraction, ils ne doivent pas apparaître dans le raffinement (comportements différents). Une obligation de preuve va s'assurer que de nouveaux blocages n'apparaissent pas durant le raffinement.

Enfin l'apparition de nouveaux événements ne doit pas empêcher les anciens événements de se déclencher. Un variant peut être défini dans la clause VARIANT et permet alors de s'assurer que les nouveaux événements ne prennent pas indéfiniment la main.

Obligations de preuves liées au raffinement

Nous présentons ici les obligations de preuves permettant de s'assurer que les modèles B abstrait et concret sont liés par une relation ayant les propriétés du raffinement souhaité. Les propriétés de la relation de raffinement sont présentées page 31 sur les systèmes de transitions. Les formules suivantes représentent ces obligations de preuves et sont spécifiques à la méthode B événementielle.

En premier lieu, la nouvelle initialisation du modèle doit rester cohérente avec l'initialisation abstraite. Elle doit également satisfaire l'invariant du système. On en déduit donc l'obligation de preuve suivante :

$$INIT(y) \Rightarrow \exists x \cdot (Init(x) \wedge J(x,y)) \qquad \text{(REF1)}$$

On va d'abord considérer un événement raffiné. Soit le prédicat avant-après $BAA(x,x')$ d'un événement abstrait et le prédicat avant-après $BAC(y,y')$ de la version concrète de l'événement. On a alors l'obligation de preuve suivante :

$$I(x) \wedge J(x,y) \wedge BAC(y,y') \Rightarrow \exists x' \cdot (BAA(x,x') \wedge J(x',y')) \qquad \text{(REF2)}$$

Cette obligation traduit le fait qu'un pas concret (l'activation de la version concrète de l'événement) correspond à un pas abstrait comme l'illustre la figure 2.1 déjà présentée page 32. Cette obligation permet de prouver que les comportements concrets sont des comportements abstraits (ou supporté par l'abstraction). L'obligation de preuve permet également de montrer la préservation de l'invariant concret $J(x,y)$.

Les *nouveaux* événements introduits dans le raffinement ne doivent pas modifier les variables de l'abstraction. En effet, ces nouveaux événements doivent raffiner l'événement SKIP, qui est l'événement laissant le système inchangé. Cette obligation de preuve est donc une simplification de la précédente avec comme prédicat avant après abstrait $BAA(x,x') == (x = x')$:

$$I(x) \wedge J(x,y) \wedge BAC(y,y') \Rightarrow J(x,y') \quad \text{(REF3)}$$

Comme dit précédemment, les comportements du système concret sont des comportements du système abstrait. En contraignant plus les événements au fur et à mesure des raffinements, des blocages non désirés pourraient apparaître. Il faut vérifier que le raffinement n'a pas plus de blocages que l'abstraction. A l'aide de la disjonction des gardes abstraites $G_1(x)...G_n(x)$ (et de l'invariant), on doit pouvoir déduire la disjonction des gardes concrètes $H_1(y)...H_m(y)$, montrant ainsi la vivacité du modèle concret :

$$\begin{array}{l} I(x) \wedge J(x,y) \wedge \\ G_1(x) \vee G_2(x) \vee \ldots \vee G_n(x) \Rightarrow \\ H_1(y) \vee H_2(y) \vee \ldots \vee H_m(y) \end{array} \quad \text{(REF4)}$$

Même une fois l'obligation de preuve précédente vérifiée, rien ne garantit encore que le comportement du système concret est cohérent avec son comportement abstrait. Les événements concrets peuvent, en particulier, prendre indéfiniment la main et empêcher les événements abstraits d'avoir lieu. En effet, les nouveaux événements raffinant SKIP, l'absence de déclenchement des événements abstraits est synonyme de blocage du système abstrait. Or le raffinement ne doit pas produire plus de blocage que l'abstraction.

L'introduction du VARIANT empêche ce cas de figure de se produire. En vérifiant que les nouveaux événements font décroître le variant $V(y)$ (un naturel), on a la garantie que les événements abstraits s'exécuteront un jour. Les obligations de preuves suivantes explicitent un peu cette affirmation :

$$I(x) \wedge J(x,y) \Rightarrow V(y) \in \mathbb{N} \quad \text{(REF5)}$$

$$I(x) \wedge J(x,y) \wedge BAC(y,y') \Rightarrow V(y') < V(y) \quad \text{(REF6)}$$

2.3.3 Exemple de raffinement

Le modèle suivant est un raffinement du modèle présenté au paragraphe 2.2.5 page 39. L'en-tête du raffinement est présenté ci-dessous puis les différents événements du modèle sont détaillés.

REFINEMENT
$synchro1$
REFINES
$synchro$
SETS
$BYTE$
CONSTANTS
$SyncByte, bsync, bdesync$
PROPERTIES
$SyncByte \in BYTE \wedge$
$bsync \in \mathbb{N}_1 \wedge$
$bdesync \in \mathbb{N}_1 \wedge$
$bdesync < bsync$
VARIABLES
$TS\_sync\_loss, countOK,$
$countKO, current$
INVARIANT
$countOK \in \mathbb{N} \wedge countKO \in \mathbb{N} \wedge$
$current \in BYTE \wedge$
$(countOK < bsync \Rightarrow TS\_sync\_loss \neq OK) \wedge$
$(TS\_sync\_loss = OK \Rightarrow countKO < bdesync)$
INITIALISATION
$TS\_sync\_loss := IND \parallel$
$countOK := 0 \parallel$
$countKO := 0 \parallel$
$current :\in BYTE$

L'en-tête de ce raffinement introduit des détails qui n'existaient pas dans l'abstraction. En premier lieu, l'ensemble abstrait $BYTE$ qui représente l'ensemble des octets possibles. Le raffinement introduit également des nouveaux détails sur l'environnement et le contexte du système. Une constante, *SyncByte*, représente l'octet de synchronisation (un octet spécifique) tandis que les deux constantes entières *bsync* et *bdesync* représentent le nombre de paquets successifs nécessaire à la synchronisation (et à la désynchronisation) du récepteur.

De nouvelles variables vont permettre d'expliciter le fonctionnement du système : deux compteurs *countOK* et *countKO* permettent de compter le nombre d'octets corrects ou incorrects successifs reçus. Des propriétés invariantes précisent que l'état de synchronisation du récepteur est en liaison avec ces compteurs. Enfin la variable *current* permet de représenter l'octet courant. Dans ce raffinement, la manière d'obtenir cet octet n'est pas encore entièrement explicité et il reste une part d'indéterminisme.

Les événements abstraits du modèle précédent sont raffinés en des versions plus concrètes prenant en compte les nouveaux détails du modèle. Le raffinement présenté n'introduit pas de nouveaux événements et tous les événements du modèle sont des raffinements des deux événements abstraits. L'événement abstrait ToSync est ainsi "séparé" en trois événements concrets distincts qui modélise le système lorsque celui-ci est synchronisé : ToSyncOK, ToSyncKO et ToSyncEnd. L'événement ToSyncEnd modélise l'acquisition de la synchronisation alors que le couple d'événement ToSyncOK et ToSyncKO représente le comportement du système lorsque celui ci est synchronisé. Par ailleurs l'événement abstrait ToDesync est également raffiné en trois événements concrets distincts : ToDesyncOK, ToDesyncKO et ToDesyncEnd.

Les deux événements ToSyncEnd et ToDesyncEnd raffinent respectivement ToSync et ToDesync mais détaillent plus précisemment le changement d'état de synchronisation :

- le récepteur n'est plus synchronisé quand il a reçu *bdesync* mauvais octets de synchronisation (càd $countKO + 1 = bdesync$).
- le récepteur devient synchronisé quand il a reçu *bsync* octets de synchronisation (càd $countOK + 1 = bsync$).

Les deux événements raffinés représentent des états "transitoires" permettant le passage d'un système synchronisé à un système désynchronisé et vice versa.

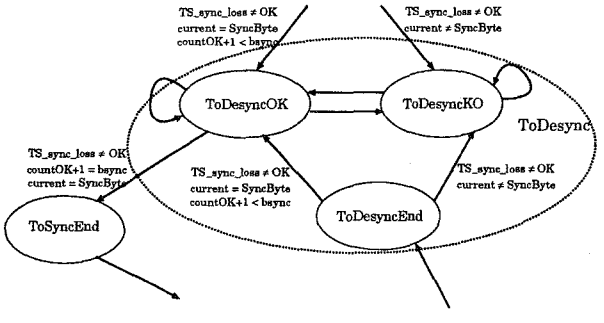
ToDesyncEnd $\widehat{=}$
SELECT
$TS\_sync\_loss = OK \wedge$
$current \neq SyncByte \wedge$
$countKO + 1 = bdesync$
THEN
$TS\_sync\_loss := KO \parallel$
$countKO := countKO + 1 \parallel$
$countOK := 0 \parallel$
$current :\in BYTE$
END

ToSyncEnd $\widehat{=}$
SELECT
$TS\_sync\_loss \neq OK \wedge$
$current = SyncByte \wedge$
$countOK + 1 = bsync$
THEN
$TS\_sync\_loss := OK \parallel$
$countOK := countOK + 1 \parallel$
$countKO := 0 \parallel$
$current :\in BYTE$
END

Les deux événements ci-dessous explicitent le comportement du système lorsque celui ci n'est pas synchronisé. On suppose le système dans un état non synchronisé (càd $TS\_sync\_loss \neq OK$). Lorsqu'un octet de synchronisation est détecté le compteur *countOK* est incrémenté (événement ToDesyncOK). Par ailleurs, si un mauvais octet est détecté, alors le compteur *countOK* est réinitialisé (événement ToDesyncKO). La recherche de la synchronisation continue ainsi jusqu'à ce que le compteur *countOK* ait atteint la valeur *bsync* qui permet le déclenchement de l'événement ToSyncEnd signifiant que la synchronisation est acquise et représentant la transition entre les deux états de synchronisation. Dans tous les cas, l'octet courant a été considéré et le "calcul" doit se poursuivre sur un nouvel octet ($current :\in BYTE$). Le schéma suivant les deux événements ToDesyncOK et ToDesyncKO est incomplet mais illustre ce comportement.

ToDesyncOK $\widehat{=}$
SELECT
$TS\_sync\_loss \neq OK \wedge$
$current = SyncByte \wedge$
$countOK + 1 < bsync$
THEN
$countOK := countOK + 1 \parallel$
$current :\in BYTE$
END

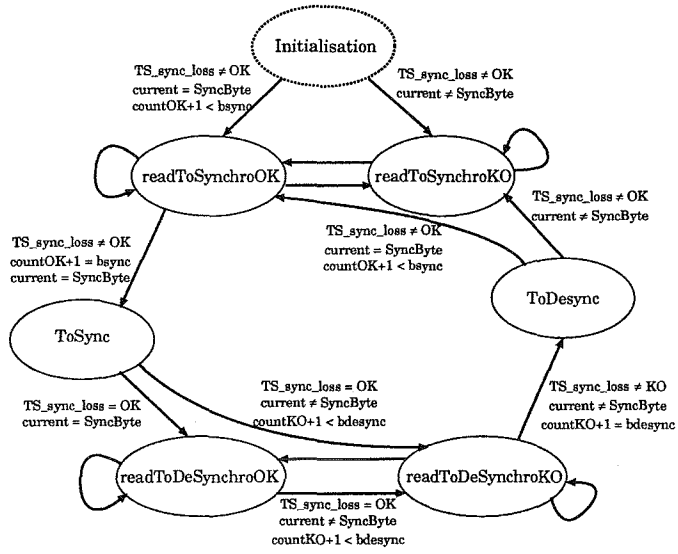
ToDesyncKO $\widehat{=}$
SELECT
$TS\_sync\_loss \neq OK \wedge$
$current \neq SyncByte$
THEN
$countOK := 0 \parallel$
$current :\in BYTE$
END



Les deux événements ci-dessous sont duales aux événements précédents et représentent le fonctionnement du récepteur synchronisé. Ils détaillent le fonctionnement du système lorsqu'il est synchronisé. C'est cette fois-ci le compteur *countKO* qui a de l'importance et qui permet de déterminer la perte ou la conservation de la synchronisation. Le comportement de ces événements est tout à fait similaire aux précédents : la perte de la synchronisation a lieu lorsque le compteur *countKO* a atteint la valeur *bdesync* (déclenchement de l'événement ToDesyncEnd). On obtient alors le système de transition présenté un peu plus loin qui illustre le comportement du système.

ToSyncOK $\widehat{=}$
SELECT
$TS\_sync\_loss = OK \wedge$
$current = SyncByte$
THEN
$countOK := countOK + 1 \parallel$
$countKO := 0 \parallel$
$current :\in BYTE$
END

ToSyncKO $\widehat{=}$
SELECT
$TS\_sync\_loss = OK \wedge$
$current \neq SyncByte \wedge$
$countKO + 1 < bdesync$
THEN
$countKO := countKO + 1 \parallel$
$current :\in BYTE$
END



Le raffinement présenté permet de détailler le comportement du système. A ce niveau de raffinement, il n'est pas encore possible de déterminer clairement un algorithme même si ce premier raffinement supprime de l'indéterminisme.

Les obligations de preuves de raffinement sont générées automatiquement et ont pour but de montrer que le raffinement proposé est correct et préserve bien les comportements abstraits. L'obligation de preuve suivante doit vérifier que la version concrète de l'événement ToDesyncEnd est cohérente avec sa version abstraite ToDesync :

$TS\_sync\_loss \in VALEUR \wedge \quad (Invariant\ Abstrait)$
$(... \wedge (countOK < bsync \Rightarrow TS\_sync\_loss_1 \neq OK) \wedge$
$(TS\_sync\_loss_1 = OK \Rightarrow countKO < bdesync) \wedge$
$TS\_sync\_loss = TS\_sync\_loss_1) \wedge \quad (Invariant\ de\ Collage)$
$(TS\_sync\_loss_1 = OK \wedge current \neq SyncByte \wedge countKO + 1 = bdesync \wedge$
$TS\_sync\_loss'_1 = KO \wedge countKO' = countKO + 1 \wedge countOK' = 0) \quad (Predicat\ BAC)$
$\Rightarrow$
$\exists\, TS\_sync\_loss'.(TS\_sync\_loss' \in \{KO, IND\} \wedge \quad (Predicat\ BAA)$
$(... \wedge (countOK' < bsync \Rightarrow TS\_sync\_loss'_1 \neq OK) \wedge$
$(TS\_sync\_loss'_1 = OK \Rightarrow countKO' < bdesync) \wedge$
$TS\_sync\_loss' = TS\_sync\_loss'_1)) \quad (Invariant\ de\ Collage)$

L'obligation ci-dessus ne présente pas les propriétés de typage de l'invariant du raffinement $J(x, y)$, celles-ci n'étant pas utiles à la preuve. Cette obligation de preuve est simplifiée (automatiquement) en une obligation plus simple. En effet, la propriété $TS\_sync\_loss' = TS\_sync\_loss'_1$ permet d'éliminer la quantification existentielle. L'outil utilisé est d'ailleurs capable de faire automatiquement ce genre de simplifications lors de la génération de telles obligations de preuves.

$$
\begin{array}{l}
TS\_sync\_loss \in VALEUR \ \wedge \quad (Invariant\ Abstrait) \\
(... \ \wedge \ (countOK < bsync \ \Rightarrow \ TS\_sync\_loss_1 \neq OK) \ \wedge \\
(TS\_sync\_loss_1 = OK \ \Rightarrow \ countKO < bdesync) \ \wedge \\
TS\_sync\_loss = TS\_sync\_loss_1) \ \wedge \quad (Invariant\ de\ Collage) \\
(TS\_sync\_loss_1 = OK \ \wedge \ current \neq SyncByte \ \wedge \ countKO + 1 = bdesync \ \wedge \\
TS\_sync\_loss'_1 = KO \ \wedge \ countKO' = countKO + 1 \ \wedge \ countOK' = 0) \quad (Predicat\ BAC) \\
\Rightarrow \\
TS\_sync\_loss'_1 \in \{KO, IND\} \ \wedge \quad (Predicat\ BAA) \\
(... \ \wedge \ (countOK' < bsync \ \Rightarrow \ TS\_sync\_loss'_1 \neq OK) \ \wedge \\
(TS\_sync\_loss'_1 = OK \ \Rightarrow \ countKO' < bdesync)) \quad (Invariant\ de\ Collage)
\end{array}
$$

Qui se simplifie finalement en l'obligation de preuve suivante qui est facilement démontrable :

$$
\begin{array}{l}
TS\_sync\_loss \in VALEUR \ \wedge \quad (Invariant\ Abstrait) \\
(... \ \wedge \ (countOK < bsync \ \Rightarrow \ TS\_sync\_loss_1 \neq OK) \ \wedge \\
(TS\_sync\_loss_1 = OK \ \Rightarrow \ countKO < bdesync) \ \wedge \\
TS\_sync\_loss = TS\_sync\_loss_1) \ \wedge \quad (Invariant\ de\ Collage) \\
(TS\_sync\_loss_1 = OK \ \wedge \ current \neq SyncByte \ \wedge \ countKO + 1 = bdesync \ \wedge \\
TS\_sync\_loss'_1 = KO \ \wedge \ countKO' = countKO + 1 \ \wedge \ countOK' = 0) \quad (Predicat\ BAC) \\
\Rightarrow \\
KO \in \{KO, IND\} \ \wedge \quad (Predicat\ BAA) \\
(... \ \wedge \ (0 < bsync \ \Rightarrow \ KO \neq OK) \ \wedge \\
(KO = OK \ \Rightarrow \ countKO + 1 < bdesync)) \quad (Invariant\ de\ Collage)
\end{array}
$$

Le raffinement proposé n'introduit pas de nouveaux événements mais sépare les événements abstraits en plusieurs événements raffinés détaillant le comportement du système. Ce type de raffinement est spécifique du B événementiel et la réalisation de ce raffinement dans l'Atelier B n'est pas problématique : les événements concrets issus de la séparation sont en réalité déjà présents dans le modèle abstrait mais ont tous le même corps.

Le raffinement présenté produit un ensemble de 14 obligations de preuve (non triviales) dont seules 2 ne sont pas déchargées par le prouveur automatique de l'outil. Ces 2 obligations de preuves sont des obligations de preuves de raffinement qui permettent de vérifier la cohérence de comportements des versions abstraites et concrètes des événements *ToSyncEnd* et *ToDesyncEnd*. La preuve de ces deux obligations se fait assez simplement par quelques interactions avec le prouveur intéractif de l'outil grâce à l'invariant du raffinement.

2.3.4 Détection d'erreurs de spécification par la preuve

L'intérêt de la méthode B est de permettre la dérivation de programmes corrects à l'aide du raffinement. Les obligations de preuve générées permettent de détecter les erreurs de spécification lorsqu'elles ne sont pas prouvables. Le raffinement précédent est un raffinement correct du modèle présenté page 39 mais la construction d'un modèle concret correct se fait généralement par des corrections déduites des obligations de preuves. Celles-ci sont alors utilisées comme un debugger permettant de souligner les incohérences entre les deux descriptions. Nous allons rapidement présenter des exemples d'incohérences facilement détectées par la preuve.

Supposons le raffinement *Erronee*, très proche du raffinement correct présenté. Dans l'en-tête, seule l'initialisation est différente : $TS\_sync\_loss :\in VALEUR$.

INITIALISATION
$TS\_sync\_loss :\in VALEUR \parallel$
$countOK := 0 \parallel$
$countKO := 0 \parallel$
$current :\in BYTE$

Cette modification de l'initialisation d'une variable peut sembler sans importance mais les obligations de preuves liées au raffinement vont montrer que ce changement rend le raffinement incorrect. L'obligation de preuve REF1 générée est présentée ci dessous :

$TS\_sync\_loss_1 \in VALEUR \wedge countOK = 0 \wedge countKO = 0 \wedge$
$current \in BYTE$ *(Initialisation concrete)*
$\Rightarrow$
$\exists TS\_sync\_loss'.(TS\_sync\_loss' = IND \wedge$ *(Initialisation abstraite)*
$(... \wedge (countOK < bsync \Rightarrow TS\_sync\_loss_1 \neq OK) \wedge$
$(TS\_sync\_loss_1 = OK \Rightarrow countKO < bdesync) \wedge$
$TS\_sync\_loss' = TS\_sync\_loss_1)$ *(Invariant de Collage)*
)

L'obligation donnée ici ne présente pas les propriétés de typage de l'invariant de raffinement $J(x, y)$. Une fois encore, cette obligation de preuve est simplifiée automatiquement en une obligation de preuve plus simple grâce à l'égalité $TS\_sync\_loss' = TS\_sync\_loss_1$. On doit alors prouver l'obligation de preuve :

$TS\_sync\_loss_1 \in VALEUR \wedge countOK = 0 \wedge countKO = 0 \wedge current \in BYTE$
$\Rightarrow$
$TS\_sync\_loss_1 = IND \wedge$
$(... \wedge (countOK < bsync \Rightarrow TS\_sync\_loss_1 \neq OK) \wedge$
$(TS\_sync\_loss_1 = OK \Rightarrow countKO < bdesync)$ *(Invariant de Collage)*

Prouver que l'initialisation des variables concrètes qui n'existaient pas dans l'abstraction respecte l'initialisation abstraite n'est pas un problème et l'on peut finalement résumer l'obligation de preuve produite en une obligation très simple mais qui ne peut être prouvée. Il est en effet impossible de déduire l'égalité $TS\_sync\_loss_1 = IND$ à l'aide de la simple propriété de typage $TS\_sync\_loss_1 \in VALEUR$, l'ensemble $VALEUR$ n'étant pas le singleton $\{IND\}$.

$$TS\_sync\_loss_1 \in VALEUR \ \wedge \ ... \ \wedge \ current \in BYTE$$
$$\Rightarrow$$
$$TS\_sync\_loss_1 = IND \ \wedge \ ...$$

Cette obligation de preuve montre bien que l'état initial du modèle concret n'est pas cohérent avec celui de son abstraction. En effet, la nouvelle initialisation est plus "souple" que l'ancienne et ne permet pas de conserver les comportements du modèle abstrait.

Un autre exemple de debuggage à l'aide des obligations de preuve peut être montré avec les nouveaux événements du système. Supposons que l'événement ToSyncKO de ce modèle soit :

ToSyncKO $\widehat{=}$
SELECT
$TS\_sync\_loss = OK \ \wedge$
$current \neq SyncByte \ \wedge$
$countKO < bdesync$
THEN
$countKO := countKO + 1 \ \|$
$current :\in BYTE$
END

Cette événement est très similaire à sa version correcte et c'est la propriété $countOK < bdesync$ qui diffère. Cette petite erreur de modélisation est détectable lors de la preuve des obligations associées au modèle. L'obligation de preuve INV2 assurant que l'activation d'un événement préserve bien l'invariant générée pour ce nouvel événement ToSyncKO est alors :

$$... \ \wedge \ (countOK < bsync \ \Rightarrow \ TS\_sync\_loss_1 \neq OK) \ \wedge$$
$$(TS\_sync\_loss_1 = OK \ \Rightarrow \ countKO < bdesync) \ \wedge \quad (Invariant)$$
$$TS\_sync\_loss_1 = OK \ \wedge \ current \neq SyncByte \ \wedge$$
$$countKO < bdesync \wedge \quad (Predicat\ avant)$$
$$countKO' = countKO + 1 \quad (Predicat\ apres)$$
$$\Rightarrow$$
$$... \ \wedge \ (countOK' < bsync \ \Rightarrow \ TS\_sync\_loss_1 \neq OK) \ \wedge$$
$$(TS\_sync\_loss_1 = OK \ \Rightarrow \ countKO' < bdesync) \quad (Invariant)$$

A l'aide du prédicat Avant-Après qui lie les nouvelles valeurs des variables aux anciennes, cette obligation de preuve est simplifiée : la variable $countKO'$ est remplacée par sa valeur. L'activation de l'événement ToSyncKO viole l'invariant du modèle et rend l'obligation improuvable :

$$... \ \wedge \ TS\_sync\_loss_1 = OK \ \wedge \ current \neq SyncByte \ \wedge$$
$$countKO < bdesync \ \wedge \ b = TRUE \ \wedge \quad (Predicat\ avant)$$
$$\Rightarrow$$
$$... \ \wedge \ (TS\_sync\_loss_1 = OK \ \Rightarrow \ countKO + 1 < bdesync)$$

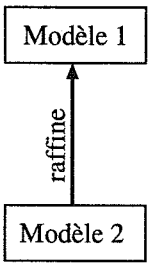


FIG. 2.4 – Exemple de projet B

La variable $TS\_sync\_loss_1$ est égale à OK d'après le prédicat Avant-Après de l'événement (voir garde de l'événement). Cependant il est impossible de prouver l'infériorité stricte entre l'expression $countKO+1$ et la constante *bdesync*. Cette propriété de l'invariant n'est pas préservée et l'impossibilité de la preuve indique donc une erreur dans la spécification que l'on peut facilement déduire (la propriété $countOK <$ *bsync* dans la garde de l'événement n'est pas assez forte pour préserver l'invariant). Cette exemple illustre bien l'importance de la pertinence de l'invariant qui ne doit pas être trop faible pour pouvoir permettre la preuve des propriétés du système.

2.4 Instanciation et paramétrisation

Un projet est un ensemble de modèles liés par la relation de raffinement et composant un développement formel. La réutilisation de développement est un point crucial pour le passage à l'échelle des méthodes formelles. Si un développement est suffisamment générique et qu'il englobe un problème plus particulier, on peut vouloir réutiliser ce développement et, de ce fait, éviter de refaire les preuves de cohérence et de raffinement des différents modèles.

2.4.1 Projet : paramètres formels

Un projet P est une liste de modèles liés par la relation de raffinement. La partie "statique" d'un modèle est constituée des ensembles et des constantes. Les ensembles génériques et les constantes représentent le *contexte* d'un modèle. Ces différents objets permettent de décrire l'environnement du système (relation sur les données, etc...). Les ensembles et les constantes utilisés dans un projet sont des *paramètres formels* des développements. La valeur de ces paramètres peut être définie pour instancier le projet pour un développement particulier. La figure 2.4 présente un exemple de projet P_1 composé de deux modèles liés par la relation de raffinement.

2.4.2 Instanciation d'un projet

Le but de l'instanciation d'un projet est la réutilisation de développements déjà prouvés sans devoir ré-écrire et reprouver les différents raffinements déjà faits. Intuitivement, l'instanciation va demander certaines vérifications sur les *paramètres effectifs* d'instanciation. Il faudra, en particulier, s'assurer que les paramètres effectifs ont les mêmes propriétés que les paramètres formels. Une fois cette vérification terminée, l'ensemble du projet est instancié pour un sous-problème spécifique.

Supposons un projet P_2 constitué d'un certain nombre de modèles liés par la relation de raffinement. La figure 2.5 présente un tel projet. Supposons que les modèles $Dvpt2$ et $Dvpt3$ du projet P_2 soient un cas particulier des modèles $Modele1$ et $Modele2$ du projet P_1. On peut souhaiter incorporer à P_2 les développements réalisés dans le cadre du projet P_1. Une première étape est d'instancier le projet P_1 pour obtenir un projet P_1' adapté. Cette instanciation se fait en donnant les valeurs utilisées dans le contexte du projet P_2 aux paramètres formels du projet P_1 (càd ses ensembles génériques et ses constantes). Les variables et les événements de P_1 peuvent également être renommés.

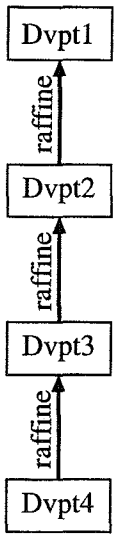


FIG. 2.5 – Un autre projet B

Sous couvert de preuves d'instanciations[7], on dispose de modèles instanciés issus du projet P_1 et utilisés pour le cas particulier traité dans le projet P_2. La figure 2.6 résume la situation et les principes de l'instanciation.

2.5 Dérivation d'algorithmes

Jean-Raymond Abrial décrit dans [6] un certain nombre de règles permettant de produire un algorithme à partir d'un modèle (suffisamment) raffiné. Intuitivement, il est clair que le modèle doit être suffisamment explicite et ne pas contenir d'indéterminisme pour pouvoir produire un algorithme complet.

2.5.1 Règles de recomposition

Abrial propose essentiellement deux règles de reconstruction d'algorithme :
- la première permet la construction d'une instruction conditionnelle.
- la seconde définit une instruction de boucle.

On a d'abord la règle suivante :

SELECT $P \wedge Q$ THEN S END
SELECT $P \wedge \neg Q$ THEN T END
$\rightsquigarrow$
SELECT P THEN
  IF Q THEN S ELSE T END
END

qui permet l'introduction d'une conditionnelle en recomposant deux événements gardés.
La deuxième règle est :

[7]qui consiste à vérifier que les paramètres effectifs respectent les propriétés des paramètres formels.

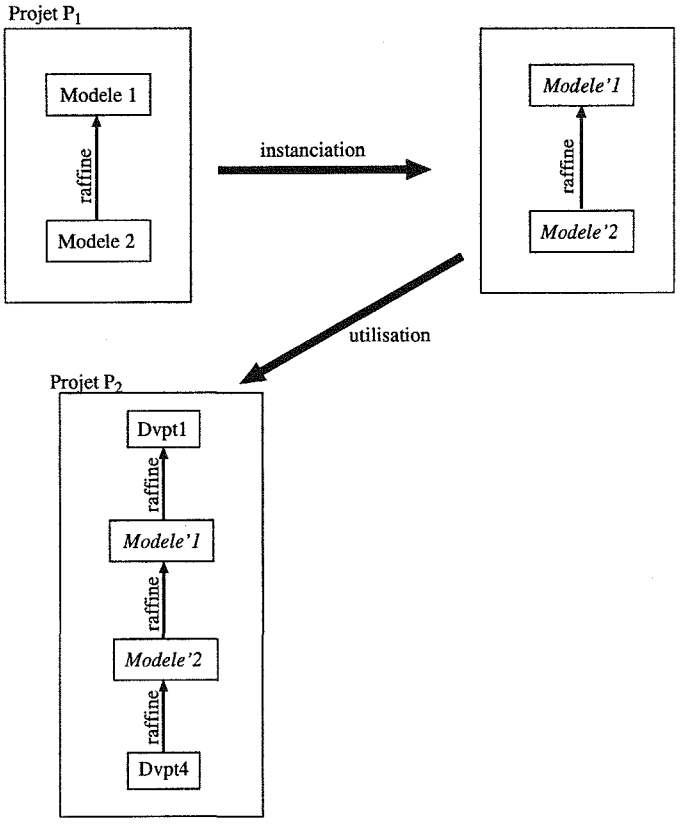


FIG. 2.6 – Réutilisation de projet

SELECT $P \wedge Q$ THEN S END
SELECT $P \wedge \neg Q$ THEN T END
$\rightsquigarrow$
SELECT P THEN
  WHILE Q DO S END; T
END

Les deux règles, bien qu'utilisant les mêmes types d'événements, ne sont pas contradictoires. En effet, des conditions d'applications accompagnent les deux règles :

- la deuxième règle nécessite que l'événement constituant le coeur de la boucle apparaisse *au moins un raffinement après le second événement.* De ce fait, nous sommes sûrs de la terminaison de la boucle car un variant a permis de montrer que les nouveaux événements ne prennent pas indéfiniment la main.
- la première règle est applicable lorsque la seconde ne l'est pas.

En appliquant "récursivement" ces deux règles, les événements du modèle sont recomposés jusqu'à l'obtention d'un unique événement dont la substitution est l'algorithme décrit par le modèle.

2.5.2 Exemple : raffinement vers une implantation

Pour illustrer l'utilisation de ces règles de production d'un algorithme nous allons raffiner le modèle présenté au paragraphe 2.3.3. Ce modèle explicitait l'algorithme de synchronisation d'un récepteur mais un certain indéterminisme était encore présent. Nous proposons donc le raffinement suivant, qui supprime les dernières traces d'indéterminisme. Les événements du modèle sont très peu changés : les gardes des événements sont légèrement modifiées afin de considérer l'ajout des nouvelles variables au modèle.

REFINEMENT
  $synchro2$
REFINES
  $synchro1$
CONSTANTS
  $Flux, TSsize$
PROPERTIES
  $Flux \in \mathbb{N} \rightarrow BYTE \wedge$
  $TSsize \in \mathbb{N}_1$
VARIABLES
  $TS\_sync\_loss, countOK, countKO, pointer$
INVARIANT
  $pointer \in \mathbb{N} \wedge$
  $current = Flux(pointer)$
INITIALISATION
  $TS\_sync\_loss := IND \parallel$
  $countOK := 0 \parallel$
  $countKO := 0 \parallel$
  $pointer := 0$

Ce dernier raffinement introduit de nouvelles informations sous la forme de constantes et de variables. Un fonction constante $Flux$ représentant le séquencement des octets est introduite ainsi qu'une constante $TSsize$, un naturel représentant le nombre d'octets qui compose un paquet. $TSsize$ est donc le nombre d'octet qui sépare normalement deux octets de synchronisation consécutifs. Enfin une variable *pointer* représente l'évolution de la réception des octets. La variable abstraite *current* n'existe plus dans ce modèle concret et un invariant de collage permet de lier la variable concrète *pointer* à celle-ci.

Les événements ci-dessous sont très semblables à ceux présentés dans le modèle précédent. Il n'y a plus aucune trace d'indéterminisme et un algorithme peut donc être produit à partir de ce modèle.

ToDesync $\widehat{=}$
SELECT
$TS\_sync\_loss = OK \wedge$
$Flux(pointer) \neq SyncByte \wedge$
$countKO + 1 = bdesync$
THEN
$TS\_sync\_loss := KO \parallel$
$countKO := countKO + 1 \parallel$
$countOK := 0 \parallel$
$pointer := pointer + 1$
END

ToSync $\widehat{=}$
SELECT
$TS\_sync\_loss \neq OK \wedge$
$Flux(pointer) = SyncByte \wedge$
$countOK + 1 = bsync$
THEN
$TS\_sync\_loss := OK \parallel$
$countOK := countOK + 1 \parallel$
$countKO := 0 \parallel$
$pointer := pointer + TSsize$
END

ToDesyncOK $\widehat{=}$
SELECT
$TS\_sync\_loss \neq OK \wedge$
$Flux(pointer) = SyncByte \wedge$
$countOK + 1 < bsync$
THEN
$countOK := countOK + 1 \parallel$
$pointer := pointer + TSsize$
END

ToDesyncKO $\widehat{=}$
SELECT
$TS\_sync\_loss \neq OK \wedge$
$Flux(pointer) \neq SyncByte$
THEN
$countOK := 0 \parallel$
$pointer := pointer + 1$
END

readToDeSynchroOK $\widehat{=}$
SELECT
$TS\_sync\_loss = OK \wedge$
$Flux(pointer) = SyncByte$
THEN
$countOK := countOK + 1 \parallel$
$countKO := 0 \parallel$
$pointer := pointer + TSsize$
END

ToSyncKO $\widehat{=}$
SELECT
$TS\_sync\_loss = OK \wedge$
$Flux(pointer) \neq SyncByte \wedge$
$countKO + 1 < bdesync$
THEN
$countKO := countKO + 1 \parallel$
$pointer := pointer + TSsize$
END

La légère modification des événements abstraits (dûe à l'apparition de la variable concrète *pointer*) produit un ensemble de 25 obligations de preuve toutes automatiquement prouvées. De ce modèle B événementiel concret, complètement déterministe, on peut déduire un algorithme par l'application des règles citées précédemment. Les règles de recomposition produisent des événements comme, par exemple, l'événement readSync résultat de la recomposition des deux événements ToSync et ToDesyncOK. Les deux événements considérés ont été introduits dans le même niveau de raffinement du modèle.

SELECT
$TS\_sync\_loss \neq OK \wedge$
$Flux(pointer) = SyncByte \wedge$
$countOK + 1 = bsync$
THEN
$TS\_sync\_loss := OK \parallel$
$countOK := countOK + 1 \parallel$
$countKO := 0 \parallel$
END

SELECT
$TS\_sync\_loss \neq OK \wedge$
$Flux(pointer) = SyncByte \wedge$
$countOK + 1 < bsync$
THEN
$countOK := countOK + 1 \parallel$
END

```
⇝
SELECT
      TS_sync_loss ≠ OK ∧
      Flux(pointer) = SyncByte ∧
  IF countOK + 1 = bsync THEN
      TS_sync_loss := OK ||
      pointer := pointer + TSsize
  ELSE
      countOK := countOK + 1 ||
      pointer := pointer + TSsize
  END
END
```

Les nouveaux événements recomposés ainsi sont à nouveau inspectés afin d'être regroupés dans un unique événement. L'événement recomposé prog obtenu est alors l'événement suivant :

```
INITILISATION
  TS_sync_loss := IND ||
  countOK := 0 ||
  countKO := 0 ||
  pointer := 0 ||
EVENTS
prog ≙ BEGIN
  IF TS_Sync_loss ≠ OK THEN
    IF Flux(pointer) ≠ SyncByte THEN
      countKO := countKO + 1; //evenement ToDesyncKO
      countOK := 0;
      pointer := pointer + 1;
    ELSEIF countOK + 1 = bsync THEN
      TS_Sync_loss := OK; //evenement ToSync
      countOK := countOK + 1;
      countKO := 0;
      pointer := pointer + TSsize;
    ELSE
      countOK := countOK + 1; //evenement ToDesyncOK
      pointer := pointer + TSsize;
    END
  ELSE
    IF Flux(pointer) = SyncByte THEN
      countOK := countOK + 1; //evenement readToDeSynchroOK
      countKO := 0;
      pointer := pointer + TSsize;
    ELSEIF countKO + 1 = bdesync THEN
      TS_Sync_loss := KO; //evenement ToDesync
      countKO := countKO + 1;
      countOK := 0;
      pointer := pointer + 1;
    ELSE
      countKO := countKO + 1; //evenement ToSyncKO
      pointer := pointer + TSsize;
    END
  END
END
```

L'algorithme obtenu est finalement un événement non gardé permettant de simuler l'arrivée continue d'octets en provenance du flux. L'algorithme calcule l'état de synchronisation en analysant l'octet courant en fonction de l'état du système. Les événements concrets du dernier modèle sont aisément détectables dans cet algorithme qui n'est pas optimisé.

2.6 Conclusions

La méthode B a l'énorme avantage d'être outillée [47] contrairement à un certain nombre d'autres méthodes formelles. En effet, la génération des obligations de preuves est un processus largement automatisé [48]. Un prouveur automatique a pour rôle de décharger les obligations de preuves ainsi produites. Les obligations de preuves non prouvées automatiquement sont alors démontrées de manière interactive à l'aide d'une interface graphique ad-hoc [8]. Les outils proposés sont actuellement en retard sur les dernières évolutions du B. En effet, l'Atelier B a été défini pour le B classique et l'utilisation de celui-ci nécessite une traduction des modèles B événementiels en des modèles B classiques. De plus, les outils de génération de code automatique ne sont plus disponibles. Cependant, il est possible de générer les preuves de variance grâce à un mécanisme d'anticipation et la disjonction des gardes des événements peut être prouvée dans les modèles en assertions [42]. L'outil actuel reste un bon support au développement de modèles B événementiels.

Les différents modèles présentés dans ce chapitre et dans le reste de ce document sont le résultat d'une réflexion et d'un processus de modélisation humain relativement peu automatisable. L'intérêt de la méthode B est de proposer à son utilisateur des gardes fous (les obligations de preuve) lui permettant de s'assurer de la cohérence interne de ses descriptions mais aussi de la correction du processus de raffinement. Des erreurs peuvent apparaîtrent aussi bien au niveau de la cohérence des modèles qu'entre deux raffinements mais les obligations de preuve permettent généralement la détection de celles-ci : une obligation de preuve improuvable est signe d'une erreur de modélisation ou d'un invariant incomplet. Le concepteur a alors la charge de trouver un invariant plus complet ou système. Il est cependant aidé par la ou les obligations de preuve non prouvées qui exhibent généralement les propriétés manquantes du modèle.

La méthode B a très tôt été utilisée pour la modélisation de vastes applications (METEOR) et a montré sa robustesse et son utilité dans des véritables projets et pas seulement des exemples-jouets. Le raffinement semble être un outil indispensable dans la conception de système de taille réelle. En effet, d'un point de vue pédagogique, il permet de mieux expliquer et mieux comprendre les algorithmes ou les comportements d'un système. Du point de vue formel, il assure une cohérence entre spécifications et implantations en assurant la préservation des comportements. Enfin, du fait de la transitivité de la relation de raffinement, la complexité d'un système est distribuée dans les différents niveaux d'abstraction. De ce fait, la méthode B nous semble un outil approprié pour une modélisation de systèmes embarqués. En nous appuyant sur la notion de raffinement B nous pourrons formaliser le lien entre les différentes descriptions architecturales utilisées dans le cadre de la conception de SoC.

Chapitre 3

Etude de cas : premières approches

Dans ce chapitre, nous allons présenter en détail l'étude de cas du projet RNRT EQUAST. Dans un premier temps, nous résumerons les grands principes de métrologie et de l'évaluation de qualité de la télévision numérique terrestre. Nous présenterons ensuite nos premiers développements formels dans le cadre du projet qui avait pour objectif de fournir un modèle du système utile aux travaux des concepteurs électroniques.

Ce premier travail a permis d'établir le dialogue entre les partenaires du projet et de nous faire une idée des problèmes de conception électronique. Nous avons acquis une meilleure maîtrise du domaine et des problèmes techniques (efficacité, temps réel). Les différents développements présentés dans cette partie ont servi de support à un grand nombre de débats. En effet, les modèles formels proposés ont permis d'éliminer les problèmes de compréhension (en particulier dûs à un vocabulaire technique) par la définition de principes mathématiques clairs. Chaque partenaire était alors en mesure d'apporter ses connaissances aux modèles en affirmant ou infirmant celui-ci.

3.1 Etude de cas : La DVB-T

Les paramètres d'évaluation de la norme TR 101 290 [55] sont hiérarchisés en trois niveaux selon leur ordre d'apparition chronologique et leur importance dans l'analyse de la qualité de service. Nous allons détailler les intérêts et les principes de certains paramètres que nous considérons comme importants. L'idée n'est pas de reprendre intégralement la norme TR mais de permettre au lecteur de se faire une idée relativement précise du type de vérifications préconisées.

3.1.1 L'information encapsulée

La transmission du signal se fait par l'envoi de données MPEG sous forme de paquets de taille constante et horodatés. Nous nommerons, par la suite, ces paquets : paquets TS (*Transport Stream*). Tous les programmes sont ainsi émis sur le même canal de communication, les paquets des différents programmes se mélangeant. Un programme est découpé en plusieurs composantes telles la vidéo, l'audio, le télétexte, etc... Le récepteur, pour pouvoir reconstruire correctement le signal, a besoin d'informations "système". Ces informations sont contenues dans des paquets spécifiques ainsi que dans l'en-tête de chaque paquet. Nous présenterons ces paquets particuliers ultérieurement (paragraphe suivant). L'en-tête de chaque paquet contient, outre des informations de synchronisation, un numéro identifiant, le **PID** (*Packet IDentifier*), qui est propre à un type d'information. Tous les paquets d'une même composante d'un programme sont ainsi estampillés par ce numéro. L'ordre des paquets ayant une importance, l'en-tête contient également un compteur, **Continuity_counter**, permettant de numéroter les paquets d'une même source. Lors de l'émission, les paquets TS peuvent être dupliqués et un même paquet peut, par conséquent, être reçu plusieurs fois. Le schéma 3.1 présente la constitution précise de l'en-tête d'un paquet.

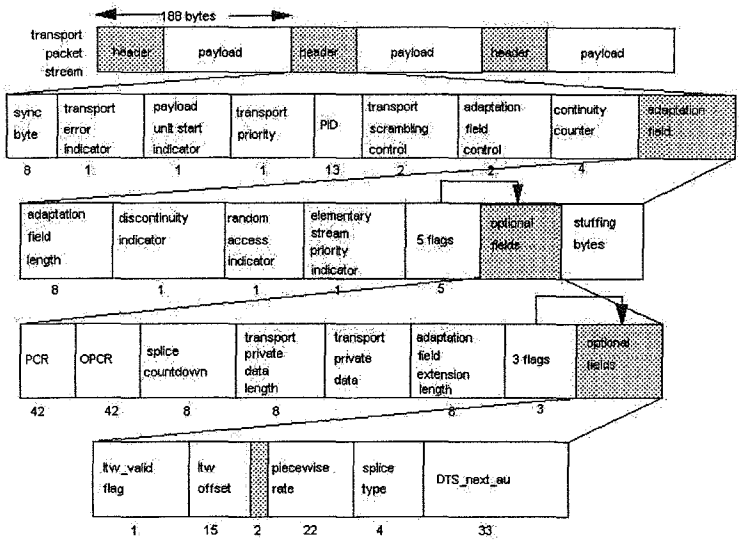


FIG. 3.1 – Détails de l'en-tête d'un paquet TS

3.1.2 La gestion du système

Les données de tous les programmes étant encapsulées et émises sur le même média, les informations permettant de reconstruire le signal correctement en bout de chaîne doivent être détectables. Les paquets "système" permettent de véhiculer des tables (annuaires) afin d'identifier facilement l'appartenance d'un paquet à telle ou telle composante d'un programme. Ces tables sont simplement constituées de la liste des PID existant dans le système et l'appartenance des PID à tel ou tel programme. En effet, un PID est attribué par type de paquet d'un programme : vidéo, audio, télétexte... Ces paquets particuliers permettant le contrôle du système sont nommés **PAT** et **PMT**. La table PAT référence les PID des paquets PMT. Elle est donc indispensable à la reconstruction des tables PMT. La table PMT est simplement la liste des PID spécifiques à un programme. Il y a donc autant de tables que de programmes (cf Figure 3.2).

Certains paramètres sont chargés de vérifier la cohérence des PID des paquets reçus et leur régularité. Ainsi le paramètre *PAT_error* vérifie que le flux est constitué régulièrement de tables PAT pour permettre une remise à jour des récepteurs. Le paramètre *PMT_error*, qui vérifie la régularité de réception des paquets PMT, remplit la même fonction. Un autre paramètre, *Unreferenced_PID*, vérifie que les PID des paquets reçus sont correctement référencés. Dans le cas contraire une alarme indique la réception d'un PID non référencé. Enfin, le paramètre *PID_error* est chargé de vérifier la réelle utilisation des PID référencés dans les tables : la norme impose que chaque PID référencé dans une table PMT soit effectivement utilisé.

En terme d'évaluation de qualité de service, la présence et la cohérence des tables est très importante. En effet, l'apparition d'erreurs dans ces tables entraîne très rapidement une dégradation de la qualité de l'image, le récepteur étant incapable de trier les différents paquets reçus. Ainsi, les tables PAT et PMT doivent être reçues très régulièrement pour permettre un suivi précis des changements éventuels de PID dans les programmes. La norme impose donc une réception de paquet PAT chaque demi-seconde afin de permettre aux récepteurs de se (re)mettre à jour. D'autres paramètres sont chargés de vérifier la bonne construction des paquets PAT et PMT et, en particulier, de vérifier la cohérence de l'en-tête de ce type de paquet.

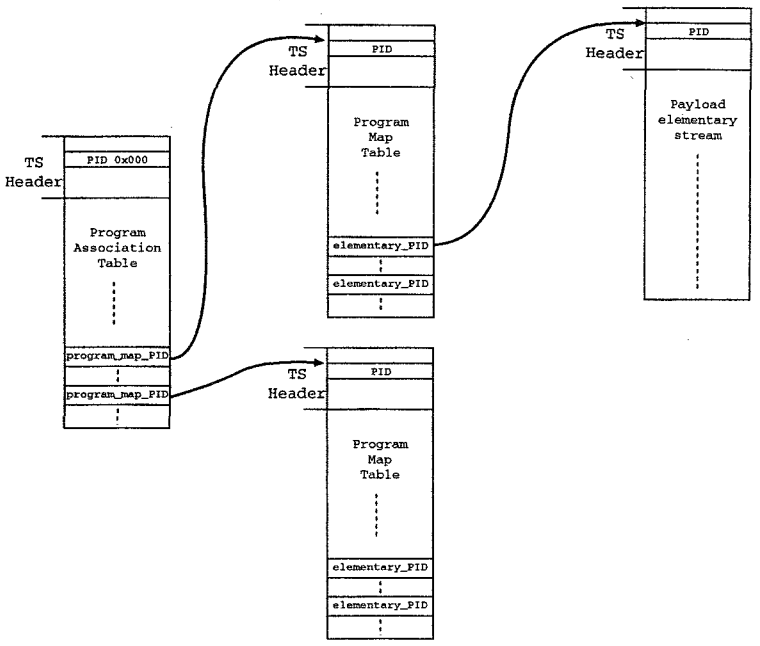


FIG. 3.2 – La structuration des données

3.1.3 La synchronisation

Pour pouvoir décoder, un récepteur doit être synchronisé sur le début des paquets. En effet, un nouveau récepteur peut se brancher à n'importe quel moment sur le flux et doit pouvoir retrouver les informations système contenues dans l'en-tête de chaque paquet. Pour ce faire, un octet particulier, dit de *synchronisation*, est utilisé. Cet octet particulier n'apparaît qu'une fois dans l'en-tête d'un paquet (en première position) et est assez rare dans le reste du signal. Ainsi, pour se synchroniser, le récepteur recherche cet octet et une fois celui-ci détecté, il attend d'en trouver n autres espacés de la taille d'un paquet afin d'être sûr de la validité de la synchronisation. C'est seulement une fois les n octets consécutifs trouvés que le récepteur est synchronisé et peut commencer à considérer les paquets TS qu'il reçoit. Dans la norme, le nombre n est fixé à 5. La perte de synchronisation est beaucoup plus rapide, et demande seulement l'absence de 2 octets consécutifs dans le flux.

Le paramètre *TS_sync_loss* de la norme permet donc de vérifier que le signal transmis permet l'acquisition de la synchronisation par les récepteurs connectés. Un autre paramètre *Sync_byte_error* vérifie que chaque paquet reçu commence bien par un octet de synchronisation.

3.1.4 La synchronisation des horloges locales

Chaque programme a sa propre horloge et les paquets le constituant sont horodatés avec la valeur de cette horloge. Cet horodatage étant essentiel à la reconstruction du signal, le récepteur et l'émetteur doivent avoir des horloges correctement réglées et synchronisées. De ce fait, un outil de monitoring doit obligatoirement gérer autant d'horloges qu'il y a de programmes et remettre celles-ci à jour quand nécessaire. Ainsi, le protocole impose l'envoi, par l'émetteur, de sa valeur d'horloge (*Program Clock Reference*, **PCR**) afin que le récepteur puisse repositionner son horloge en fonction de l'horloge émise. Ce principe est résumé sur le schéma suivant, qui présente un cas simplifié dans lequel émetteur et récepteur ne se transmettent qu'une horloge :

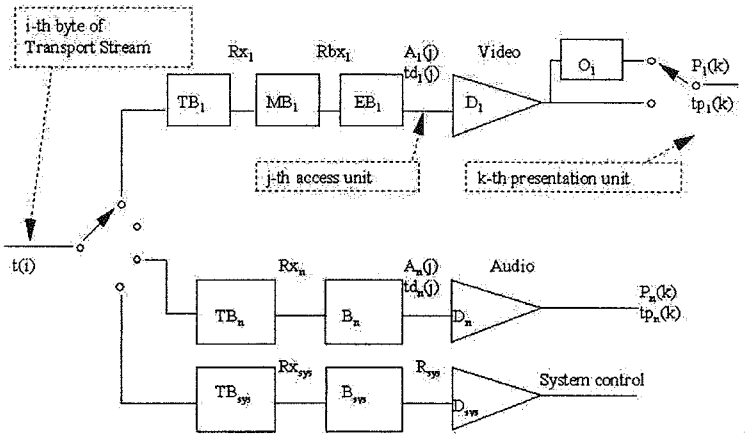
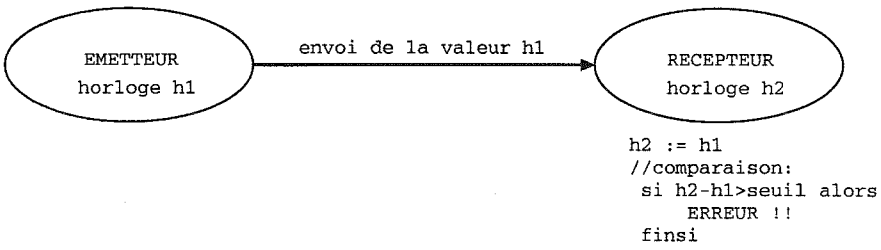


FIG. 3.3 – Le décodeur T-STD



Pour que ce système fonctionne correctement il faut que l'émission de l'horloge de l'émetteur soit suffisamment régulière et que les deux horloges ne divergent pas trop vite. Ces deux sources d'erreurs sont évidemment prises en compte dans l'évaluation de la qualité de service et sont l'objet d'un protocole de vérification assez important.

Les paramètres *PCR_repetition_error*, *PCR_accuracy_error* et *PCR_discontinuity_error* sont les paramètres permettant la vérification des différentes contraintes imposées aux horloges.

3.1.5 T-STD : un modèle de décodeur

Le décodeur théorique utilisé par la norme est nommé T-STD. Il est composé de buffers permettant le tri et la reconstruction des différentes données vidéo et/ou audio. Le décodeur T-STD décode un seul programme à la fois, il est donc précédé d'un filtre permettant la séparation des divers programmes composant le flux. Comme présenté figure 3.3, le décodeur est séparé en trois parties principales : le traitement des paquets vidéo, le traitement des paquets audio et le traitement des paquets systèmes. Dans le cadre de notre étude, nous nous sommes essentiellement penché sur la modélisation du décodage des données vidéo c'est pourquoi nous ne présenterons ici que cette partie des traitements [8].

Les différents traitements sur les paquets d'un programme se font à partir de 3 buffers. Le premier est nommé Transport Buffer (TB) et reçoit l'intégralité des paquets vidéo du programme émis par le réseau. Ce buffer est chargé de filtrer les paquets redondants, de désencapsuler les informations contenues dans les paquets et de les transmettre au buffer suivant à un certain taux de transfert. Le deuxième buffer, Multiplexing buffer (MB), doit analyser les informations ainsi reçues, supprimer les informations de gestion encore existantes et transmettre les données utiles au dernier buffer. Le dernier buffer, Elementary

[8] Les autres chaînes de décodage sont similaires à celle-ci.

Buffer (EB) est chargé de conserver les données utiles d'une image jusqu'à son instant de présentation (PTS). Il envoie alors les données à un décodeur qui reconstruit l'image. Pour le décodeur abstrait T-STD, on considère les temps d'émission et de décodage comme nuls.

Nous résumerons donc le fonctionnement de la partie vidéo du décodeur comme suit :

- Le buffer TB reçoit tous les paquets TS d'un PID donné en provenance du réseau. Il épure ces paquets de leur en-tête (après certaines vérifications) et transmet la partie utile au buffer suivant. Les paquets éventuellement dupliqués ne sont transmis qu'une seule fois. Le taux de transmission des données du buffer TB au buffer suivant est fonction du remplissage de TB.
- Le buffer MB reçoit des données en provenance du buffer TB. Ces données sont des paquets PES (*Packetized Elementary Stream*) (qui étaient encapsulés dans des paquets TS). Le buffer MB réalise des vérifications et des analyses sur l'en-tête PES puis envoie, à un taux constant, la partie utile du paquet (sans l'en-tête) au dernier buffer.
- Le buffer EB reçoit des données en provenance du buffer MB. Ces données sont des parties d'images. En effet, une image est constituée d'un certain nombre de paquets PES et la succession de deux images dans le flux PES est marquée par une séquence de début d'image. Lorsque l'instant de présentation d'une image arrive, le buffer émet l'ensemble des données reçues concernant l'image en question. Il doit donc conserver les octets constituant une image jusqu'au moment de son émission.

Le T-STD fonctionne selon deux méthodes distinctes : *vbv_delay* et *leak*. Ces deux méthodes sont des modes de fonctionnement du T-STD légèrement différents et impliquant des vérifications différentes. La méthode *vbv_delay* se distingue par un fonctionnement en flux tendu qui interdit au buffer MB d'être vide.

Un nombre important de paramètres est chargé de vérifier le bon comportement du buffer T-STD afin d'évaluer la décodabilité du signal reçu. Les paramètres *TB_buffering_error*, *MB_buffering_error* et *EB_buffering_error* vérifient l'état des différents buffers constituant le décodeur et déclenchent des alarmes en cas de débordements. Les mêmes vérifications sont réalisées sur la partie "réception de paquets système" par les paramètres *TBsys_buffering_overflow*, ... La paramètre *Empty_buffer_error* est, lui, chargé de vérifier que les buffers ne sont pas vides et fonctionnent en flux tendu.

3.1.6 Les paramètres de QdS

Avec la multiplication des vérifications demandées par les normes, l'établissement d'un diagnostique et la correction des problèmes sur un réseau DVB demandent un haut niveau d'expertise. En effet, la plupart du temps, un nombre important d'alarmes se déclenchent (du fait de la non-hiérarchisation de ces alarmes) et une grande expérience est nécessaire pour déterminer la cause réelle de ces alarmes en cascade.

Les nouveaux paramètres QdS (ou de synthèse) ajoutés sont issus d'une étude psycho-visuelle de grande envergure menée par TDF C2R dans le cadre des projets européens QuoVadis [93] et Mosquito [84]. Cette étude est basée sur un principe simple : présenter un flux entaché d'erreurs (contrôlées et introduites volontairement) à des téléspectateurs, ceux-ci devant alors donner leurs appréciations sur la qualité visuelle et auditive du programme. L'étude a permis de dégager une première classification empirique des paramètres permettant de juger de la qualité de transmission sur un nombre très réduit de paramètres (trois), obtenus par composition de paramètres de base.

Perte de signal : *Service_Availability_error*

Le but de ce premier paramètre de synthèse *Service_Availability_error* (ou *SA_error*) est d'identifier des situations de perte du signal. Ce paramètre indique l'impossibilité de recevoir un programme ou bien l'existence d'importantes distorsions de celui-ci. La valeur de ce paramètre est calculée sur une fenêtre de temps choisie ΔT. L'équation suivante exprime le calcul du paramètre :

$$Max[TS\_Sync\_loss(\Delta T), PAT\_error(\Delta T), PMT\_error(\Delta T)] \tag{3.1}$$

Le paramètre *Service_Availibility_error* est donc composé de paramètres basiques permettant de vérifier la synchronisation et la gestion des paquets (voir section 3.1.3).

Graves perturbations : *Service_Degradation_error* :

Le rôle du paramètre *Service_Degradation_error* (ou *SD_error*) est l'identification d'importantes dégradations de la qualité de l'image et des conditions de réception. Ce paramètre est censé intervenir avant la perte complète de la réception. Les projets Mosquito et QuoVadis ont permis d'identifier un ensemble de paramètres ETSI symptomatiques de ce genre de situation et le paramètre *SD_error* est donc fonction de ces paramètres de base comme le présente la formule suivante :

$$Max[CRC\_error(\Delta T), PCR\_error(\Delta T), NIT\_error(\Delta T), SDT\_error(\Delta T)] \quad (3.2)$$

Le paramètre *Service_Degradation_error* est le plus complexe des paramètres de QdS. Il utilise les paramètres d'évaluation concernant les horloges et l'horodatage (PCR) mais aussi des paramètres concernant des tables "secondaires" (les NIT et les SDT). Enfin, il utilise le paramètre *CRC_error* qui est lui même une composition d'un ensemble de vérifications.

Faibles perturbations : *Service_Impairments_error*

Enfin, le dernier paramètre, *Service_Impairments_error* ou *SI_error*, a pour objectif d'indiquer de légères dégradations des conditions de réception. Une fois encore, le choix des paramètres caractéristiques de cette situation de diffusion est issu des résultats statistiques des études Mosquito et QuoVadis. La valeur du paramètre *SI_error* est donnée par l'équation suivante :

$$Max[Continuity\_count\_error(\Delta T), Transport\_error(\Delta T)] \quad (3.3)$$

Le paramètre *Service_impairement_error* est constitué de deux paramètres *Continuity_count_error* et *Transport_error* qui permettent des vérifications dans l'ordre de réception des paquets et la correction du paquet suite à l'application de codes correcteurs (Viterbi, ...).

3.1.7 Conclusions

Nous avons décrit rapidement les grands principes de fonctionnement du système que nous modélisons. Une difficulté importante de l'étude de cas est le nombre important de paramètres à prendre en compte. L'évaluation des différents paramètres doit permettre de diagnostiquer les pannes et les dysfonctionnements du système et la réalisation, dans un SoC, de ce grand nombre de traitements se confronte à l'ensemble des problèmes traditionnels de conception (architecture, consommation, coût, etc).

La cohérence de l'ensemble des paramètres, au vu du processus de normalisation, n'est pas garantie. En effet, chaque leader du monde de la télévision numérique impose ses propres vérifications. De ce fait on peut d'emblée se poser des questions sur le niveau de couverture des paramètres proposés : l'ensemble des pannes du réseau est-il détecté avec ce jeu de vérifications ? Si c'est le cas, existe-t'il des vérifications redondantes ?

L'ajout de paramètres de *Qualité de Service* semble indiquer qu'il est possible de hiérarchiser les paramètres proposés afin de dégager les grandes lignes de l'état de la transmission en fonction d'un nombre relativement réduit de paramètres.

De plus la norme n'indique pas ou très peu l'ordonnancement des différents paramètres. Le spécialiste du domaine connaît, intuitivement, un ordre implicite des calculs à réaliser mais cette information n'est jamais explicitée complètement.

L'évaluation de certains paramètres est très complexe et pose des problèmes électroniques assez importants notamment en matière de temps réel. Le problème de concevoir une application implantant de manière cohérente l'ensemble des paramètres est relativement complexe en raison des les interactions entre les différents paramètres.

3.2 Réalisations

Au début du projet EQUAST, les problématiques de l'électronique ne nous étaient pas familières. De même, les problématiques de la télévision numérique paraissaient floues, la norme n'étant que très peu structurée et peu adaptée aux novices du domaine.

De leur côté, les chercheurs en électronique n'avaient pas une idée très précise de l'apport d'une méthode formelle dans leurs développements. Nous avons donc décidé de travailler ensemble sur des points particuliers de la norme.

L'équipe électronique a suivi le flot de conception standard (voir figure 1, page 3). Ayant déjà une spécification de l'application (la norme) et une vue approximative du système, elle a proposé un partitionnement naturel : chaque paramètre de la norme est un bloc fonctionnel séparé.

Du fait de leur expertise dans les domaines électroniques, les concepteurs électroniques et les experts en télévision numérique ont jugé "difficiles" un certain nombre de paramètres. Ces blocs fonctionnels ou modules ont donc fait immédiatement l'objet de développements (VHDL et/ou SystemC) et d'estimation de performances.

Afin d'établir un dialogue et une méthodologie commune, nous avons modélisé ces paramètres en parallèlement à la conception électronique. Les modèles que nous avons proposés ont permis un grand nombre de discussions entre électroniciens et informaticiens. Le principal intérêt de ces modèles est d'avoir suscité ces discussions qui ont amené une meilleure compréhension de la télévision numérique et l'établissement d'une méthodologie pour le projet. Les travaux dans ce cadre regroupent quatre études distinctes :

1. L'algorithme d'acquisition de la synchronisation. Cet algorithme est à la base de toute l'analyse TS. La question de la validité de l'algorithme s'est également posée.
2. La mesure du débit du flux.
3. L'analyse de la décodabilité des données transmises par le T-STD.
4. L'étude de la dérive temporelle des horloges (paramètres concernant le PCR). Ce travail a été réalisé par Clearsy et ne sera pas présenté dans ce document.

Nous allons détailler ces premières approches des problématiques électroniques. Chaque développement a permis d'essayer une approche différente, de souligner et dégager des solutions intéressantes dans le cadre d'une conception architecturale.

3.3 L'acquisition de la synchronisation

Comme présentée dans la partie 3.1.3, l'acquisition de la synchronisation est fondamentale pour tout récepteur DVB-T. Dans le chapitre 1, et plus particulièrement dans la section concernant la méthode B nous avons déjà présenté, à titre d'exemple, les modèles et l'algorithme obtenus pour ce problème. Un des problèmes concernant cet algorithme a été de vérifier que, sous couvert d'un flux "bien construit", l'algorithme permettait également une acquisition correcte de la synchronisation. En effet, on peut supposer que la valeur spécifique d'octet servant de base à la synchronisation est utilisée dans le codage des informations utiles (payload). La synchronisation peut elle être acquise dans ce cas et quelle est la valeur du paramètre TS_sync_loss censé indiquer la bonne construction du flux en matière de synchronisation ?

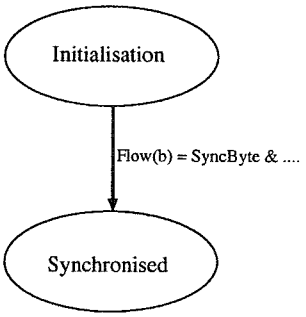
Nous avons donc refait des modèles en nous attachant à définir plus formellement la notion de flux "bien construit". Ces modèles vont également permettre de montrer la terminaison de l'algorithme d'acquisition de la synchronisation.

3.3.1 Modèle abstrait : caractériser le flux

Le modèle suivant est un peu différent de l'exemple pris pour présenter la méthode B événementielle. Ce modèle ne s'intéresse pas ou peu à l'algorithme d'acquisition de la synchronisation (c'est l'objectif du développement précédent) mais s'intéresse surtout à décrire l'interaction environnement-système et ses conséquences sur la détection de la synchronisation.

Notre modèle définit deux ensembles abstraits $BYTE$ et $VALUE$ qui ont la même signification que dans le premier développement concernant la synchronisation. De même pour les constantes $SyncByte$, $Flux$ et $TSsize$ qui représentent respectivement l'octet de synchronisation, le flux des données et la taille d'un paquet TS. On considère que le flux contient réellement des octets de synchronisation ($SyncByte \in ran(Flux)$) et on ne fixe pas de bornes au flux. Ce flux est considéré comme parfait : les paquets sont correctement identifiés et débutent tous par un octet de synchronisation conformément à la norme ($\forall\, x.(x \in \mathbb{N} \Rightarrow Flow(x * TSsize) = SyncByte)$).

MODEL
$SYNCHRONI$
SETS
$BYTE$;
$VALUE = OK, KO, IND$
CONSTANTS
$SyncByte, Flow, TSsize$
PROPERTIES
$SyncByte \in BYTE \wedge$
$Flow \in \mathbb{N} \rightarrow BYTE \wedge$
$TSsize \in \mathbb{N}_1 \wedge$
$SyncByte \in ran(Flow)$
$\forall x.(x \in \mathbb{N} \Rightarrow Flow(x * TSsize) = SyncByte)$
VARIABLES
TS_sync_loss
INVARIANT
$TS\_sync\_loss \in VALUE$
DEFINITIONS
$nb \triangleq 5$
INITIALISATION
$TS\_sync\_loss := IND$



L'événement Synchronised permet de définir une transition du système. S'il est possible de trouver dans le flux, à partir d'un octet de synchronisation (l'octet numéro b) nb paquets successifs commençant par un octet de synchronisation ($\forall x.(x \in 1..nb \Rightarrow Flow(b + TSsize * x) = SyncByte)$, alors le système acquiert la synchronisation ($TS\_sync\_loss := OK$).

Synchronised $\widehat{=}$
ANY
b
WHERE
$b \in \mathbb{N} \wedge Flow(b) = SyncByte \wedge$
$\forall x.(x \in 1..nb \Rightarrow Flow(b + TSsize * x) = SyncByte)$
THEN
$TS\_sync\_loss := OK$
END

Ce modèle formel permet de se poser une importante question sur la constitution du flux. En premier lieu, est-il possible de trouver nb **faux** octets de synchronisation espacés de la taille d'un paquet TS dans un flux ? D'après les experts de la télévision numérique, un tel phénomène est impossible (même en cas de fortes perturbations électromagnétiques). Par ailleurs, pour les experts, si un tel phénomène se produisait, il serait normal que les différents appareils du réseau se synchronisent sur "cette" fausse synchronisation. Grâce à sa simplicité, le modèle formel proposé a grandement amélioré notre compréhension de la philosophie du monde DVB-T et a permis des débats de fond. Ce premier modèle est le résultat d'un apprentissage et de discussions avec les experts du domaine de la DVB-T.

3.3.2 Premier raffinement

Dans ce raffinement, nous nous contentons d'expliciter l'acquisition de la synchronisation. Dans l'abstraction, le système est vu dans sa globalité et un récepteur connaît l'intégralité du flux. Ce raffinement va permettre de passer à une vue plus locale (encore très abstraite) afin de montrer que le fait de comptabiliser les octets successifs de synchronisation permet bien d'acquérir la synchronisation.

On se munit pour cela d'un pointeur *pointer* indiquant l'octet en cours d'analyse et d'un compteur *count* indiquant le nombre de paquets successifs correctement reçus. C'est la variable *pointer* qui symbolise l'aspect local du traitement, le système ne considérant le flux que par l'intermédiaire de ce pointeur.

On établit des propriétés intéressantes comme la corrélation entre la valeur de l'octet référencé par le pointeur et le nombre de paquets corrects successifs déjà comptés. Une autre propriété intéressante est le fait que le pointeur *pointer* se rapproche toujours d'un octet de synchronisation lors de l'exécution des nouveaux événements (variant) : il existe toujours un octet de synchronisation à moins de $TSsize$ octets. Pour simuler le branchement d'un appareil sur le réseau, on initialise au hasard la valeur du pointeur.

REFINEMENT
$SYNCHRO2$
REFINES
$SYNCHRONI$
VARIABLES
$TS\_sync\_loss, pointer, count$
INVARIANT
$pointer \in \mathbb{N} \wedge$
$count \in \mathbb{N} \wedge$
$pointer - TSsize * count \geq 0 \wedge$
$\forall x.(x \in 1..count \wedge Flow(pointer) = SyncByte \Rightarrow Flow(pointer - TSsize * x) = SyncByte) \wedge$
$\exists b.(b \in \mathbb{N} \wedge b * TSsize \geq pointer \wedge Flow(b * TSsize) = SyncByte \wedge b - pointer \leq TSsize)$
VARIANT
$min(\{b \mid b \in \mathbb{N} \wedge b * TSsize \geq pointer \wedge Flow(b * TSsize) = SyncByte\}) - pointer$
DEFINITIONS
$nb \triangleq 5$
INITIALISATION
$TS\_sync\_loss := IND \parallel$
$count := 0 \parallel$
$pointer :\in \mathbb{N}$

La version raffinée de l'événement Synchronised permet de définir sans ambiguïté le passage à l'état synchronisé. Lorsque le système a compté nb paquets successifs corrects, il est synchronisé.

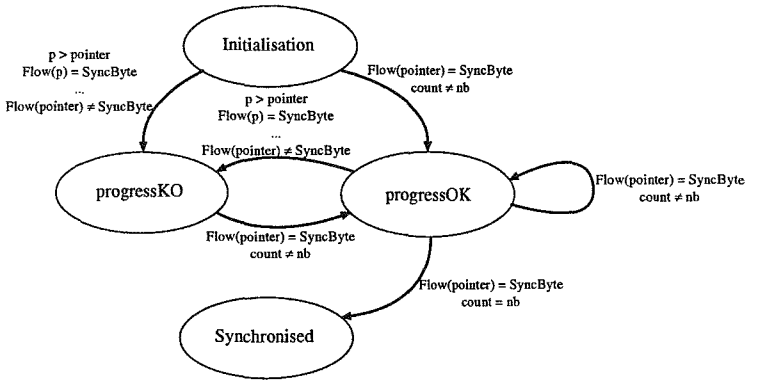
Synchronised $\widehat{=}$
SELECT
$Flow(pointer) = SyncByte \wedge$
$count = nb$
THEN
$TS\_sync\_loss := OK$
END

Le couple d'événements progressKO et progressOK décrit la progression du système lorsque celui-ci n'est pas synchronisé. Lorsque le matériel connecté ne trouve pas d'octet de synchronisation à la position courante, celui-ci attend le premier octet de synchronisation p dans le flux. L'événement progressKO décrit bien ce phénomène. Par ailleurs, l'événement progressOK décrit le comportement du système lorsqu'un octet de synchronisation a été détecté. Dans ce cas, le système incrémente un compteur (*count*) et attend le prochain paquet afin de vérifier qu'il y a encore un octet de synchronisation. La substitution $pointer := pointer + TSsize$ permet de simuler cette attente et la réception progressive des données.

progressKO $\widehat{=}$
ANY
p
WHERE
$p \in \mathbb{N} \wedge$
$p > pointer \wedge$
$Flow(p) = SyncByte \wedge$
$\forall b.(b \in \mathbb{N} \wedge b > pointer \wedge b < p \Rightarrow$
$Flow(b) \neq SyncByte) \wedge$
$Flow(pointer) \neq SyncByte$
THEN
$count := 0 \parallel$
$pointer := p$
END

progressOK $\widehat{=}$
SELECT
$Flow(pointer) = SyncByte \wedge$
$count \neq nb$
THEN
$pointer := pointer + TSsize \parallel$
$count := count + 1$
END

On simule la réception du flux par la progression de la variable *pointer*. Le système surveille l'arrivée d'un premier octet de synchronisation et ensuite "compte" les octets reçus. Une fois le nombre d'octets composant un paquet atteint, le récepteur doit trouver un octet de synchronisation indiquant le début d'un nouveau paquet et ce jusqu'à atteindre le nombre de *nb* paquets successifs. Le récepteur est alors considéré comme synchronisé (cf événement Synchronised). Le système de transition suivant illustre ce nouveau raffinement. Deux nouveaux états, issus des deux nouveaux événements du système, apparaissent alors que l'état d'acquisition de la synchronisation est désormais conditionné par l'incrémentation d'un compteur (événement progressOK).



3.3.3 Raffinements de l'algorithme d'acquisition de la synchronisation

Les nouveaux raffinements vont consister en une explicitation de l'algorithme d'acquisition de la synchronisation. En effet, dans la version abstraite précédente, on considère que l'intégralité du flux est disponible et la progression dans le flux est modélisé par un pointeur *pointer*. Lorsque l'on a trouvé un octet de synchronisation correct, le pointeur est déplacé de manière immédiate jusqu'au début du paquet TS suivant ($pointer := pointer + TSsize$). Dans le système réel, un tel saut n'est pas possible car les données constituant le flux sont reçues au fur et à mesure. La progression dans le flux n'est donc pas réalisée comme décrit dans le précédent modèle.

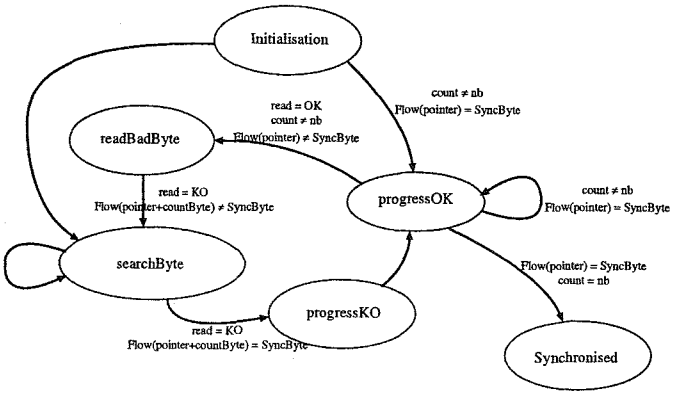
Détails de la recherche d'un octet de synchronisation

Dans ce nouveau raffinement nous allons introduire deux variables *read* et *countByte*. Ces variables vont permettre d'expliciter un peu le comportement du système lors de la recherche d'un octet *SyncByte*. La variable *read* prend ses valeurs dans l'ensemble $\{OK, KO\}$:

- $read = OK$ signifie qu'un ou plusieurs octets *SyncByte* (distant chacun de $TSsize$) ont été détectés. Le système est dans un état de "pré-synchronisation".

- $read = KO$ signifie que le système n'a pas trouvé d'octet *SyncByte* dans le flux déjà reçu ou que les octets trouvés ont été *invalidés* car ils ne permettaient pas de constituer une chaîne de *nb* paquets successifs.

Le système de transition ci-dessous représente ce nouveau raffinement. Les trois états readBadByte, searchyte, et progressKO illustrent bien le comportement du système lors de découverte d'un octet de synchronisation incorrect. Dans un premier temps, le système constate que l'octet courant est défectueux (readbadByte) puis il se met en recherche d'un octet correct (searchByte). Lorsqu'un octet correct est trouvé, la phase de recherche est terminée et le système se trouve dans l'état concret progressOK qui est cohérent avec sa version abstraite : l'octet courant est incorrect mais un octet correct est trouvé.



La variable *countSize* représente la réception des octets du flux. Cette variable n'a de sens que lorsque $read = OK$. L'invariant du modèle résume ces propriétés et précise qu'il n'existe pas d'octet *SyncByte* dans les octets reçus depuis la perte de l'état de *pré-synchronisation*, c'est-à-dire entre *pointer* et $pointer + countSize$:

$$
\begin{array}{l}
countByte \in NATURAL \wedge \\
read \in \{OK, KO\} \wedge \\
(read = OK \Rightarrow countByte = 0) \wedge \\
\forall b.(b \in NATURAL \wedge read = KO \wedge \\
\quad b \geq pointer \wedge b < pointer + countByte \Rightarrow Flow(b) \neq SyncByte)
\end{array}
$$

La dernière propriété de l'invariant est particulièrement intéressante. Elle permet de garantir que, lors de la recherche d'un octet de synchronisation, aucun octet de synchronisation n'est présent dans la tranche $pointer..pointer + countByte$ de flux déjà reçu. Le flux étant un flux bien formé, le système se rapproche donc progressivement d'un octet de synchronisation. Un nouvel événement searchByte représente la réception du flux et l'événement progressKO est raffiné. Les octets du flux sont analysés lors de leurs réceptions tant qu'un octet de synchronisation n'a pas été détecté.

progressKO $\widehat{=}$
SELECT
$read = KO \wedge$
$Flow(pointer + countByte) = SyncByte \wedge$
$Flow(pointer) \neq SyncByte$
THEN
$count := 0 \parallel$
$read := OK \parallel$
$pointer := pointer + countByte \parallel$
$countByte := 0$
END

searchByte $\widehat{=}$
SELECT
$read = KO \wedge$
$Flow(pointer + countByte) \neq SyncByte$
THEN
$countByte := countByte + 1$
END

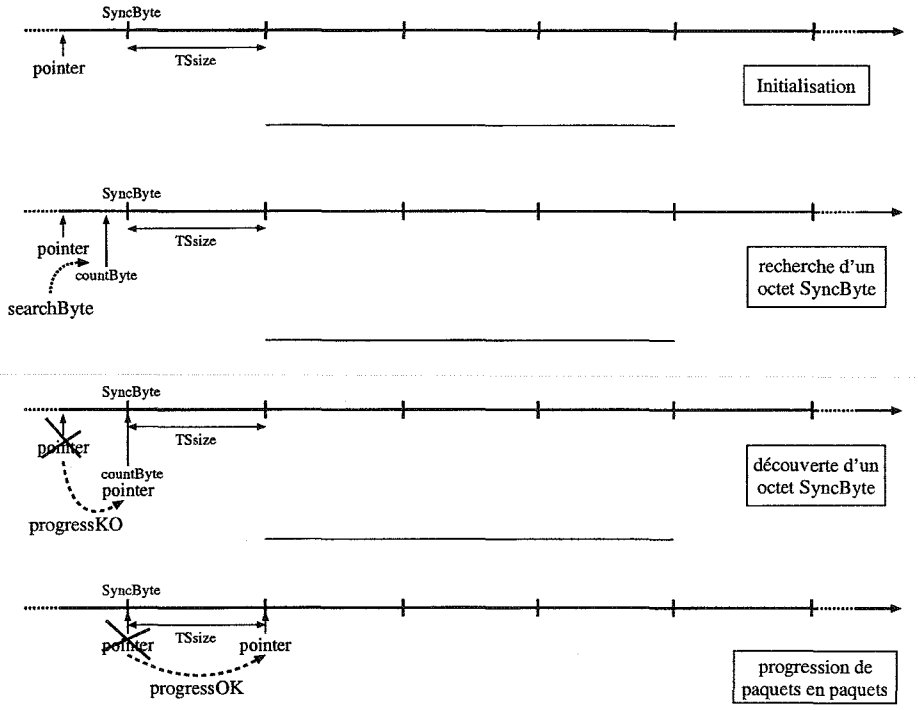


FIG. 3.4 – Comportement de la synchronisation

Les autres événements du modèle sont conservés, la prise en compte des nouvelles variables les modifiant cependant légèrement. A titre d'exemple, nous présentons ici le code de l'événement raffiné progressOK et du nouvel événement readBadByte permettant de passer de l'état de pré-synchronisation à l'état de recherche d'un octet de synchronisation :

progressOK $\widehat{=}$
SELECT
$Flow(pointer) = SyncByte \wedge$
$count \neq nb$
THEN
$read := OK \parallel$
$pointer := pointer + TSsize \parallel$
$count := count + 1$
END

readBadByte $\widehat{=}$
SELECT
$read = OK \wedge$
$count \neq nb \wedge$
$Flow(pointer) \neq SyncByte$
THEN
$read := KO \parallel$
$countByte := 0$
END

La figure 3.4 illustre le comportement de la variable *pointer* : tous les octets sont inspectés jusqu'à la découverte d'un octet de synchronisation. Ce type d'octet étant caractéristique des débuts de paquets, la variable *pointer* se déplace alors par "bonds" successifs afin d'inspecter les octets supposés être des débuts de paquets.

Expliciter l'état de pré-synchronisation

Lorsque le système est pré-synchronisé, il a trouvé un ou plusieurs octets séparés deux à deux par une distance de $TSsize$ octets. Les octets séparant ces débuts de paquets implicites ne sont pas examinés lors de la recherche de la synchronisation. De ce fait, de "faux" octets de synchronisation (càd des octets ayant la valeur *SyncByte* mais n'étant pas des débuts de paquets) peuvent être contenus dans les paquets sans pour autant perturber l'acquisition de la synchronisation. Ce nouveau raffinement illustre ce phénomène

en introduisant une nouvelle variable $POINTER$ représentant la réception d'octets du flux durant la phase de pré-synchronisation. Une seconde variable, $countSize$ est introduite et permet de compter le nombre d'octets reçus suite à la détection d'un octet de synchronisation.

$$
\begin{array}{l}
POINTER \in NATURAL \wedge \\
countSize \in 0..TSsize \wedge \\
(read = KO \Rightarrow countSize = 0) \wedge \\
(read = OK \Rightarrow pointer + countSize = POINTER)
\end{array}
$$

L'introduction de la variable concrète $POINTER$ n'implique pas la suppression de la variable abstraite *pointer* qui continue à être présente dans ce raffinement. Le modèle introduit un nouvel événement progressByte qui représente la réception et le traitement des octets du flux durant la phase de présynchronisation. L'événement progressOK est à nouveau raffiné :

progressByte $\widehat{=}$
SELECT
$read = OK \wedge$
$countSize \neq TSsize$
THEN
$POINTER := POINTER + 1 \parallel$
$countSize := countSize + 1$
END

progressOK $\widehat{=}$
SELECT
$Flow(pointer) = SyncByte \wedge$
$countSize = TSsize \wedge$
$count \neq nb$
THEN
$read := OK \parallel$
$pointer := POINTER \parallel$
$countSize := 0 \parallel$
$count := count + 1$
END

Les deux événements précédents modélisent le comportement du système lorsque celui-ci est fixé sur un octet de synchronisation. Les octets suivants ne sont pas inspectés et le système se contente d'attendre le $TSsize$ième octets afin de vérifier sa validité. Si l'octet consulté n'est pas un octet de synchronisation, l'état de pré-synchronisation du système est perdu :

readBadByte $\widehat{=}$
SELECT
$read = OK \wedge$
$count \neq nb \wedge$
$countSize = 0 \wedge$
$Flow(pointer) \neq SyncByte$
THEN
$read := KO \parallel$
$countByte := 0$
END

La figure 3.5 explicite donc le fonctionnement du système après détection d'un premier octet de synchronisation : le système se contente de compter les octets reçus et attend la réception des $TSsize$ octets constituant un paquet TS. La variable abstraite *pointer* modélise ce comportement par une progression dans le flux par sauts successifs. La variable $POINTER$ explicite la réception progressive des octets constituant un paquet.

Modèle final : réception continue du flux

Dans ce dernier modèle nous supprimons les variables abstraites *pointer*, $countByte$ et $POINTER$ et introduisons la variable concrète $realPointer$ permettant de modéliser la réception des octets du flux. Tous

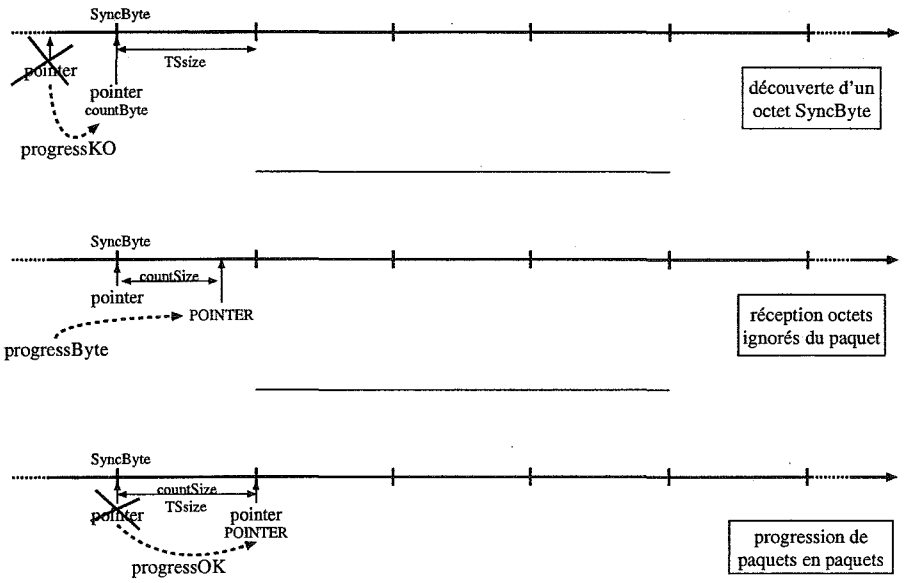


FIG. 3.5 – Comportements des variables *pointer* et $POINTER$

les événements de ce modèle incrémentent cette variable $realPointer$, hormis l'événement readBadByte qui représente une simple consultation, modélisant ainsi la réception des nouveaux octets. L'invariant du modèle permet de lier les variables abstraites et la nouvelle variable :

$$
\begin{array}{l}
realPointer \in \mathbb{N} \wedge \\
(read = OK \Rightarrow realPointer = POINTER) \wedge \\
(read = KO \Rightarrow realPointer = pointer + countByte) \wedge \\
(read = OK \ \wedge \ countSize \neq 0 \Rightarrow Flow(realPointer - countSize) = SyncByte)
\end{array}
$$

L'événement progressByte permet de représenter l'attente du $TSsize$ième octet suivant un octet de synchronisation par le système. Durant cette attente, les octets reçus ne sont pas considérés mais simplement comptés :

progressByte $\widehat{=}$
SELECT
$read = OK \wedge$
$Flow(realPointer - countSize) = SyncByte \wedge$
$countSize \neq TSsize$
THEN
$realPointer := realPointer + 1 \ ||$
$countSize := countSize + 1$
END

Les deux événements progressOK et readBadByte représentent la consultation du $TSsize$ième octet. L'événement progressOK valide l'octet de synchronisation précédent ($realPointer - countSize$) alors que readBadByte modélise le comportement du système en cas d'une mauvaise valeur de l'octet :

readBadByte $\widehat{=}$
SELECT
$read = OK \wedge$
$count \neq nb \wedge$
$countSize = 0 \wedge$
$Flow(realPointer) \neq SyncByte$
THEN
$read := KO$
END

progressOK $\widehat{=}$
SELECT
$countSize = TSsize \wedge$
$count \neq nb$
THEN
$read := OK \parallel$
$countSize := 0 \parallel$
$count := count + 1$
END

Dans le cas opposé, où le système cherche un premier octet de synchronisation, les événements searchByte et progressKO représentent la réception des octets du flux et les traitements réalisés par le système.

searchByte $\widehat{=}$
SELECT
$read = KO \wedge$
$Flow(realPointer) \neq SyncByte$
THEN
$realPointer := realPointer + 1$
END

progressKO $\widehat{=}$
SELECT
$read = KO \wedge$
$countSize \neq 0 \wedge$
$Flow(realPointer) = SyncByte$
THEN
$count := 0 \parallel$
$read := OK$
END

La version finale de l'événement progressKO modélise la découverte d'un nouvel octet de synchronisation et le début d'une nouvelle phase de pré-synchronisation. Ce raffinement terminal explicite le fonctionnement du mécanisme de recherche de la synchronisation qui inspecte l'intégralité des octets reçus jusqu'à trouver un octet de synchronisation. Une fois cet octet découvert, le système se contente d'inspecter les octets de "début" de paquets ou considérés comme tels. En cas d'échec (un octet de début de paquet n'est pas un octet de synchronisation) le système retourne à l'état précédent.

3.3.4 Synchronisation : conclusions

Ce développement simple nous a permis de tisser des premiers liens avec les partenaires du projet. Nous avons tout d'abord explicité l'algorithme d'acquisition de la synchronisation (voir page 43). Ce premier travail a permis un dialogue sur l'utilité et l'utilisation d'une méthode formelle telle que B dans le cadre du projet EQUAST. Nos développements ont montré que, sous réserce de recevoir un flux correctement construit, l'algorithme utilisé permettait bien d'acquérir la synchronisation attendue. En effet, la synchronisation se faisant par la détection d'un octet à intervalle régulier, on peut considérer que le cas de figure n'est pas unique dans un flux et peut provoquer un fausse synchronisation. Le modèle abstrait indique bien que la synchronisation est acquise lorsqu'il existe une suite d'octets *SyncByte* correctement espacée et suffisamment longue. Sous l'hypothèse qu'une telle suite est caractéristique des débuts de paquets, nos développements indiquent que la synchronisation obtenue est la bonne. Dans le cas réel, les flux reçus et étudiés ne sont jamais parfaits mais les raisonnements menés sur nos modèles restent tout à fait applicables : la question étudiée ici, le comportement d'un système en phase de synchronisation reste la même.

L'intérêt de la méthode B événementielle dans ce cadre est de provoquer un certain nombre de questions sur la validité de l'algorithme. En effet, l'expression des problèmes dans un formalisme rigoureux a permis aux différents intervenants du projet de s'interroger sur le sens d'une expression comme "un flux bien construit" et de considérer des situations qui n'auraient pas été prises en compte.

Par ailleurs, les différents modèles présentés ont permis une explication et une compréhension assez fine de l'acquisition de la synchronisation d'un point de vue algorithmique mais surtout d'exhiber les contraintes du flux. La méthode B a donc eu un rôle *pédagogique* qui a permis de comprendre plus profondément l'algorithme de synchronisation de la DVB-T ainsi que sa correction vis-à-vis de l'environnement du système et de la nature des données manipulées.

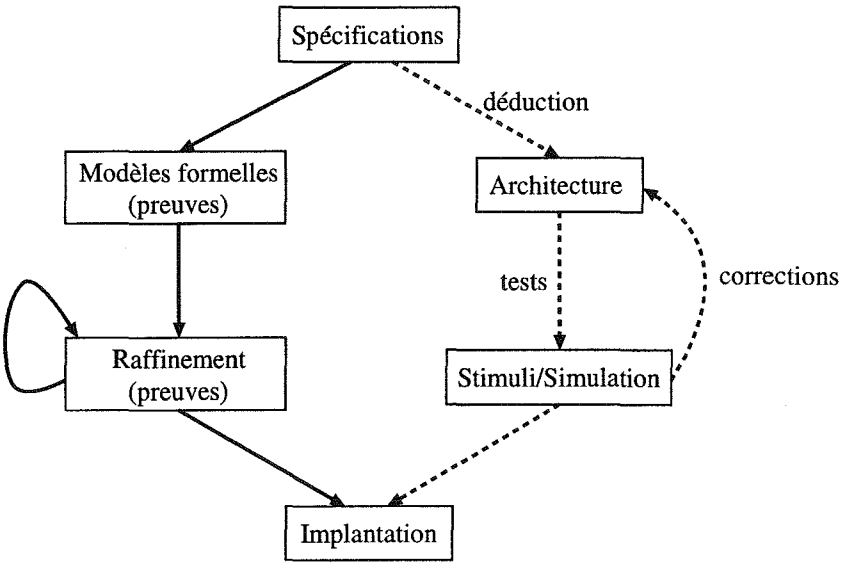


FIG. 3.6 – Schéma de conception basée sur la preuve

3.4 Le calcul du débit

Une seconde étape de notre travail a été la prise en compte de problématiques plus électroniques. En effet, les développements formels concernant l'acquisition de la synchronisation par un récepteur ne sont pas spécifiques au domaine de l'électronique et sont symptomatiques d'une utilisation "pédagogique" et standard de la méthode B.

L'idée générale mise en oeuvre dans cette seconde étude est la validation de choix architecturaux. Nous pensons que le raffinement et la preuve sont des outils puissants permettant de diriger la conception d'une architecture ou, dans cet exemple particulier, de justifier une architecture électronique.

3.4.1 Objectifs et principes de modélisation

Nous proposons de construire l'architecture par une succession de raffinements en utilisant la méthode B. Le premier modèle capture les comportements abstraits de la spécification. Les modèles ajoutent des détails et ne contredisent jamais l'abstraction (l'ensemble des modèles abstraits). Les comportements du système concret (l'architecture) sont conformes aux spécifications et la validité est garantie par la preuve. La figure 3.6 montre la démarche suivie actuellement par les électroniciens (flèches en pointillés) et celle que nous préconisons (flèches pleines).

Les électroniciens du LIEN ont commencé à modéliser leur architecture à partir de leur savoir-faire. Cette architecture suit les principes énoncés dans la norme. Différentes vues de l'architecture semblaient être de plus en plus précises. Nous avons alors proposé de lier les différentes vues par la notion de raffinement de B. Notre travail de modélisation a donc eu lieu après celui des électroniciens et nous avons profité de la connaissance de l'architecture finale pour nous guider dans le processus de raffinement.

L'avantage de cette approche est de pouvoir se dispenser des simulations et des nombreux tests, très coûteux en temps et non exhaustifs. En effet, la preuve de raffinement va nous permettre de garantir que le comportement de l'architecture finale est celui de la spécification.

3.4.2 L'algorithme de la mesure du débit

La mesure du débit est un paramètre important pour l'évaluation de la qualité d'un programme. Même si le débit du flux TS MPEG-2 est relativement constant, il n'en est pas de même pour celui d'une chaîne donnée (toutes les chaînes sont dans le même flux TS). Les paquets TS qui ont le même numéro de

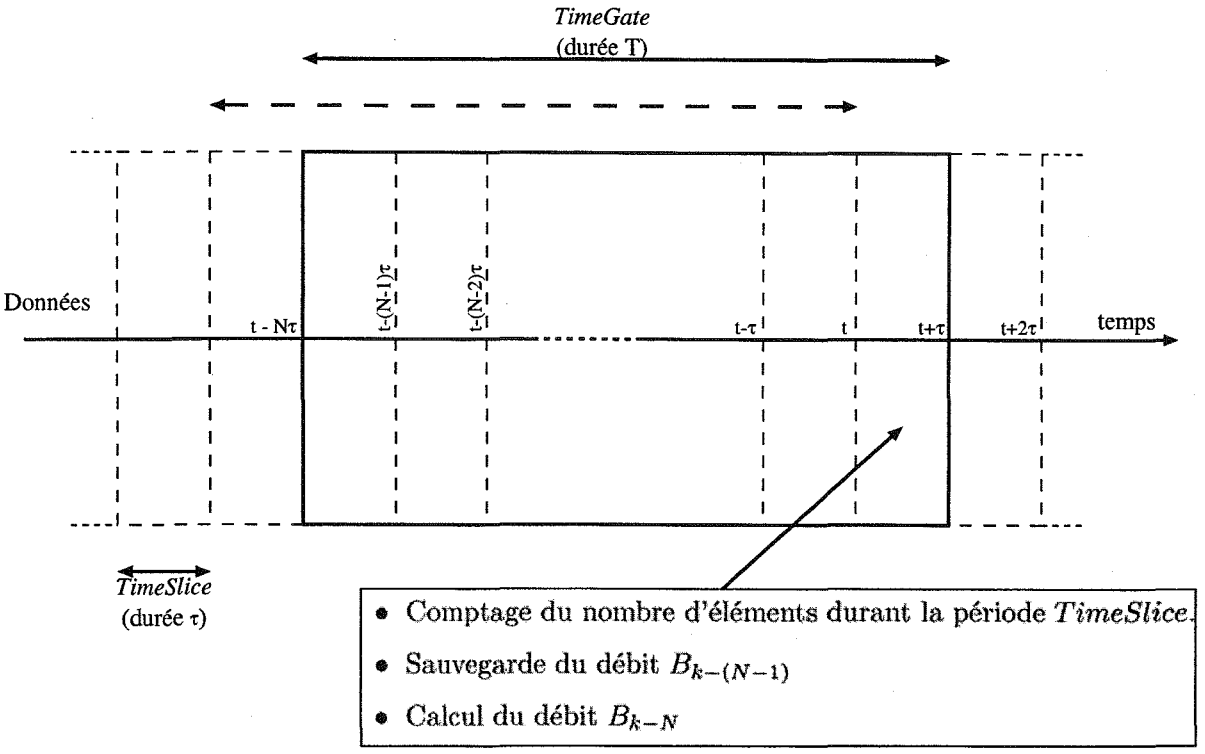


FIG. 3.7 – Illustration de la mesure du débit binaire spécifié par le standard TR 101 290 [55]

PID forment un flux TS partiel dont le débit peut varier. La mesure du débit correspond à une moyenne de valeurs calculées sur une fenêtre glissante. La figure 3.7 présente le principe général de la mesure qui dépend d'un certain nombre de paramètres :

- type d'éléments à compter (Paquets TS, Octets, Bits).
- durée de comptage (*Timegate*).
- pas de déplacement de la fenêtre de mesures (*TimeSlice*).

Le débit B_k est défini par l'équation (3.4) où T est la durée de *TimeGate* en secondes. L'unité de mesure correspond à *ElementSize*/seconde. *ElementSize* correspond soit à un paquet TS, soit à un octet ou un bit. La valeur C_{k-n} est le nombre d'*ElementSize* comptés au cours de la $k - n^{ime}$ période *TimeSlice*. N est le nombre de périodes *TimeSlice* composant une fenêtre de mesure T. A chaque glissement de la fenêtre, (pour chaque période *TimeSlice*) une nouvelle valeur du débit est calculée et maintenue pendant la durée *TimeSlice* courante. Cette mesure nécessite de mémoriser les valeurs successives de comptage (C_{k-n}) durant la durée de la fenêtre de mesure T.

$$B_k = \frac{1}{T} \times \sum_{n=1}^{N} C_{k-n} \qquad \text{(ElementSize/seconde)} \tag{3.4}$$

3.4.3 Architecture du circuit pour la mesure du débit

La mesure du $k$ème débit B_k consiste à sommer N valeurs consécutives C_i. La valeur C_i correspond à une accumulation de valeurs sur le $i$ème *TimeSlice*. L'équation récurrente suivante définit le calcul :

$$\begin{array}{lcl} D_0 & = & \sum_{k=1}^{N} C_k \\ D_{k-N} & = & D_{k-1-N} + C_k - C_{k-N} \quad \text{if } k > N \end{array} \tag{3.5}$$

De l'équation (3.5), nous déduisons la valeur du débit B_k (voir équation (3.4)). Cette équation récursive

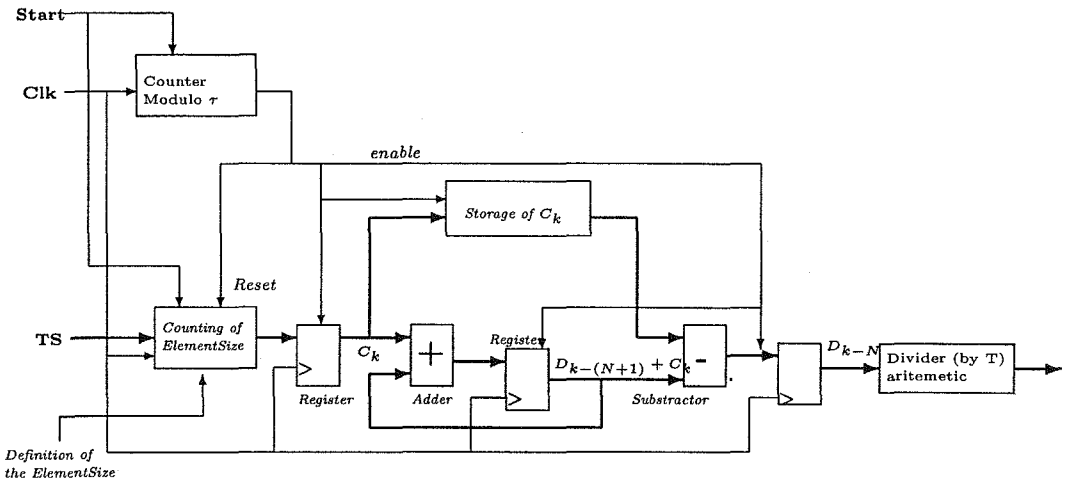


FIG. 3.8 – Architecture de la mesure du débit B_t

permet de déduire qu'une certaine période de latence est nécessaire au circuit. En effet, le calcul de taux *Bitrate* nécessite d'avoir préalablement laissé passer les N périodes *TimeSlice* composant une fenêtre de calcul. L'architecture produite par les électroniciens à partir de l'équation précédente est présentée dans la figure 3.8.

L'architecture proposée est synchronisée par une horloge Clk et est constituée des composants suivants :

- deux compteurs pour implémenter la base de temps (*TimeSlice* et *TimeGate*) à partir de l'horloge *Clk*
- le compteur qui définit *ElementSize*,
- un module de mémoire pour stocker les C_k,
- un module de calcul additionneur, soustracteur, et diviseur arithmétique.

Ces différents modules sont implantés matériellement sous la forme de portes logiques, d'ALU (Unité Arithmétique et Logique) ou de registres (Bascule LR).

3.4.4 Développement incrémental de l'architecture pour la mesure du débit

Dans cette section nous présentons les différents modèles de l'architecture que nous avons réalisés. Notre premier modèle est une abstraction qui explicite mathématiquement ce que l'architecture doit calculer. De ce fait, ce premier modèle est très proche de l'équation 3.4. Le second modèle, raffinement du précédent, explicite le calcul et se comporte comme l'équation 3.5. Le troisième modèle explicite le calcul des éléments C_i qui sont comptabilisés durant la période *TimeSlice*. Enfin le dernier modèle implémente l'ensemble infini des C_i à l'aide d'une file FIFO.

Ces différentes étapes de raffinements suivent les étapes de conception des ingénieurs électroniques ayant réalisé l'implantation. Nos modèles correspondent aux descriptions informelles des concepteurs mais la rigueur du formalisme et les preuves de raffinement vont permettre de garantir une cohérence aux différentes descriptions du système.

Le premier modèle

Comme annoncé précédemment, le but de ce modèle est de posé le problème. Nous allons donc considérer que le système abstrait reçoit en entrée le nombre d'éléments C_i à compter pour les différents intervalles de temps *TimeSlice*. Les différents C_i sont reçus dans le bon ordre et sont "calculés" de manière abstraite par l'environnement. Le système produira une mesure du débit D_k dès qu'il aura reçu assez de C_i.

Pour modéliser l'environnement et son comportement nous utilisons la technique proposée par Jean-Raymond Abrial [5]. Le circuit est modélisé avec son environnement (en respectant le principe du système

FIG. 3.9 – Première architecture

clos). Le modèle abstrait donne alternativement la main à l'environnement qui calcule les entrées du circuit puis au circuit lui-même qui calcule ses sorties. L'environnement et le circuit jouent au ping-pong et se donnent mutuellement la main. Nous introduisons donc une variable d'état *mode* qui prendra ses valeurs dans l'ensemble $\{env, cir\}$. La figure 3.9 illustre le fonctionnement global de notre première architecture.

Pour réussir à faire toutes les preuves avec le prouveur interactif [8] nous avons défini formellement et de manière récursive la fonction *sigma* comme dans un travail précédent des membres de l'équipe MOSEL [37].

$$\begin{array}{l} \forall f \cdot (f \in \mathbb{N} \nrightarrow \mathbb{N} \Rightarrow sigma(\emptyset, f) = 0) \ \wedge \\ \forall (f, n, m) \cdot \left(\begin{array}{l} f \in \mathbb{N} \nrightarrow \mathbb{N} \ \wedge \\ n \in \mathbb{N} \ \wedge \ n \leq m \ \wedge \ n..m \subseteq \mathsf{dom}(f) \\ \Rightarrow \\ sigma(n..m, f) = sigma(n..m-1, f) + f(m)) \end{array} \right) \end{array}$$

La constante N définit le nombre de termes dans la somme.

L'événement proda spécifie l'environnement. Il initialise $C(j+1)$ qui est la prochaine valeur dans la liste avec n'importe quel entier positif. Puis il redonne la main au circuit.

proda $\widehat{=}$
ANY n WHERE
$mode = env \ \wedge \ n \in \mathbb{N}$
THEN
$C(j+1) := n \ \|$
$mode := cir$
END

L'événement latence spécifie le comportement initial du circuit, lorsqu'il n'y a pas assez de C_j calculés pour pouvoir fournir une valeur de la mesure du débit. Le circuit est dans une phase de latence et ne produit rien. L'événement se prépare à recevoir le prochain C_j ($j := j+1$). Puis il redonne la main à l'environnement.

L'événement bitrate spécifie la sortie (la variable *deb*) du circuit abstrait. Cet événement représente le calcul du débit par le circuit une fois l'ensemble des C_j nécessaires au calcul reçus. La somme des différentes mesures C_j est calculée de manière immédiate à l'aide de la fonction *sigma*. L'événement se prépare à recevoir le prochain C_j ($j := j+1$) et à calculer la prochaine mesure ($i := i+1$). Puis il redonne la main à l'environnement.

latence $\widehat{=}$
SELECT
$N \notin \mathsf{dom}(C) \ \wedge \ mode = cir$
THEN
$mode := env \ \|$
$j := j+1$
END

bitrate $\widehat{=}$
SELECT
$N \in \mathsf{dom}(C) \ \wedge \ mode = cir$
THEN
$deb := sigma(i+1..(i+N), C) \ \|$
$i := i+1 \ \|$
$j := j+1 \ \|$
$mode := env$
END

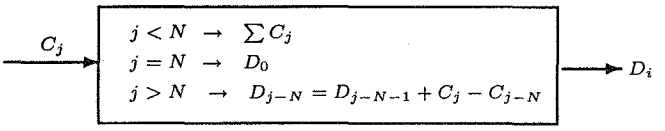


FIG. 3.10 – Une architecture plus précise (fonctionnelle)

Le deuxième modèle

Dans ce raffinement nous introduisons l'essence de l'algorithme donné par l'équation (3.5). Pour calculer le premier débit nous devons sommer les N premiers C_i puis, pour calculer les suivants, il suffira d'enlever le plus ancien des C_i et ajouter le nouveau. La figure 3.10 montre cette nouvelle architecture.

L'événement proda ne change pas. L' événement concret latence somme tous les premiers C_j dans une nouvelle variable DEB tant que j est plus petit que $N-1$.

latence $\widehat{=}$
SELECT
$mode = cir \wedge j < N-1$
THEN
$DEB := DEB + C(j+1) \parallel$
$j := j+1 \parallel$
$mode := env$
END

L'événement bitrate est raffiné en deux nouveaux événements bitrate1 et bitrate2. Ils raffinent tous les deux l'événement abstrait bitrate. Le premier initialise le processus de calcul de la mesure du débit (cas D_0 de l'équation 3.5) et le second continue le processus de calcul (autre cas de l'équation 3.5).

bitrate1 $\widehat{=}$
SELECT
$mode = cir \wedge j = N-1$
THEN
$deb := DEB + C(j+1) \parallel$
$j := j+1 \parallel$
$mode := env$
END

bitrate2 $\widehat{=}$
SELECT
$mode = cir \wedge j > N-1$
THEN
$deb := deb + C(j+1) - C(j-N-1) \parallel$
$j := j+1 \parallel$
$mode := env$
END

Pour mener à bien l'ensemble des obligations de preuves nous avons prouvé dans le premier modèle les lemmes suivants sur *sigma*. Les lemmes sont dans la clause ASSERTIONS.

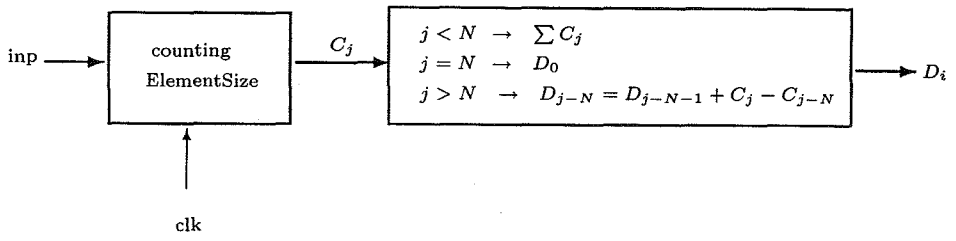


FIG. 3.11 – Architecture du deuxième raffinement

$$\forall(f,n)\cdot\left(\begin{array}{l} f\in\mathbb{N}\pfun\mathbb{N}\ \wedge\\ n\in\mathbb{N}\ \wedge\\ n\in\mathsf{dom}(f)\\ \Rightarrow\\ sigma(n..n,f)=f(n)) \end{array}\right)$$

$$\forall(f,n,m)\cdot\left(\begin{array}{l} f\in\mathbb{N}\pfun\mathbb{N}\ \wedge\\ n\in\mathbb{N}\ \wedge\\ n\leq m\ \wedge\ n..m\subseteq\mathsf{dom}(f)\\ \Rightarrow\\ sigma(n..m,f)=sigma(n+1..m,f)+f(n)) \end{array}\right)$$

$$\forall(f,n,m)\cdot\left(\begin{array}{l} f\in\mathbb{N}\pfun\mathbb{N}\ \wedge\\ n\in\mathbb{N}\ \wedge\\ n\leq m\ \wedge\\ n..m\subseteq\mathsf{dom}(f)\\ \Rightarrow\\ sigma(n..m,f)\geq f(n)) \end{array}\right)$$

Le deuxième lemme est prouvé par induction (Théorème de récurrence). Il permet de faire la preuve du raffinement de l'événement bitrate, lorsque l'on enlève la valeur la plus ancienne et que l'on ajoute la plus récente.

Nous avons également ajouté (et donc prouvé) l'invariant suivant :

$$\begin{array}{l} (j>N-1\Rightarrow i=j-N+1)\ \wedge\\ (j\leq N-1\Rightarrow i=0)\ \wedge\\ (j\leq N-1\Rightarrow DEB=sigma(1..j,C))\ \wedge\\ (j>N-1\Rightarrow deb=sigma(j-N+1..j,C))\ \wedge\\ (mode=cir\Rightarrow\mathsf{dom}(C)=1..j+1)\ \wedge\\ (mode=env\Rightarrow\mathsf{dom}(C)=1..j) \end{array}$$

Le troisième modèle

Dans ce raffinement nous introduisons une horloge et une entrée du circuit qui permettra de déterminer si le PID sur lequel nous faisons le calcul du débit est présent ou non. Nous expliquons comment est calculé $C(j)$ par l'environnement. La variable *inp* représente les données entrantes du circuit. Cette variable prends ses valeurs dans l'ensemble $estPID = \{OK, KO\}$. La variable *inp* permet de représenter l'arrivée d'un nouveau paquet et son éventuelle appartenance au PID étudié. Dans ce nouveau modèle nous détaillons le comportement de l'environnement, c'est donc un raffinement un peu spécial. La figure 3.11 montre l'architecture que nous obtenons.

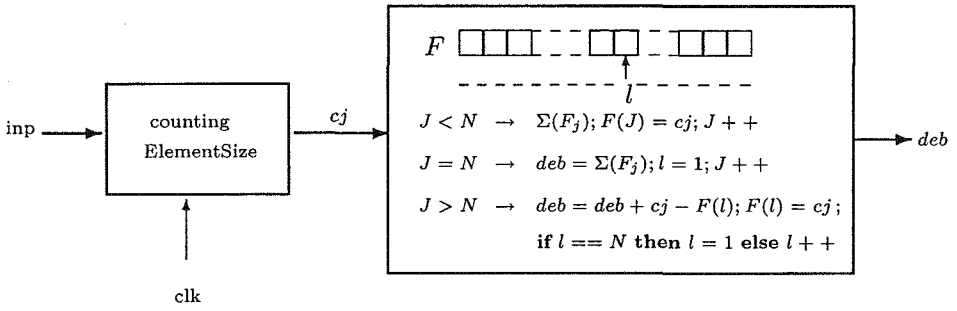


FIG. 3.12 – Architecture du dernier raffinement

count1 $\widehat{=}$
SELECT
$mode = env \wedge clk < end \wedge$
$inp = OK$
THEN
$counter := counter + 1 \parallel$
$clk := clk + 1 \parallel$
$inp :\in OKO$
END

count0 $\widehat{=}$
SELECT
$mode = env \wedge$
$clk < end \wedge$
$inp = KO$
THEN
$clk := clk + 1 \parallel$
$inp :\in OKO$
END

proda $\widehat{=}$
SELECT
$mode = env \wedge$
$clk = end$
THEN
$C(j+1) := counter \parallel$
$mode := cir$
$clk := 0 \parallel$
$counter := 0$
END

Le dernier raffinement

Dans les modèles précédents certaines valeurs de variables, comme i ou j, peuvent déborder (non codable pour une implémentation). La variable C utilise un nombre infini de place mémoire. Nous proposons, dans ce raffinement, d'implémenter C à l'aide d'une fonction F dont le domaine est fini ($\mathsf{dom}(F) = 1..N$). La table F est une file FIFO (First In First Out), elle contient toutes les valeurs C_j présentes dans la somme. Un index l dans $\mathsf{dom}(F)$ permet de connaître l'indice du premier élément de la somme. Une nouvelle variable J est introduite. Elle remplace la variable j. Elle n'est incrémentée qu'au début (latence). Pour cette période, un invariant de collage permet de faire le lien entre la variable abstraite j et la variable concrète J. La variable i disparaît donc du modèle. La figure 3.12 illustre notre dernière architecture.

L'invariant suivant montre que le contenu de chaque variable est borné (première ligne). Le lien entre j et J est décrit tout comme celui entre la fonction infinie C et la FIFO bornée F. Comme C disparaît nous avons introduit une nouvelle variable cj pour stocker la valeur produite par l'événement proda.

$$
\begin{array}{l}
J \in 0..N \ \wedge\ l \in 1..N \ \wedge\ cj \in \mathbb{N} \ \wedge \\
F \in 1..N \nrightarrow \mathbb{N} \ \wedge \\
(J \leq N-1 \Rightarrow j = J) \ \wedge \\
(J > N-1 \Rightarrow j > N-1) \ \wedge \\
(J > N-1 \Rightarrow \forall x \cdot (x \in l..N \Rightarrow F(x) = C(j-N+1+x-l))) \ \wedge \\
(J > N-1 \Rightarrow \forall x \cdot (x \in 1..(l-1) \Rightarrow F(x) = C(j+1-l+x))) \ \wedge \\
(J \leq N-1 \ \wedge\ mode = cir \Rightarrow F \lhd\!\!\!\!- \{j+1 \mapsto cj\} = C) \ \wedge \\
(J \leq N-1 \ \wedge\ mode = env \Rightarrow F = C) \ \wedge \\
(mode = cir \Rightarrow cj = C(j+1))
\end{array}
$$

Nous avons donc les événements concrets suivants :

proda $\widehat{=}$
SELECT
$mode = env \wedge$
$clk = end$
THEN
$cj := counter \parallel$
$mode := cir$
END

latence $\widehat{=}$
SELECT
$mode = cir \wedge$
$J < N - 1$
THEN
$DEB := DEB + cj \parallel$
$J := J + 1 \parallel$
$F(J+1) := cj \parallel$
$mode := env$
END

bitrate1 $\widehat{=}$
SELECT
$mode = cir \wedge$
$J = N - 1$
THEN
$deb := DEB + cj \parallel$
$J := J + 1 \parallel$
$F(J+1) := cj \parallel$
$l := 1 \parallel$
$mode := env$
END

On peut remarquer que l'événement bitrate1 initialise la variable l à 1 qui est la première place pour stocker le prochain cj. Il y a deux raffinements de l'événement bitrate2. Le premier qui se nomme bitrate21 et qui incrémente l'index l et le second qui se nomme bitrate22 et qui réinitialise la variable l à 1 pour simuler la file FIFO cyclique.

bitrate22 $\widehat{=}$
SELECT
$mode = cir \wedge J > N - 1 \wedge l \neq N$
THEN
$deb := deb + cj - F(l) \parallel$
$F(l) := cj \parallel$
$l := l + 1 \parallel$
$mode := env$
END

bitrate21 $\widehat{=}$
SELECT
$mode = cir \wedge J > N - 1 \wedge l = N$
THEN
$deb := deb + cj - F(l) \parallel$
$F(l) := cj \parallel$
$l := 1 \parallel$
$mode := env$
END

3.4.5 Mesure de débit : conclusions

Ce développement montre que l'on peut utiliser une méthode formelle et surtout le raffinement pour modéliser de manière incrémentale et prouvée une architecture. Les preuves de raffinement permettent d'être sûr que le modèle concret se comporte comme le modèle abstrait. Les différentes vues utilisées par les concepteurs de l'architecture ont été liées par la notion de raffinement du B événementiel et la vérification de la cohérence des comportements de ces vues est garantie par la preuve plutôt que par des simulations incomplètes.

Cette étude de cas, basée sur un paramètre particulier de la norme DVB-T, nous a surtout permis de comprendre la démarche de conception d'un circuit électronique et a soulevé certains problèmes. En effet, les concepteurs de circuit manipulent des descriptions architecturales de différents niveaux ; ces descriptions précisent souvent le fonctionnement de sous-parties particulières des circuits. Le problème est que les différentes descriptions ne sont pas reliées formellement les unes aux autres et rien ne garantit le respect des propriétés d'un modèle à l'autre. La méthode B événementielle a permis d'illustrer dans

ce développement l'intérêt de la preuve pour produire un circuit valide remplissant les fonctionnalités désirées.

Cependant, une telle démarche de conception ne permet pas d'utiliser au mieux les bénéfices du raffinement et de la preuve. En effet, le travail de conception est fait deux fois puisque nous avons rejoué l'étude des électroniciens. Par ailleurs, la modélisation ayant eu lieu après le travail de conception architecturale, le raffinement formel n'a pas guidé la réalisation mais a juste permis une validation a posteriori. Nous pensons que l'utilisation de la méthode B événementielle avant toute exploration architecturale aurait un bien meilleur impact sur la qualité et la sûreté de l'architecture produite.

3.5 Modélisation d'un buffer de réception

Afin d'évaluer la qualité de réception des programmes transmis, la norme propose de vérifier la gestion des buffers d'un décodeur théorique qui sert de référence et de générer un certain nombre d'erreurs en fonction des problèmes de décodage rencontrés. Ces indicateurs d'erreurs sont regroupés dans les paramètres de troisième priorité *Buffer_error*, *Empty_buffer_error* et *Data_delay_error*.

Cette dernière étude de cas simple nous a permis de suivre un flot de conception plus standard. En effet, les concepteurs électroniques n'avaient pas encore commencé d'études poussées du système. Par ailleurs, la norme proposait une description complexe du fonctionnement du système. J'ai donc commencé la modélisation du buffer de réception T-STD à partir de la norme et j'ai proposé aux électroniciens de se servir de mes modèles pour débuter leurs travaux.

En effet, les modèles construits étaient plus lisibles, plus synthétiques et plus formels que la norme. Nous avons donc proposé un document de qualité aux concepteurs afin de les aider dans leur tâche en éclaircissant le fonctionnement du T-STD. Les ingénieurs électroniques ont finalement proposé une implantation originale et performante des vérifications réalisées sur le décodeur T-STD.

3.5.1 T-STD : Abstraction

Pour simplifier le problème de modélisation, nous avons procédé de manière incrémentale. Notre première abstraction est très simple et ne reflète qu'une sous-partie visible du système global. Nous ajoutons alors des détails au fur et à mesure des différents raffinements.

Notre premier modèle n'explicite aucun détail du décodeur et considère celui-ci comme une boîte noire. Comme dans l'étude de cas précédente, l'environnement et le circuit vont se rendre mutuellement la main afin de simuler un système réactif. On considérera donc que la réception des paquets du flux se fait de manière instantanée. Les paquets sont donc fournis par l'environnement du décodeur et sont reçus intégralement. Le séquencement du modèle se fait par l'éjection d'un résultat, c-à-d une unité de présentation (vidéo ou audio) qui a lieu à un instant précis. Un certain nombre d'ensembles abstraits et de constantes caractérisent l'environnement du décodeur mais aussi les traitements réalisés ou encore des données particulières. Nous avons les ensembles et constantes suivants :

SETS
$IMAGE$;
PES;
$BIT = \{OK, KO\}$
CONSTANTS
$Dsize$
PROPERTIES
$Dsize \in \mathbb{N}_1$

La constante *Dsize* représente la taille de stockage totale du décodeur. Les ensembles PES, $DATA$ et BIT représentent les différentes données reçues. Une image est constituée de paquets PES. Les relations entre ces différents ensembles sont explicitées par les variables de notre modèle :

VARIABLES
$Deco, BO, Isize$
INVARIANT
$BO \in BIT \wedge$
$Deco \in IMAGE \leftrightarrow PES \wedge$
$Isize \in IMAGE \nrightarrow \mathbb{N}_1 \wedge$
$(\mathbf{card}(Deco) \leq Dsize)$

A l'aide de ces propriétés de typage, on définit le rôle des différentes variables. La variable *Deco* est le décodeur lui même qui est une relation des *IMAGE* vers les paquets *PES*. Intuitivement, pour une image donnée i, l'ensemble $Deco[\{i\}]$ est l'ensemble des paquets reçus constituant l'image i. De même, la fonction partielle *Isize* associe à chaque image sa taille. La taille des images n'est initialement pas connue par le décodeur, c'est en parcourant l'en-tête des différents paquets que le décodeur détermine la taille des images. Il conserve les différentes valeurs dans une mémoire. Le modèle abstrait ne précise pas comment cette information est obtenue (lecture de l'en-tête, calcul ...) on considérera dans un premier temps qu'elle est obtenue de manière spontanée comme le montre l'événement newImage :

newImage $\widehat{=}$
ANY i, x WHERE
$i \in IMAGE \wedge$
$i \notin dom(Deco) \wedge$
$x \in \mathbb{N}_1$
THEN
$Isize := Isize \lhd\!\!\!- \{i \mapsto x\}$
END

Enfin, la dernière propriété montre que le décodeur n'a pas une mémoire infinie pour stocker les différentes images mais que la taille de sa mémoire est *Dsize*. La variable *BO* indique une erreur abstraite. La variable *BO* permet donc de considérer, dès l'abstraction, le cas de mauvais fonctionnement et la génération d'erreurs (cf événement EBoverflow). Notre premier modèle ne prend pas en compte l'ordre des paquets et permet de s'affranchir de tout éventuel compteur.

La dynamique du système est représentée par les événements du modèle. L'événement produire ci-dessous caractérise la réception d'une partie d'image. La garde de l'événement précise que le décodeur reçoit une partie d'image qu'il n'avait pas déjà reçue et la stocke. Dans ce modèle abstrait le flux est donc parfait sans redondance ni erreur et l'ordre des paquets n'a pas d'importance. On considère un cas idéal et le décodeur peut produire des images régulièrement. Dans un modèle plus concret, les redondances de paquets seront considérées (c'est le rôle du premier buffer TB de supprimer les paquets redondants.).

produire $\widehat{=}$
ANY i, j WHERE
$i \in IMAGE \wedge$
$j \in PES \wedge$
$i \mapsto j \notin Deco \wedge \mathbf{card}(Deco) < Dsize$
THEN
$Deco := Deco \cup \{i \mapsto j\}$
END

L'événement consommer caractérise l'émission d'une image par le décodeur. Dans notre abstraction nous ne nous soucions pas de l'ordre d'émission des images. Une image peut être émise par le décodeur lorsque celui-ci connaît sa taille et qu'elle est effectivement complète (voir garde de l'événement). Dans le modèle abstrait l'émission consiste à supprimer le couple $i \mapsto Deco(i)$ de la fonction partielle *Deco*. De même, l'image étant émise, le décodeur ne conserve pas en mémoire la taille de l'image passée. L'indicateur

BO est positionné de manière aléatoire. En effet, le décodeur étant considéré comme une boite noire, les erreurs indiquées par celui-ci paraissent incompréhensibles dans un premier temps. C'est le raffinement qui permettra d'expliciter ces erreurs par la suite.

consommer $\widehat{=}$
ANY i WHERE
$i \in \mathbf{dom}(Deco) \wedge$
$i \in \mathbf{dom}(Isize) \wedge \mathbf{card}(Deco(i)) = Isize(i)$
THEN
$Deco := \{i\} \lhd Deco \parallel$
$Isize := Isize - \{i \mapsto Isize(i)\} \parallel$
$BO :\in BIT$
END

La détection d'erreur étant le but du décodeur T-STD, notre modèle abstrait contient deux événements chargés de modéliser la détection d'erreurs abstraites. L'événement ExternOverflow représente une perte de partie d'image extérieure au décodeur. Cet événement représente un cas dans lequel le décodeur ne peut pas stocker les informations qu'il reçoit. Ce cas de débordement déclenche une alarme, la variable d'erreur abstraite *BO* étant positionnée à *KO*.

Le second événement InternOverflow modélise une perte d'information interne au décodeur. Dans le cadre du décodeur T-STD, de l'information est perdue lorsque des données sont reçues et ne sont pas stockées. Tant que le décodeur est à sa capacité de stockage maximum, les données reçues ne sont pas conservées et par conséquent perdues. Le décodeur étant considéré comme une boite noire sans visibilité extérieure, le modèle abstrait se contente de signaler la perte d'information ainsi que l'erreur associée.

InternOverflow $\widehat{=}$
ANY i, j WHERE
$i \in IMAGE \wedge$
$j \in PES \wedge$
$i \mapsto j \in Deco$
THEN
$Deco := Deco - \{i \mapsto j\} \parallel$
$BO := KO$
END

ExternOverflow $\widehat{=}$
ANY i, j WHERE
$i \in IMAGE \wedge$
$j \in PES \wedge$
$i \mapsto j \notin Deco$
THEN
$BO := KO$
END

Cette première abstraction permet de décrire simplement le système. Le modèle comporte les grands principes du décodeur :

- les paquets sont reçus dans l'ordre sans redondance,
- des erreurs se produisent et des informations sont perdues.

3.5.2 Raffinement : ajout des différents buffers.

Nous allons maintenant préciser le fonctionnement de notre décodeur en introduisant et en explicitant le fonctionnement de ses différents buffers. Ce raffinement va expliciter la boite noire représentée par la variable *Deco* mais également raffiner l'environnement.

En effet, l'abstraction considérait que le décodeur ne recevait que des parties d'images non déjà reçues (non dupliquées). Dans la réalité les parties d'images peuvent être dupliquées car elles sont encapsulées dans les paquets TS qui peuvent être émis plusieurs fois sur le réseau. Cette répétition de paquets n'est pas en soi une source d'erreur et peut être causée par des perturbations du réseau ou par la réception simultanée du même flux en provenance de deux émetteurs distincts. C'est le rôle du buffer TB de supprimer les paquets TS redondants et de ne transférer aux autres buffers du décodeur que l'information utile (suppression de l'en-tête des paquets TS).

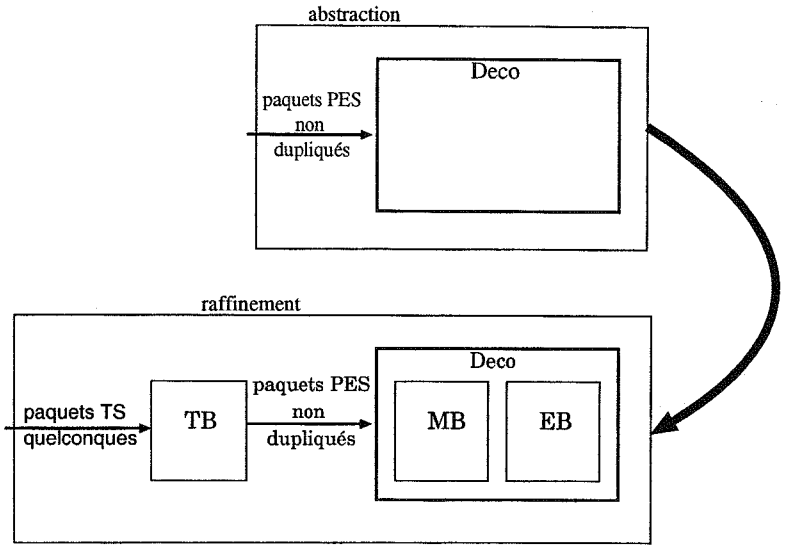


FIG. 3.13 – Raffinement du décodeur abstrait

Deux autres buffers (MB et EB) constituent le décodeur et sont chargés d'extraire l'information utile (les données vidéo) des paquets PES et de stocker ces données jusqu'à la constitution complète de l'image puis son émission. La figure 3.13 montre le lien entre l'abstraction et le raffinement que nous décrivons maintenant.

Afin d'introduire la notion de paquets TS, nous introduisons l'ensemble abstrait TS. Les propriétés invariantes présentées plus loin montrent que notre décodeur abstrait représenté par la variable *Deco* est bien composé des deux buffers MB et EB en précisant que la taille du décodeur est égale à la somme des tailles des deux buffers.

Les traitements du buffer TB se font de manière abstraite par la fonction $TStoPES$ qui simule la suppression des en-têtes TS par le buffer TB lors de la transmission des ces données au buffer MB lui succédant. Par ailleurs, les caractéristiques de la transmission entre buffers (la vitesse de transmission en particulier) varient selon le mode de fonctionnement du décodeur. Il existe deux modes de fonctionnement principaux qui sont donc introduits dans ce raffinement par l'intermédiaire de l'ensemble $METHOD$. Ces deux modes de fonctionnement ont des exigences différentes qui sont détaillées dans la norme DVB. Une des principales différences est l'obligation d'un débit relativement constant et suffisamment important pour la méthode VBV_DELAY.

SETS
TS;
$METHOD = \{LEAK, VBV\_DELAY\}$
CONSTANTS
$TBsize, MBsize, EBsize, TStoPES$
PROPERTIES
$TBsize \in \mathbb{N}_1 \wedge$
$TStoPES \in TS \rightarrow PES \wedge$
$MBsize \in \mathbb{N}_1 \wedge$
$EBsize \in \mathbb{N}_1 \wedge$
$Dsize = MBsize + EBsize$

Les constantes $TBsize$, $MBsize$ et $EBsize$ représentent la taille des buffers composant le décodeur. Le contenu des différents buffers est explicité par les variables *TBbuffer*, *MBbuffer* et *EBbuffer*. Un invariant de collage précise que la variable abstraite *Deco* est raffinée par les deux variables $MBbuffer$

et $EBbuffer$. Enfin, les erreurs détectées par ce modèle de décodeur sont également représentées. Les différentes propriétés invariantes permettent d'établir les relations entre variables.

VARIABLES
$Deco, BO, Isize, method, low\_delay, TBbuffer, MBbuffer, EBbuffer,$
$TB\_buffering\_error, EB\_buffering\_error, MB\_buffering\_error$
INVARIANT
$TBbuffer \in \mathbb{N} \nrightarrow (IMAGE \nrightarrow TS) \wedge$
$\mathbf{card}(TBbuffer) \leq TBsize \wedge$
$TB\_buffering\_error \in BIT \wedge$
$(TB\_buffering\_error = KO \Rightarrow \mathbf{card}(TBbuffer) = TBsize) \wedge$
$MBbuffer \in IMAGE \leftrightarrow PES \wedge$
$\mathbf{card}(MBbuffer) \leq MBsize \wedge$
$MB\_buffering\_error \in BIT \wedge$
$EBbuffer \in IMAGE \leftrightarrow PES \wedge$
$\mathbf{card}(EBbuffer) \leq EBsize \wedge$
$EB\_buffering\_error \in BIT \wedge$
$Deco = MBbuffer \cup EBbuffer \wedge$
$MBbuffer \cap EBbuffer = \emptyset$

L'invariant de collage $Deco = MBbuffer \cup EBbuffer$ est également présenté graphiquement à la figure 3.13 et indique que le décodeur est constitué de deux buffers non redondants ($MBbuffer \cap EBbuffer = \emptyset$). La variable *method* permet de représenter le mode de fonctionnement du décodeur. Deux événements toVBV et toLEAK, non détaillés ici car sans intérêt particulier, permettent de passer d'un mode de fonctionnement à l'autre.

Ce raffinement introduit le buffer TB qui reçoit les paquets TS d'un programme et les désencapsule pour les transmettre au buffer MB. Ce buffer se charge également de la suppression des paquets redondants qui sont stockés mais non envoyés. La réception d'un paquet est symbolisée par l'événement inputTB. Nous considérons que le buffer TB peut désormais recevoir des paquets redondants. Si le buffer TB n'est pas plein, il stocke les paquets TS reçus sous la forme d'une "séquence". Le nouvel événement inputTB modélise la réception d'un paquet TS et son stockage dans le buffer TB.

Par ailleurs, l'événement TBoverflow modélise le débordement du buffer de réception TB. En effet, lorsque le buffer TB est rempli une erreur (débordement) se produit à la réception d'un paquet TS.

inputTB $\widehat{=}$
ANY i, j, x WHERE
$i \in IMAGE \wedge$
$j \in TS \wedge$
$x \in \mathbb{N} \wedge x \notin \mathbf{dom}(TBbuffer) \wedge$
$\mathbf{card}(TBbuffer) < TBsize$
THEN
$TBbuffer := TBbuffer \cup \{x \mapsto \{i \mapsto j\}\}$
END

TBoverflow $\widehat{=}$
SELECT
$\mathbf{card}(TBbuffer) = TBsize$
THEN
$TB\_buffering\_error := KO$
END

Le buffer TB transmet régulièrement des octets de données au buffer MB. A ce niveau de modélisation nous considérons que la transmission des données utiles d'un paquet TS (càd un paquet PES) a lieu de manière immédiate et intégrale. Le buffer TB agit comme une FIFO en transmettant en premier lieu les informations les plus anciennes (voir garde de l'événement $\forall z \cdot (z \in \mathbf{dom}(TBbuffer) \Rightarrow x \leq z)$). L'événement abstrait produire est donc raffiné en une version plus concrète et, comme sa version abstraite, la nouvelle version de produire modélise la réception de parties d'image (des paquets PES : $TStoPES(j)$) non redondantes. En effet, le rôle du buffer TB est de supprimer les informations inutiles et les redondances.

produire $\widehat{=}$
ANY i, j, x WHERE
$i \in IMAGE \wedge$
$j \in TS \wedge$
$x \in \mathbf{dom}(TBbuffer) \wedge$
$\forall z \cdot (z \in \mathbf{dom}(TBbuffer) \Rightarrow x \leq z) \wedge$
$i \mapsto j \in TBbuffer(x) \wedge$
$i \mapsto TStoPES(j) \notin Deco \wedge$
$\mathbf{card}(MBbuffer) < MBsize$
THEN
$MBbuffer := MBbuffer \cup \{i \mapsto TStoPES(j)\} \parallel$
$TBbuffer := TBbuffer - \{x \mapsto \{i \mapsto j\}\} \parallel$
$TB\_buffering\_error := OK \parallel$
$Deco := Deco \cup \{i \mapsto TStoPES(j)\}$
END

L'événement produire modélise la transmission de données entre les buffers TB et MB. Il repositionne de manière systématique l'indicateur *TB_buffering_error* à *OK* ce qui signifie que en ayant émis des données le buffer TB ne se trouve plus dans un état d'overflow et peut à nouveau recevoir et stocker.

L'émission de données du buffer TB peut également, si le buffer MB est plein, provoquer un overflow du buffer MB. L'événement MBoverflow raffine l'événement abstrait externOverflow et modélise la perte des données transférées entre TB et MB lors d'un overflow de ce dernier. La perte de donnée se traduit par la suppression du paquet TS $\{i \mapsto j\}$ du buffer TB (émission des données) et par l'absence d'écriture dans le buffer MB. Les indicateurs d'erreurs sont également positionnés :

MBoverflow $\widehat{=}$
ANY i, j, x WHERE
$i \in IMAGE \wedge$
$j \in TS \wedge$
$x \in \mathbf{dom}(TBbuffer) \wedge$
$x \mapsto \{i \mapsto j\} \in TBbuffer \wedge$
$\forall z \cdot (z \in \mathbf{dom}(TBbuffer) \Rightarrow x \leq z) \wedge$
$i \mapsto TStoPES(j) \notin Deco \wedge$
$\mathbf{card}(MBbuffer) = MBsize$
THEN
$TBbuffer := TBbuffer - \{x \mapsto \{i \mapsto j\}\} \parallel$
$MB\_buffering\_error := KO \parallel$
$TB\_buffering\_error := OK \parallel$
$BO := KO$
END

Enfin, la partie du décodeur représentée par la variable *Deco* n'étant plus une boite noire, des événements détaillent le fonctionnement interne du décodeur et en particulier les différents cas de transmission entre les buffers TB et EB. Le nouvel événement MBtoEB est très semblable à l'événement produire et est un événement interne au décodeur. Il représente une transmission correcte de paquets PES entre les deux buffers. Les autres événements concrets représentent des erreurs de transmissions dues à des débordements des buffers. Les deux événements EBoverflow et EBoverMBunder raffinent tous deux l'événement abstrait InternOverflow. Ces événements caractérisent un overflow du buffer EB.

La méthode B événementielle autorise un tel raffinement : un événement abstrait est scindé en deux événements concrets distincts. Ces deux événements ont des gardes disjointes mais raffinent l'événement abstrait et la garde abstraite est dans les deux cas renforcée par la garde concrète. Avec l'outil actuellement utilisé (la Balbulette), il n'est pas possible de réaliser cette séparation d'un événement abstrait ou même simplement d'ajouter de nouveaux événements. Pour réaliser ce genre de raffinement nous devons

introduire dès l'abstraction deux événements distincts ayant des gardes et des substitutions identiques. Ces deux événements peuvent être raffinés indépendamment et par conséquent "simuler" la séparation d'un événement abstrait en deux événements distincts.

L'événement MBunderflow représente une autre source d'erreur (voir également l'événement EBoverMBunder). En effet, selon la méthode de décodage considérée, le décodeur est supposé recevoir des données de manière plus ou moins continue. En particulier la méthode VBV_DELAY exige une gestion des buffers en flux tendu : les données arrivent à une vitesse régulière sans provoquer ni débordement des buffers, ni pénurie de donnée. Lorsque le décodeur fonctionne suivant ce type de méthode, l'absence de donnée dans le buffer MB est donc considérée comme anormale et ce buffer ne doit jamais être vide.

MBtoEB $\widehat{=}$
ANY i, j WHERE
$i \in IMAGE \wedge$
$j \in PES \wedge$
$i \mapsto j \in MBbuffer \wedge$
$\mathbf{card}(EBbuffer) < EBsize \wedge$
$\mathbf{card}(MBbuffer) > 1$
THEN
$MBbuffer := MBbuffer - \{i \mapsto j\} \parallel$
$EBbuffer := EBbuffer \cup \{i \mapsto j\} \parallel$
$MB\_buffering\_error := OK$
END

EBoverflow $\widehat{=}$
ANY i, j WHERE
$i \in IMAGE \wedge$
$j \in PES \wedge$
$i \mapsto j \in MBbuffer \wedge$
$\mathbf{card}(EBbuffer) = EBsize \wedge$
$\mathbf{card}(MBbuffer) > 1$
THEN
$MBbuffer := MBbuffer - \{i \mapsto j\} \parallel$
$Deco := Deco - \{i \mapsto j\} \parallel$
$MB\_buffering\_error := OK \parallel$
$EB\_buffering\_error := KO \parallel$
$BO := KO$
END

MBunderflow $\widehat{=}$
ANY i, j WHERE
$i \in IMAGE \wedge$
$j \in PES \wedge$
$method = VBV\_DELAY \wedge$
$\{i \mapsto j\} = MBbuffer \wedge$
$\mathbf{card}(EBbuffer) < EBsize \wedge$
THEN
$MBbuffer := MBbuffer - \{i \mapsto j\} \parallel$
$EBbuffer := EBbuffer \cup \{i \mapsto j\} \parallel$
$MB\_buffering\_error := KO$
END

EBoverMBunder $\widehat{=}$
ANY i, j WHERE
$i \in IMAGE \wedge$
$j \in PES \wedge$
$method = VBV\_DELAY \wedge$
$\{i \mapsto j\} = MBbuffer \wedge$
$\mathbf{card}(EBbuffer) = EBsize \wedge$
THEN
$MBbuffer := MBbuffer - \{i \mapsto j\} \parallel$
$Deco := Deco - \{i \mapsto j\} \parallel$
$MB\_buffering\_error := KO \parallel$
$EB\_buffering\_error := KO \parallel$
$BO := KO$
END

3.5.3 T-STD : conclusions

Dans ce dernier modèle, les buffers internes du décodeur T-STD ont été introduits et permettent la description de l'intégralité des erreurs dues à des débordements. D'autres erreurs, dues au non respect de contraintes horaires, sont décrites dans la norme. Nous avons continué le processus de raffinement afin d'inclure ces erreurs à nos modèles mais le résultat reste mitigé en raison de la complexité de description des modèles dû à la complexité de la norme MPEG. Les raffinements suivants introduisent la gestion en FIFO des buffers ainsi que le transfert des données bit à bit.

Les deux premiers modèles permettent d'expliciter suffisamment le problème du T-STD. Les concepteurs électroniques se sont donc appuyés sur ces modèles afin de concevoir une architecture électronique implantant les vérifications normalisées. La méthode B a, à nouveau, joué un rôle pédagogique important dans le cadre de ce développement. En effet, la lecture initiale de la norme ne permettait pas au néophyte

de comprendre aisément le but du décodeur "virtuel" T-STD proposé. Ce décodeur "virtuel" est en réalité un décodeur générique permettant la vérification d'un certain nombre de contraintes afin de s'assurer que les décodeurs réels sont en mesure de décoder correctement le flux qu'ils reçoivent. Nos modèles nous ont permis de comprendre qu'il n'était pas nécessaire de stocker réellement les informations du flux. De ce fait, l'implantation électronique proposée n'utilise pas de file FIFO ou de zone mémoire mais de simples compteurs permettant la simulation des débordements ou des pénuries des différents buffers. Ces compteurs permettent de calculer le taux de remplissage des buffers et une simple mémoire permet la comparaison avec le champs *continuity_counter* de l'en-tête des paquets.

3.6 Bilan général des premières approches

Durant ces premiers contacts avec la conception électronique, nous nous sommes appuyés sur nos partenaires et avons suivi le flot de conception *standard* qu'ils nous ont proposé (voir figure 1 page 3). Le développement de certains modules plus complexes a été immédiatement commencé afin de pouvoir être traité en détail, d'autres parties du système étant considérées comme plus facile à implanter.

3.6.1 Récapitulatif

Les trois études de cas présentées ont été menées afin de permettre l'établissement d'une certaine communication. La première modélisation du paramètre d'acquisition de la synchronisation *TS_sync_loss* nous a permis de présenter la méthode B à nos partenaires. Les premiers modèles, section 2.2.5 page 39, restent très algorithmiques et ne permettent pas d'apporter des solutions ou suggestions intéressantes aux problèmes de la conception de circuits ou plus généralement de SoCs. Ils ont cependant l'intérêt d'être pédagogiques et ont permis des échanges entre les partenaires du projet sur les perspectives d'utilisation d'une méthode formelle. Les modèles présentés dans ce chapitre permettent de montrer la terminaison de la recherche de synchronisation (sous réserve d'un flux permettant d'acquérir la synchronisation) et de préciser cette recherche.

Le second exemple m'a permis d'appréhender les problèmes de conception architecturale. L'étude architecturale ayant été complètement menée et terminée, nous avons pu avoir sous les yeux l'ensemble de la démarche architecturale avant de commencer nos propres développements. Nous avons ainsi pu constater les difficultés engendrées par le passage d'une équation calculant le débit au circuit électronique réalisant effectivement ce calcul. Une notion de "raffinement" informel guide les ingénieurs électroniques qui raisonnent par modèles plus ou moins détaillés. Nous avons donc formalisé le lien entre leurs différents modèles par l'intermédiaire de la notion de raffinement du B événementiel. Le principal désavantage de cette démarche est son caractère *a posteriori* qui impose de refaire le travail déjà fait et qui, en cas de détection d'erreurs, est lourde de conséquences au regard du temps de développement déjà écoulé. Nous avons donc décidé de procéder de manière plus conventionnelle en développant des modèles du système avant sa réalisation et de nous baser sur ces modèles pour la conception architecturale.

Le troisième exemple suit cette démarche. Les développements B ont permis, grâce au raffinement et à un formalisme précis, d'expliciter les buts de l'application ainsi que son fonctionnement. L'architecture finalement produite à partir des modèles se sert de simples compteurs pour simuler les taux de remplissage des différents buffers. Nos modèles ont servi de support à un certain nombre de débats et ont permis de motiver les choix architecturaux finalement faits.

3.6.2 Critique des approches proposées

Le principal inconvénient des développements réalisés est l'isolement. En effet, en suivant le flot de conception standard de développement de SoC nous développons des éléments particuliers sans aucune vision globale du système. L'ajout, à ce niveau, d'une méthode formelle a certes un intérêt mais qui reste limité car ne permettant pas de justifier les choix architecturaux généraux. La cohérence des implantations et la validation de leurs interactions ne sont absolument pas considérées.

Pour illustrer notre propos nous allons détailler un exemple d'interaction entre paramètres qui n'est pas pris en compte par les développements réalisés jusqu'alors. Lors de l'analyse de décodabilité par le

décodeur T-STD, les paquets TS sont reçus les uns après les autres et stockés dans le buffer TB pour traitements. Les mesures et les traitements réalisés dans le cadre de la vérification de la décodabilité n'ont un sens et un intérêt que si le récepteur est préalablement synchronisé. En effet, le découpage du flux en paquet TS ne peut se faire qu'une fois le récepteur correctement synchronisé. Si le flux proposé a un défaut de synchronisation ($TS\_Sync\_loss = KO$), les récepteurs ne peuvent reconstituer les paquets et il est inutile de chercher à vérifier la décodabilité au moyen d'un décodeur virtuel.

De même, si le programme décodé comporte une erreur de continuité (arrivé des paquets TS constituant son flux dans le désordre) alors la définition du T-STD ne permet pas de conclure sur la décodabilité du signal, le buffer EB étant incapable de replacer dans l'ordre les paquets reçus. Une constatation grave, issue de nos modèles et de l'analyse de la norme, est que, en cas de non réception d'un paquet TS constitutif du programme décodé, le décodeur peut afficher une ou plusieurs images incomplètes (il manque les informations contenues dans le paquet perdu) sans indiquer le moindre problème tant que les buffers ne sont pas en débordement. C'est le paramètre *Continuity_count_error* qui va réagir à ce type de malformation du flux. L'analyse de décodabilité n'a donc bien de sens que si le flux présenté est suffisamment bien construit et a passé l'épreuve de vérifications plus générales.

Ce genre de considérations simples nous a permis de comprendre les limitations du modèle *standard* de conception de SoC. De plus, la nature de l'application étudiée, l'analyse de la qualité de transmission, implique une structuration forte des traitements. Nous avons donc compris que nous devions modéliser l'application dans son ensemble. En effet, en suivant le schéma de conception orienté *modèle* et en considérant l'application dans sa globalité, nous pouvons déterminer clairement l'ordonnancement et les interactions des diverses vérifications décrites (sans ordre très précis) dans la norme. Nous avons donc repris notre travail de modélisation afin d'inclure l'intégralité de la norme [55].

Chapitre 4

Etude de cas : globalisation

La nécessité d'avoir une vue plus globale de l'application s'est relativement vite imposée, nous avons décidé de modéliser l'intégralité de la norme DVB. Le document initial n'étant que très peu structuré, une étude poussée de la norme a été nécessaire.

Le raffinement va permettre de structurer le cahier des charges (ici la norme) et va permettre également une hiérarchisation des différents paramètres d'analyse. En partant d'une vue abstraite (donc incomplète) du système, nous pouvons nous focaliser sur le coeur du système puis, avec les raffinements, introduire les détails d'implantation ou les aspects plus secondaires du système.

Hiérarchiser les différents paramètres de la norme demande un haut niveau d'expertise que nous avons atteint grâce à l'étude de la norme et au raffinement. Les débats suscités par les modèles, que nous allons présenter par la suite, nous ont également grandement fait progresser dans la compréhension de la norme.

4.1 Développement d'une hiérarchie incrémentale

Nous allons présenter la série de raffinements qui a permis de modéliser l'ensemble du système de métrologie considéré. Chaque raffinement ajoute des détails sous la forme de nouveaux paramètres ou bien de précisions sur l'environnement du système. Les relations entre les différents paramètres calculés par l'outil de mesure sont également raffinées et précisées en fonction du niveau de détail du modèle courant.

L'ensemble du processus de raffinement représentant 7 modèles (1 modèle abstrait et 6 raffinements successifs), nous ne présenterons pas ici l'intégralité du processus. Les différents paramètres sont introduits au fur et à mesure des raffinements successifs qui permettent de se focaliser sur des points particuliers du système. Nous nous focaliserons sur quatre modèles particulièrement importants à nos yeux :

- l'abstraction initiale qui capture l'essence même du système ;
- le premier raffinement qui introduit l'évaluation des premiers paramètres ;
- le second raffinement qui illustre la hiérarchie établie entre les paramètres ;
- le dernier modèle qui prend en compte l'ensemble des paramètres TS de la norme TR 101 290 [55].

Le développement suivant a été réalisé à partir de la norme DVB-T [55]. Celle-ci préconise un certain nombre d'indicateurs évaluant l'état de transmission du réseau et propose 3 niveaux de priorité pour ces indicateurs. Le niveau de priorité maximum proposé par la norme impose 6 vérifications principales alors que le second niveau décrit 7 paramètres d'évaluation et le troisième et dernier niveau en décrit une vingtaine.

N'étant pas expert du domaine, nous avons dû réaliser les modèles avec la compréhension du système que nous avions et nous sommes appuyés sur cette première base de travail. Nous avons donc établi un modèle abstrait du système qui, d'après notre compréhension, reflétait son fonctionnement. Cette première abstraction se base essentiellement sur le fait qu'il existe différentes natures de données. Un récepteur doit d'abord réceptionner des "informations de gestion" avant de pouvoir traiter des données réelles. Partant de cette première abstraction, nous avons dérivé une série de modèle de plus en plus précis en introduisant les détails du système suivant notre compréhension et le niveau de détails. Chaque raffinement introduit

des nouveaux paramètres (les plus prioritaires de préférence) comme des feuilles d'une hiérarchie, sauf exceptions. Les différents modèles ont ensuite été discutés avec des experts de la DVB-T afin de vérifier que notre compréhension du système était correcte. En procédant de cette façon nous avons affiné les différents niveaux de priorité (ou d'importance) proposés par la norme et avons détecté une incohérence : un paramètre du second niveau (*Transport_error*) est en réalité plus important que le laisse présager le classement initial.

4.1.1 Abstraction initiale : le coeur du système

Le premier modèle abstrait de notre système est basé sur la compréhension globale de la norme après une première étude de celle-ci. Ce premier modèle doit poser les bases de fonctionnement du système et de son interaction avec l'environnement, la méthode B événementielle étant basée sur l'hypothèse du mode clos. La caractéristique principale du système est, à notre avis, l'existence de différentes natures de paquets. En effet, pour pouvoir commencer son analyse, un outil de monitoring doit comme tout récepteur reconstruire les tables PID à l'aide des paquets dit "système". L'existence de plusieurs familles de paquets est donc le point de départ de ce modèle qui représente le système comme un consommateur de paquets.

Constantes et ensembles

La liste des constantes et ensembles définis dans cette première machine est présentée dans le tableau ci-dessous avec une définition succincte. Les propriétés de ces constantes vont permettre de caractériser les relations entre environnement et système.

Constante ou ensemble	Définition
TS_PACKET	Ensemble des paquets possibles (bien construits comme mal construits).
$TYPE$	Ensemble des types de paquets connus par le système.
PAT	Type de paquet (élément de l'ensemble TYPE).
PMT	Type de paquet (élément de l'ensemble TYPE).
PaT	Sous-ensemble de paquets PAT bien construits.
PmT	Sous-ensemble de paquet PMT bien construits.
$getType$	Fonction donnant le type d'un paquet déduit (élément de TYPE) par le système.
$getPID$	Fonction de lecture du PID de l'en-tête d'un paquet.

L'ensemble TS_PACKET est l'ensemble des paquets TS existants. Cet ensemble contient donc les paquets correctement formés comme les paquets erronés. Les deux sous-ensembles PaT et PmT représentent les ensembles de paquets PAT et PMT correctement construits (en-tête et contenu corrects). L'ensemble abstrait $TYPE$ représente l'ensemble des types de paquets reconnus par le système suite à une analyse de l'en-tête du paquet. Un récepteur déduit le type d'un paquet par la lecture d'un certain nombre de champs et leurs comparaisons avec des valeurs prédéfinies. La lecture de l'en-tête et la déduction du type du paquet sont modélisées ici par la fonction $getType$. Cette fonction symbolise les déductions de type réalisées par le système contrairement aux ensembles PaT, PmT... qui représentent le type réel "absolu" des paquets. Dans ce modèle abstrait, le système ne reconnaît que deux types de paquets, les PAT et les PMT. Les propriétés suivantes définissent l'environnement du système et détaillent les interactions entre différentes natures de paquets.

$$
\begin{array}{l}
PaT \subseteq TS\_PACKET \wedge \\
PmT \subseteq TS\_PACKET \wedge \\
PaT \cap PmT = \emptyset \wedge \\
PaT \neq \emptyset \wedge \\
PmT \neq \emptyset \wedge \\
PAT \in TYPE \wedge \\
PMT \in TYPE \wedge \\
getType \in TS\_PACKET \rightarrow TYPE \wedge \\
\forall p.(p \in PaT \Rightarrow getType(p) = PAT) \wedge \\
\forall p.(p \in PmT \Rightarrow getType(p) = PMT) \wedge \\
\forall p.(p \in TS\_PACKET \wedge p \notin PaT \wedge p \notin PmT \\
\Rightarrow getType(p) \neq PAT \wedge getType(p) \neq PMT) \wedge \\
getValue \in TS\_PACKET \rightarrow (BITS \rightarrow DATA) \wedge \\
getPID \in TS\_PACKET \rightarrow PID
\end{array}
$$

Les propriétés sur les deux sous-ensembles PaT et PmT permettent de préciser que l'intersection de ces deux ensembles est vide (un paquet bien construit ne peut-être une PAT et une PMT à la fois). La fonction *getType* modélise la perception des données par le système. Elle représente l'analyse de l'en-tête du paquet courant par le système et la déduction faite par celui-ci sur le type du paquet courant. De manière identique, la fonction $getPID$ permet la lecture dans l'en-tête du paquet de la valeur du PID. Les propriétés précédentes indiquent que l'analyse de l'en-tête des paquets ($getPID$ et *getType*) se fait sans erreur et que les résultats donnés par cette analyse sont cohérents avec la réalité des choses : un paquet PAT est bien reconnu comme tel.

Variables

Les premières variables définies sont répertoriées dans ce tableau :

Variable	Définition
flow	Le flux traité par le système.
table	Les tables "système" construites par l'outil.
current	Le paquet courant.

Les tables contenues dans la variable *table* sont construites à l'aide des paquets PAT et PMT. Un paquet PAT est un paquet particulier ayant un PID (*Packet IDentifier*) réservé et fixe. Comme déjà énoncé précédemment, ce paquet contient la liste des PIDs des différentes tables PMTs. Pour pouvoir reconstruire le signal, chaque programme a sa propre PMT qui liste simplement les PIDs des différents types de paquets associés au programme. Le premier invariant est essentiellement composé de typage :

$$
\begin{array}{l}
flow \subseteq TS\_PACKET \wedge \\
table \in PID \nrightarrow (PID \nrightarrow TS\_PACKET) \wedge \\
current \in flow
\end{array}
$$

La variable $flow$ représente le flux que l'on est en train de traiter et également un sous-ensemble de tous les paquets existants. La variable *table* représente les tables extraites lors de la réception de paquets PAT et PMT. La variable *table* est une fonction partielle ($\nrightarrow$) qui permet d'associer les différents PID aux différents paquets. Enfin, la variable *current* représente le paquet courant (toujours un paquet du flux). Ce paquet est le paquet actuellement analysé. La figure 4.1 résume la situation définie par les constantes et les variables de ce premier modèle.

Evénements abstraits

La liste des événements définis dans cette abstraction est assez courte. En effet, les seuls paquets traités sont les paquets "système" de type PAT et PMT.

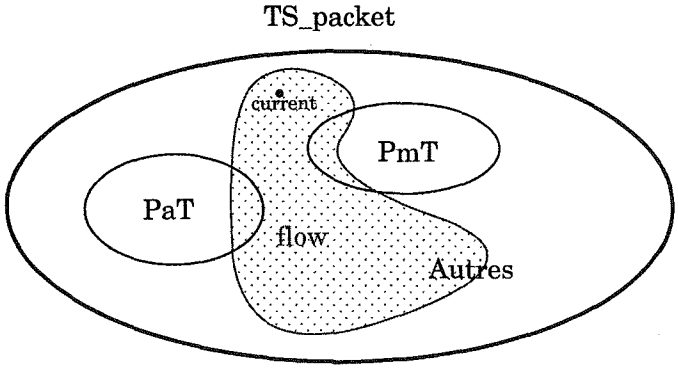


FIG. 4.1 – Relation entre types de paquets et variables abstraites

Evénement	Définition
uploadedPAT	Mise à jour des tables sur réception d'un paquet PAT.
uploadedPMT	Mise à jour des tables sur réception d'un paquet PMT.
nosystem	Réception d'un paquet quelconque (non PAT et non PMT).
trap	Certains paquets ne sont pas traités.

Les événements définis dans cette première machine abstraite représentent pour nous l'essence même du système à modéliser : l'aspect important du système étant l'existence de paquets particuliers nécessaires à la reconstruction du signal. Nous détaillons ci-dessous les événements listés dans le tableau en précisant leurs motivations.

uploadedPAT $\widehat{=}$
SELECT
$getType(current) = PAT$
THEN
$table :\in PID \nrightarrow (PID \nrightarrow TS\_PACKET) \parallel$
$current :\in flow$
END

L'événement uploadedPAT représente la réception d'un paquet analysé (*getType*) comme une PAT. La conséquence (partie action de l'événement) est la mise à jour des tables construites par le système. Cette opération est décrite de manière abstraite à l'aide de l'axiome du choix. En effet, à un tel niveau d'abstraction nous ne souhaitons pas décrire précisément l'extraction de tables d'un paquet PAT. De ce fait, l'axiome du choix permet de signifier une mise à jour des tables (la variable *table* est modifiée) sans pour autant détailler le mécanisme de modification lui-même : le symbole $:\in$ exprime un choix indéterministe d'un élément dans un ensemble défini. Enfin, la réaffectation de la variable *current* représente l'arrivée d'un nouveau paquet du flux.

uploadedPMT $\widehat{=}$
SELECT
$getType(current) = PMT$
THEN
$table :\in PID \nrightarrow (PID \nrightarrow TS\_PACKET) \parallel$
$current :\in flow$
END

L'événement uploadedPMT est l'équivalent de uploadedPAT mais dans le cas de la réception d'un paquet analysé comme PMT.

nosystem $\widehat{=}$
SELECT
$getType(current) \neq PAT \wedge$
$getType(current) \neq PMT$
THEN
$current :\in flow$
END

trap $\widehat{=}$
BEGIN
$current :\in flow$
END

L'événement nosystem représente la réception et le traitement d'un paquet n'étant ni une PAT, ni une PMT. A ce niveau de détail, aucun calcul n'est encore associé aux paquets "quelconques", l'événement se contente de réaffecter la variable *current* càd d'attendre le paquet suivant.

L'événement trap représente la perte d'un paquet. Cet événement attrape des paquets (sans souci de leur nature) et passe au paquet suivant empêchant tout traitement. Dans un prochain raffinement cet événement prendra son sens, la véritable motivation de cet événement n'étant pas encore perceptible à ce niveau d'abstraction. L'introduction de cet événement dans ce niveau d'abstraction est une anticipation sur la suite des développements.

Les deux événements nosystem et trap sont des anticipations des futurs raffinements et du comportement du système en cas de réception de paquets non-systèmes.

4.1.2 Premier raffinement : SYS1

Le premier raffinement de notre modèle nous permet d'ajouter les premiers paramètres définis par la norme. En effet, ce raffinement détaille suffisamment le système et son environnement pour permettre la modélisation de calculs et de vérifications. Nous ajoutons à ce modèle les paramètres reposant sur ces calculs et vérifications sous forme de variables B événementielles. Les paramètres ajoutés sont choisis de manière "arbitraire" en fonction du niveau de détail du modèle.

Constantes et ensembles

Les nouvelles constantes utilisées par ce raffinement sont présentées dans le tableau ci-dessous.

Constante ou ensemble	Définition
VALUE	L'ensemble des 3 valeurs prises par les paramètres.
Delay	Le temps s'écoulant entre deux réceptions de paquet.
DelayPAT	Le temps maximum autorisé entre deux paquets PAT (0.5s dans la norme)
DelayPMT	Le temps maximum autorisé entre deux paquets PMT (0.5s dans la norme)
PAT_PID	La valeur du PID d'une PAT.
PAT_Scr	valeur du champ Scrambling pour une PAT
PMT_Scr	valeur du champ Scrambling pour une PMT
Scrambling	Fonction de lecture de l'en-tête (champ scrambling)

$VALUE = \{OK, KO, IND\} \wedge$
$Delay \in \mathbb{N}_1 \wedge$
$DelayPAT \in \mathbb{N}_1 \wedge$
$DelayPMT \in \mathbb{N}_1 \wedge$
$PAT\_PID \in PID \wedge$
$PAT\_Scr \in \mathbb{N} \wedge$
$PMT\_Scr \in \mathbb{N} \wedge$
$\forall p.(p \in PaT \Rightarrow getPID(p) = PAT\_PID) \wedge$
$Scrambling \in TS\_PACKET \rightarrow \mathbb{N}$

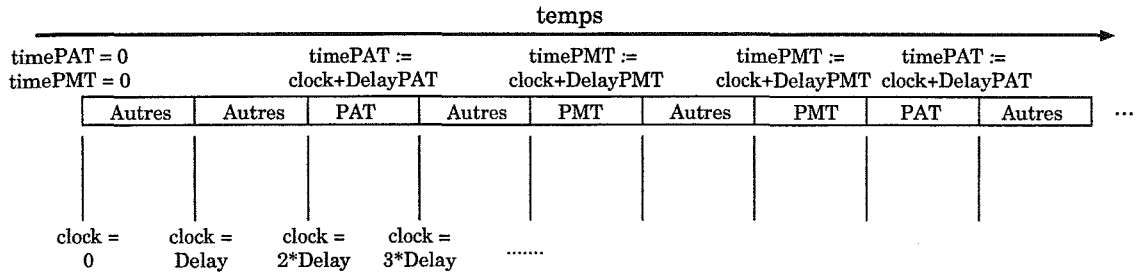


FIG. 4.2 – Séquencement du temps

Les constantes *Delay*, *DelayPAT* et *DelayPMT* sont des entiers chargés de représenter une valeur *temporelle*. La constante *Delay* représente le temps s'écoulant entre la réception de deux paquets successifs. Les variables *DelayPAT* et *DelayPMT* représentent le délai maximum autorisé entre deux paquets de même nature (PAT, PMT) La constante *PAT_PID* est la valeur fixe du PID d'un paquet de type PAT. La propriété quantifiée signifie que la fonction *getPID* du système réalise correctement la lecture et que tous les paquets de type PAT ont bien un PID de PAT. Enfin, la fonction *Scrambling* est une simple fonction de lecture de l'en-tête des paquets. Elle représente la lecture d'un champ ayant une valeur particulière selon la nature du paquet. Cette valeur doit être *PAT_Scr* pour un paquet de type PAT et *PMT_Scr* pour un paquet de type PMT. Ce champ permet la vérification de la cohérence des informations contenues dans l'en-tête du paquet, en particulier au niveau des PIDs. Notre modélisation considère que le champ *Scrambling* de l'en-tête est un entier. Dans la réalité, ce champ contient effectivement un entier mais de taille restreinte (la longueur du champ n'est que de 4 octets). Dans le cadre de notre étude ce point particulier n'a pas une importance fondamentale. La signification des éléments de l'ensemble *VALUE* de l'en-tête du modèle sera précisée par la suite.

Variables

Variable	Définition
Unreferenced_PID	Vérifie l'existence du PID du paquet courant dans les tables construites.
PAT_error	Vérifie les temps de réception entre deux paquets PAT successifs (cas 1).
PMT_error	Vérifie les temps de réception entre deux paquets PMT successifs (cas 1).
PAT_error_crypt	Vérifie la bonne construction (PID et Scrambling) d'une PAT (cas 2).
PMT_error_crypt	Vérifie la bonne construction (PID et Scrambling) d'une PMT (cas 2).
PAT_error_struct	Vérifie la bonne construction (PID et table_id) d'une PAT (cas 3).
PMT_error_struct	Vérifie la bonne construction d'une PMT (PID et table_id) (cas 3).
clock	L'horloge locale du système.
timeTS	L'instant d'arrivée du prochain paquet.
timePAT	Date limite d'arrivée de la prochaine PAT dans le flux.
timePMT	Date limite d'arrivée de la prochaine PMT dans le flux.

On peut représenter temporellement l'arrivée des paquets pour clarifier la signification des variables temporelles. La figure 4.2 illustre le principe que nous avons mis en oeuvre dans nos modèles afin d'exprimer la notion de temps.

Certaines variables définies ici sont utilisées pour représenter la valeur des paramètres d'évaluation préconisés par la norme. Ces variables portent le même nom que les paramètres. Les autres variables sont des variables "locales" du système : elles sont utiles à la modélisation ou représentent un composant du système (comme de la mémoire, par exemple). Nous avons introduit dans nos modèles des *sous-paramètres* en décomposant différents cas d'erreurs de paramètres. Le tableau précédent illustre cet état de fait ; les variables *PAT_error_crypt* ou encore *PAT_error_Scr* représentent des vérifications réalisées dans le cadre du paramètre *PAT_error*. Cependant la nature profondément différente de ces vérifications nous a amené à les distinguer.

Une variable chargée de représenter l'évaluation d'un paramètre prend sa valeur dans l'ensemble $VALUE$. Les trois éléments constituant cet ensemble ont des sémantiques bien distinctes et permettent de raisonner sur les rapports entre paramètres :

- $Param_A = OK$ signifie que le paramètre associé à la variable $Param_A$ a été évalué et que le résultat de l'évaluation est correct.
- $Param_A = KO$ signifie que le paramètre associé à la variable $Param_A$ a été évalué et que le résultat de l'évaluation est incorrect.
- $Param_A = IND$ signifie que le paramètre associé à la variable $Param_A$ n'a pas été évalué pour une raison non précisée (en cours d'évaluation, non évalué car non adéquat, etc...)

Nous allons maintenant détailler l'invariant de ce modèle qui permet de considérer les premiers paramètres de la norme.

$$
\begin{array}{l}
Unreferenced\_PID \in VALUE \wedge \\
PAT\_error \in VALUE \wedge \\
PMT\_error \in VALUE \wedge \\
PAT\_error\_struct \in VALUE \wedge \\
PMT\_error\_struct \in VALUE \wedge \\
PAT\_error\_crypt \in VALUE \wedge \\
PMT\_error\_crypt \in VALUE \wedge \\
clock \in \mathbb{N} \wedge \\
timeTS \in \mathbb{N} \wedge \\
timePAT \in \mathbb{N} \wedge \\
timePMT \in \mathbb{N} \wedge \\
timeTS \geq clock \wedge \\
(getType(current) = PAT \Rightarrow Unreferenced\_PID = IND) \wedge \\
(getType(current) = PMT \Rightarrow Unreferenced\_PID = IND) \wedge \\
PAT\_PID \notin \mathbf{dom}(table)
\end{array}
$$

Seules les trois dernières propriétés sont particulièrement intéressantes, les autres n'étant que des propriétés de typage. La dernière proposition est une hypothèse assez forte mais néanmoins réaliste signifiant que le PID réservé aux PAT n'est pas présent dans la liste des PIDs (*table*) des PMTs. En effet, dans le cas contraire, le système risque de confondre une PAT avec une PMT de manière systématique[9]. De ce fait, un décodeur risque d'être incapable de retrouver dans le flux les différents paquets nécessaires à la reconstruction du signal.

Les deux autres propriétés précédentes expriment le fait que le paramètre $Unreferenced\_PID$ n'a de sens que si l'on évalue un paquet de "données"(donc ni une PAT et ni une PMT). Les propriétés invariantes suivantes sont nettement plus intéressantes :

$$
\begin{array}{l}
(PMT\_error = OK \Rightarrow PAT\_error = OK) \wedge \\
(Unreferenced\_PID \neq IND \Rightarrow PMT\_error = OK) \wedge \\
(Unreferenced\_PID \neq IND \Rightarrow PAT\_error = OK) \wedge \\
(PAT\_error\_crypt \neq IND \Rightarrow PAT\_error\_struct = OK) \wedge \\
(PMT\_error\_crypt \neq IND \Rightarrow PMT\_error\_struct = OK)
\end{array}
$$

Ces propriétés permettent de hiérarchiser les paramètres et elles permettront d'ordonnancer les calculs. On définit donc une relation de *dépendance* entre les paramètres :

Définition 1 *On dira que* $Param_A$ *dépend de* $Param_B$ *(*$Param_A \prec Param_B$*) quand les deux variables* $Param_A$ *et* $Param_B$ *sont liées par l'une des deux propriétés équivalentes suivantes :*
$Param_A \neq IND \Rightarrow Param_B = OK$ *ou*
$Param_B \neq OK \Rightarrow Param_A = IND$

[9]La norme préconise la vérification de la cohérence des paquets ayant un PID de PAT mais ceci ne doit pas être une erreur systématique.

C'est une connaissance approfondie du domaine qui permet de dégager les diverses relations de dépendance. Il est, par exemple, évident que vérifier l'appartenance aux tables systèmes d'un PID donné n'a d'intérêt que si ces tables sont correctement construites : $Unreferenced\_PID \prec PMT\_error$, $Unreferenced\_PID \prec PAT\_error$. D'après la première propriété, on ne peut vérifier la cohérence d'une sous-table que si la table la référençant est elle-même cohérente : $PMT\_error \prec PAT\_error$.

Afin de permettre au lecteur de mieux comprendre le travail d'analyse qui a été fourni, nous citons ci-dessous une partie du texte, issu de la norme TR 101 290 [55], qui présente les différents paramètres que nous ajoutons dans ce modèle :

PAT_error :
The Program Association Table (PAT), which only appears in PID 0x0000 packets, tells the decoder, what programs are in the TS and points to the Program Map Tables (PMT) which in turn point to the component video, audio and data streams that make up the program. If the PAT is missing then the decoder can do nothing, no programs is decodable.

PID 0x0000 does not occur at least every 0.5s. a PID 0x0000 does not contain a table_id 0x00 (ie a PAT). Scrambling_control_field is not 00 for PID 0x0000. Nothing other than a PAT should be contained in a PID 0x0000.

PMT_error :
The Program Association Table (PAT) tells the decoder how many programs there are in thye stream and points to the PMTs which contain the information where the parts for any given event can be found. ...

Unreferenced_PID :
Each non-private program data stream should have its PID listed in the PMTs. ...

Dans la norme, les paramètres sont présentés selon trois niveaux d'importance mais il n'existe pas de réel classement des paramètres à l'intérieur d'un même niveau. Il n'est pas fait mention d'un quelconque lien de dépendance entre les différents paramètres ou d'un quelconque algorithme d'implantation. De ce fait, la découverte des dépendances entre différents paramètres de la norme est une question de *bon sens* ou plutôt d'expérience et de connaissance du domaine. N'étant pas experts en DVB, nos modèles ont permis de présenter de manière formelle nos conclusions aux experts en s'affranchissant du problème de vocabulaire.

Hiérarchie incrémentale

La figure 4.3 illustre graphiquement les propriétés énoncées dans l'invariant de dépendance. Les flèches reliant les tâches les unes aux autres sont une illustration graphique de la relation de dépendance définie précédemment et permettent une visualisation rapide de l'invariant du modèle et des liens entre tâches. En effet, durant le projet EQUAST, les illustrations étaient plus parlantes que les invariants des modèles et nous avons pris l'habitude d'illustrer nos modèles par un graphe des relations de dépendance entre les paramètres considérés.

Il est important de souligner le rôle pédagogique de ces illustrations. En effet, c'est seulement suite à ces travaux et à la présentation des différents schémas de dépendance que nous présentons dans ce chapitre que les partenaires industriels, experts DVB du projet ont réagi à nos modèles. Des débats s'en sont suivis indiquant des compréhensions de la norme différentes selon les experts. Ces débats, avec l'aide des modèles, ont finalement débouché sur des accords entre partenaires et sur la "re-découverte" du sens de certains paramètres et de leur intérêt.

Pour le moment, on constate l'existence d'une *forêt* de dépendances, séparant en trois familles les paramètres considérés dans le modèle. D'un côté une chaîne de paramètres relatifs aux données et demandant une cohérence du système ($Unreferenced\_PID$, PMT_error et PAT_error) et de l'autre, les paramètres spécifiques à la réception de paquets "système", vérifiant la bonne construction du paquet traité.

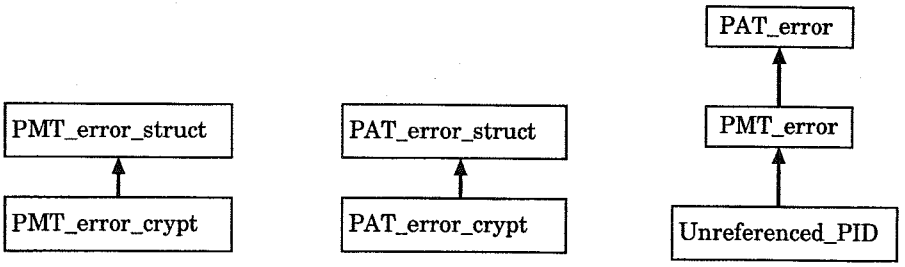


FIG. 4.3 – Hiérarchie incrémentale : système **SYS1**

Nouveaux événements et événements raffinés

testPIDKO $\widehat{=}$
SELECT
$getType(current) \neq PAT \wedge$
$getType(current) \neq PMT \wedge$
$PMT\_error = OK \wedge$
$getPID(current) \notin \mathbf{dom}(\mathbf{ran}(table))$
THEN
$Unreferenced\_PID := KO$
END

testPIDOK $\widehat{=}$
SELECT
$getType(current) \neq PAT \wedge$
$getType(current) \neq PMT \wedge$
$PMT\_error = OK \wedge$
$getPID(current) \in \mathbf{dom}(\mathbf{ran}(table))$
THEN
$Unreferenced\_PID := OK$
END

Les événements testPIDKO et testPIDOK modélisent la vérification du PID d'un paquet de données. Si le PID du paquet concerné est bien présent dans les tables (càd qu'il existe une sous-table contenant ce PID) alors aucune erreur n'est déclenchée. Dans le cas contraire, le paramètre *Unreferenced_PID* est positionné à *KO* signifiant une erreur.

testPAT $\widehat{=}$
SELECT
$clock > timePAT \wedge$
$PAT\_error \neq KO \wedge$
$getType(current) \neq PAT$
THEN
$PAT\_error := KO \parallel$
$PMT\_error := IND \parallel$
$Unreferenced\_PID := IND$
END

uploadedPAT $\widehat{=}$
ANY
t
WHERE
$getType(current) = PAT \wedge$
$t \in PID \nrightarrow (PID \nrightarrow TS\_PACKET) \wedge$
$PAT\_PID \notin \mathbf{dom}(t)$
THEN
$table := t \parallel$
$current :\in flow \parallel$
$timePAT := clock + DelayPAT \parallel$
$timeTS := timeTS + Delay \parallel$
$PAT\_error := OK \parallel$
$PAT\_error\_struct := IND \parallel$
$PAT\_error\_crypt := IND$
END

Les deux événements précédents représentent la gestion des paquets PAT et, en particulier, leur régularité de réception. Deux paquets PAT successifs doivent être reçus dans un certain délai, sinon l'événement testPAT est déclenché. Il positionne alors le paramètre *PAT_error* à *KO*, afin de signaler une erreur. Du fait des dépendances entre paramètres, l'erreur détectée sur *PAT_error* annule la valeur de tous les paramètres dépendants tels *PMT_error* et *Unreferenced_PID* (cf invariant). Pour les mêmes raisons que précédemment, le paramètre *PMT_error* est invalidé en cas d'erreur du paramètre *PAT_error*. Ces deux paramètres évaluent le flux et ne se contentent pas de réaliser un certain nombre de tests sur

les paquets. Ces deux paramètres vérifient la réception, à intervalles réguliers, de paquets systèmes, la reconnaissance des paquets PMT n'étant possible que lorsque la table PAT est correcte. L'événement uploadedPAT modélise la réception puis la mise à jour des tables lors d'une réception de paquet PAT. Cet événement réalise la réinitialisation des paramètres travaillant sur le paquet courant. En effet, au regard de la hiérarchie incrémentale, l'événement uploadedPAT est l'un des derniers événements exécutés pour des paquets PAT. Le système doit cependant être capable de traiter le paquet suivant (*current* :∈ *flow*) et les différents paramètres évaluant un paquet doivent être réinitialisés.

testPATKO $\widehat{=}$
SELECT
$getType(current) \neq PAT \wedge$
$getPID(current) = PAT\_PID$
THEN
$PAT\_error\_struct := KO \|$
$PAT\_error\_crypt := IND$
END

testPATOK $\widehat{=}$
SELECT
$getType(current) = PAT \wedge$
$getPID(current) = PAT\_PID$
THEN
$PAT\_error\_struct := OK$
END

Le couple d'événements testPATKO et testPATOK symbolise les deux cas de vérification de cohérence sur l'en-tête d'un paquet de type PAT et positionne l'indicateur correspondant.

testPATcryptOK $\widehat{=}$
SELECT
$PAT\_error\_struct = OK \wedge$
$PAT\_error\_crypt = IND \wedge$
$Scrambling(current) = PAT\_Scr$
THEN
$PAT\_error\_crypt := OK$
END

testPATcryptKO $\widehat{=}$
SELECT
$PAT\_error\_struct = OK \wedge$
$PAT\_error\_crypt = IND \wedge$
$Scrambling(current) \neq PAT\_Scr$
THEN
$PAT\_error\_crypt := KO$
END

Comme précédemment, ce couple d'événements permet de vérifier la cohérence de l'en-tête d'un paquet PAT. En effet, certains champs doivent avoir des valeurs particulières lors de réception de paquets particuliers comme les PATs. Ces champs, constituants de l'en-tête du paquet, permettent de détecter les paquets mal-construits et de les écarter avant de provoquer des perturbations plus graves sur la reconstruction du signal. Ainsi, on peut imaginer qu'un paquet de données a un PID de PAT suite à une perturbation du réseau en cours de transport. Pour ne pas provoquer une perte d'image à la réception, les récepteurs doivent être capables de détecter que ce paquet ayant un PID de PAT n'est pas une PAT. Les événements précédents représentent donc la vérification du champ *Scrambling* qui doit avoir une valeur spécifique dans le cas d'un PAT.

testPMT $\widehat{=}$
SELECT
$clock > timePMT \wedge$
$PMT\_error \neq KO \wedge$
$getType(current) \neq PMT$
THEN
$PMT\_error := KO \|$
$Unreferenced\_PID := IND$
END

uploadedPMT $\widehat{=}$
ANY
t
WHERE
$t \in PID \nrightarrow (PID \nrightarrow TS\_PACKET) \wedge$
$PAT\_PID \notin \mathbf{dom}(t) \wedge$
$getType(current) = PMT \wedge$
$getPID(current) \in \mathbf{dom}(table) \wedge$
$PAT\_error = OK$
THEN
$table := t \|$
$current :\in flow \|$
$timePMT := clock + DelayPMT \|$
$timeTS := timeTS + Delay \|$
$PMT\_error := OK \|$
$PMT\_error\_struct := IND \|$
$PMT\_error\_crypt := IND$
END

Les deux événements testPMT et uploadedPMT ont la même fonction que les événements testPAT et uploadedPAT mais pour des paquets PMT. Du fait de la dépendance existant entre PATs et PMTs, une mise à jour des tables lors de la réception d'un paquet PMT ne peut se faire que s'il n'existe pas de *PAT_error*. La garde de l'événement est donc renforcée pour prendre en compte cette nouvelle contrainte. Comme le couple d'événements testPAT et uploadedPMT, ce couple d'événements doit réinitialiser le système afin de pouvoir correctement considérer le paquet suivant ($Param_X := IND$).

testPMTKO $\widehat{=}$
SELECT
$getType(current) \neq PMT \wedge$
$getPID(current) \in \mathbf{dom}(table)$
THEN
$PMT\_error\_struct := KO \|$
$PMT\_error\_crypt := IND$
END

testPMTOK $\widehat{=}$
SELECT
$getType(current) = PMT \wedge$
$getPID(current) \in \mathbf{dom}(table)$
THEN
$PMT\_error\_struct := OK$
END

Le couple d'événements précédent a un rôle similaire aux événements testPATKO et testPATOK. Les deux couples d'événements manipulent des paquets de natures différentes.

testPMTcryptOK $\widehat{=}$
SELECT
$PMT\_error\_struct = OK \wedge$
$PMT\_error\_crypt = IND \wedge$
$Scrambling(current) = PMT\_Scr$
THEN
$PMT\_error\_crypt := OK$
END

testPMTcryptKO $\widehat{=}$
SELECT
$PMT\_error\_struct = OK \wedge$
$PMT\_error\_crypt = IND \wedge$
$Scrambling(current) \neq PMT\_Scr$
THEN
$PMT\_error\_crypt := KO$
END

De la même manière que précédemment, ces deux nouveaux événements sont similaires aux événements testPATcryptOK et testPATcryptKO au type des données manipulées près.

testPCR3 $\widehat{=}$
SELECT
$current \in$ flow
THEN
$clock \in$: 0..timeTS
END

time $\widehat{=}$
SELECT
$clock <$ timeTS
THEN
$clock :=$ timeTS
END

Les événements précédents représentent la gestion du temps. L'événement time réalise la mise à jour de l'horloge par saut à la réception du nouveau paquet. L'horloge abstraite est donc figée durant le temps de traitement du paquet courant puis s'incrémente à la réception du paquet suivant. L'événement testPCR3 est un événement abstrait indiquant que, pour des raisons encore non précisées, la valeur de l'horloge locale au système peut être remise à jour ou corrigée.

errorsync $\widehat{=}$
SELECT
$Unreferenced\_PID = IND$
THEN
$PAT\_error :=$ IND ||
$PMT\_error :=$ IND
END

L'événement errorsync est également un événement abstrait qui prendra tout son sens une fois le mécanisme de synchronisation introduit. Le mécanisme en question est introduit dans le prochain raffinement. Pour l'instant, cet événement indique que des paquets ne peuvent être correctement traiter.

nosystem $\widehat{=}$
SELECT
$getType(current) \neq PAT \wedge$
$getType(current) \neq PMT \wedge$
$Unreferenced\_PID \neq IND$
THEN
$current :\in flow$ ||
$timeTS := timeTS + Delay$ ||
$Unreferenced\_PID := IND$
END

trap $\widehat{=}$
BEGIN
$current :\in flow$ ||
$timeTS := timeTS + Delay$ ||
$Unreferenced\_PID := IND$
END

Ces deux derniers événements permettent le traitement des paquets qui ne sont ni des PATs ni des PMTs et dont le PID a été vérifié (nosystem). L'événement trap représente la perte du paquet pour des raisons encore ignorées.

4.1.3 Second raffinement : SYS2

Le second raffinement permet de détailler certaines parties du système et d'intégrer l'évaluation de nouveaux paramètres. Les variables représentant ces évaluations sont ajoutées au modèle et prennent part à la hiérarchie en construction.

Constantes et propriétés

Le modèle **SYS2** précise peu le comportement de l'environnement. Seule la constante *Synchro* est ajoutée aux constantes. La fonction *Synchro* modélise la lecture de l'octet de synchronisation de l'en-tête des paquets par le système.

$$Synchro \in TS\_PACKET \rightarrow VALUE \wedge$$
$$\forall(p).(p \in TS\_PACKET \wedge p \notin PaT \Rightarrow getType(p) \neq PAT)$$

Variables et invariant

Les nouvelles variables introduites sont listées dans le tableau suivant. Deux nouveaux paramètres sont introduits : ils sont chargés d'évaluer l'état de synchronisation du réseau. Les deux variables *count* et *count2* sont de simples compteurs utiles à l'acquisition de la synchronisation (voir chapitre 2.3.3, page 43).

Variable	Définition
TS_sync_loss	Paramètre d'acquisition de la synchronisation
Sync_byte_error	Vérification de l'octet de synchro du paquet courant
count	Nombre de paquets successifs incorrects (synchro)
count2	Nombre de paquets successifs corrects (non synchro)

L'invariant de typage suivant identifie les nouveaux paramètres et permet de préciser les relations entre types de paquets et paramètres. La plupart des propriétés vise à assurer la cohérence des traitements réalisés avec la nature du paquet analysé. En effet, en fonction du type du paquet courant, l'outil de mesure n'effectue pas les mêmes vérifications.

$$
\begin{array}{l}
TS\_sync\_loss \in VALUE \wedge \\
Sync\_byte\_error \in VALUE \wedge \\
count \in \mathbb{N} \wedge \\
count2 \in \mathbb{N} \wedge \\
(getType(current) \neq PAT \ \wedge \ getPID(current) \neq PAT\_PID \Rightarrow PAT\_error\_struct = IND) \wedge \\
(PAT\_error\_struct = OK \Rightarrow current \in PaT) \wedge \\
(PAT\_error\_struct = KO \Rightarrow getPID(current) = PAT\_PID \ \wedge \ getType(current) \neq PAT) \wedge \\
(PMT\_error\_struct = OK \Rightarrow getPID(current) \in \mathbf{dom}(table) \ \wedge \ getType(current) = PMT) \wedge \\
(PMT\_error\_struct = KO \Rightarrow getType(current) \neq PMT)
\end{array}
$$

L'invariant de dépendance suivant permet de montrer l'importance des deux nouveaux paramètres ajoutés (*Sync_byte_error* et *TS_sync_loss*) représentant la gestion de la synchronisation. En effet, sans l'acquisition de la synchronisation, le système est incapable de faire un quelconque traitement. Les nouveaux paramètres de ce raffinement se retrouvent donc au sommet de la hiérarchie en construction.

$$
\begin{array}{l}
(Sync\_byte\_error \neq OK \Rightarrow Unreferenced\_PID = IND) \wedge \\
(Sync\_byte\_error \neq OK \Rightarrow PAT\_error\_crypt = IND) \wedge \\
(Sync\_byte\_error \neq OK \Rightarrow PMT\_error\_crypt = IND) \wedge \\
(Sync\_byte\_error \neq OK \Rightarrow PMT\_error\_struct = IND) \wedge \\
(Sync\_byte\_error \neq OK \Rightarrow PAT\_error\_struct = IND) \wedge \\
(TS\_sync\_loss \neq OK \Rightarrow PAT\_error = IND) \wedge \\
(TS\_sync\_loss \neq OK \Rightarrow Unreferenced\_PID = IND) \wedge \\
(TS\_sync\_loss \neq OK \Rightarrow Sync\_byte\_error = IND) \wedge \\
(TS\_sync\_loss \neq OK \Rightarrow PAT\_error\_struct = IND) \wedge \\
(TS\_sync\_loss \neq OK \Rightarrow PMT\_error\_struct = IND) \wedge \\
(PAT\_error \neq OK \Rightarrow Unreferenced\_PID = IND) \wedge \\
(PMT\_error \neq OK \Rightarrow Unreferenced\_PID = IND) \wedge \\
(PAT\_error\_struct \neq IND \Rightarrow PMT\_error\_struct = IND) \wedge \\
(PMT\_error\_struct \neq IND \Rightarrow PAT\_error\_struct = IND)
\end{array}
$$

On en déduit finalement les relations suivantes : $Unreferenced\_PID \prec Sync\_byte\_error$, $PAT\_error\_struct \prec Sync\_byte\_error$, $PMT\_error\_struct \prec Sync\_byte\_error$, etc..

Ce nouvel invariant de dépendance introduit une nouvelle relation entre les paramètres. En effet, les deux dernières propriétés énoncées ne sont pas de la forme précédemment définie. Ces propriétés expriment en réalité une notion d'exclusion mutuelle plutôt qu'un rapport de dépendance. Elles montrent que les paramètres impliqués dans une telle relation ne peuvent être évalués en même temps. De plus,

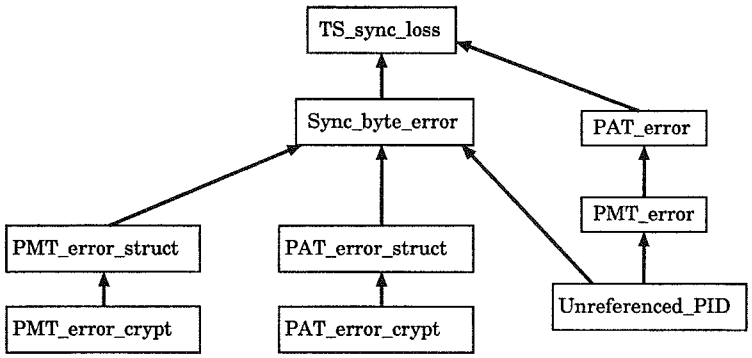


FIG. 4.4 – Hiérarchie **SYS2** : un graphe acyclique

tant qu'un des deux paramètres a un sens (càd une valeur fixée différente de IND) l'autre n'en a pas et inversement.

Définition 2 *On dira que* $Param_A$ *et* $Param_B$ *sont en exclusion mutuelle (*$Param_A \sharp Param_B$*) quand les deux variables* $Param_A$ *et* $Param_B$ *sont liées par l'une des deux propriétés équivalentes suivantes :*
$Param_A \neq IND \Rightarrow Param_B = IND$ *ou*
$Param_B \neq IND \Rightarrow Param_A = IND$

L'existence d'une telle relation entre deux paramètres est une information importante : elle permet la déduction d'un partitionnement des traitements en permettant de définir des familles (classes) de paramètres n'étant jamais exécutées au même instant et pouvant être "mobiles". Nous débattrons de ce type d'informations plus en détail dans la section 4.2.

Hiérarchie incrémentale

Comme annoncé précédemment, l'importance de la synchronisation dans le système implique que l'ajout des paramètres de synchronisation dans la hiérarchie se fait au sommet. La figure 4.4 présente la nouvelle apparence de la hiérarchie. La forêt de dépendance se transforme donc en un arbre ayant une racine et un certain nombre de branches contenant des paramètres en rapport avec les différentes natures des paquets traités. Certaines branches de cet arbre ne sont jamais évaluées en même temps (*PAT_error_struct* et *PMT_error_struct* par exemple). Par ailleurs, certaines branches sont toujours évaluées quel que soit le type de paquets et ne sont pas concernées par la nature du paquet courant (*PAT_error*, *PMT_error* par exemple). Ces paramètres sont en réalité des paramètres concernant le flux lui-même, plus que les paquets. Ces paramètres sont particuliers et c'est l'ajout du paramètre *TS_sync_loss* qui permet de réunir les paramètres concernant les paquets et les paramètres concernant le flux. De fait, ces paramètres concernant le flux ne sont pas réinitialisés à chaque nouveau paquet. Ce sont les propriétés d'exclusion mutuelle contenues dans l'invariant du modèle qui nous renseignent sur les branches concernées et celles non concernées.

Le modèle abstrait, choisi initialement qui est basé sur l'existence de différentes natures de paquets, a conduit à l'introduction initiale de paramètres vérifiant la bonne formation des paquets. Ainsi, nos deux premiers modèles **SYS0** et **SYS1** ont surtout mis l'accent sur le respect des règles de bonne construction des paquets et la première hiérarchie est une forêt reflétant les différentes natures des paquets. Avec ce nouveau raffinement et l'introduction des paramètres de synchronisation, cette forêt devient un graphe ayant un unique sommet indiquant que quelle que soit la nature des paquets la synchronisation doit être obtenue. L'ajout des deux paramètres de synchronisation (*TS_sync_loss* et *Sync_byte_error*) permet de faire la jonction entre les paramètres concernant les paquets et ceux concernant le flux. En effet, le paramètre *Sync_byte_error* est à évaluer pour chaque nouveau paquet et permet l'évaluation (en partie) du paramètre *TS_sync_loss* qui est une évaluation du flux. La synchronisation est donc au coeur du système, le raffinement nous a permis, malgré une abstraction qui n'en tenait pas vraiment compte, de retrouver sa véritable importance (comme décrit dans la norme).

Evénements

Les nouveaux événements suivants représentent l'évaluation des paramètres *TS_Sync_Loss* et *Sync_byte_error*. L'évaluation de ces paramètres nous ayant déjà servi d'exemple lors de la présentation du B événementiel, nous ne détaillerons pas ici ces événements qui restent assez abstraits. En effet, l'intérêt de ces modèles n'est pas la production des différents algorithmes du système mais, l'étude des relations entre les différents paramètres (ou traitements) dans le but de produire une architecture adaptée au problème.

Le couple d'événements testsyncOK et testsyncKO représente l'évaluation du paramètre *Sync_byte_error*. D'après l'invariant du modèle, ce paramètre n'a de sens que lorsque le paramètre *TS_Sync_loss* est correct (càd que la synchronisation est acquise). Les traitements sont, ici, extrêmement simples et se résument à une simple lecture du premier octet de l'en-tête du paquet (*Synchro(current)*) et à l'incrément du compteur *count*.

testsyncOK $\widehat{=}$
SELECT
$TS\_sync\_loss = OK \ \wedge$
$Sync\_byte\_error = IND \ \wedge$
$Synchro(current) = OK$
THEN
$Sync\_byte\_error := OK \ \|$
$b := TRUE \ \|$
$count := 0$
END

testsyncKO $\widehat{=}$
SELECT
$TS\_sync\_loss = OK \ \wedge$
$Sync\_byte\_error = IND \ \wedge$
$Synchro(current) = KO$
THEN
$Sync\_byte\_error := KO \ \|$
$b := TRUE \ \|$
$count := count + 1$
END

Lorsque le système n'est pas synchronisé ($TS\_Sync\_loss \neq OK$), le paramètre *Sync_byte_error* n'a pas de sens. Cependant, le système doit retrouver l'octet de synchronisation pour détecter le début des paquets. Le couple d'événements suivants simule cette recherche.

testNOsyncOK $\widehat{=}$
SELECT
$TS\_sync\_loss \neq OK \ \wedge$
$b = FALSE \ \wedge$
$Synchro(current) = OK$
THEN
$b := TRUE \ \|$
$count2 := count2 + 1$
END

testNOsyncKO $\widehat{=}$
SELECT
$TS\_sync\_loss \neq OK \ \wedge$
$b = FALSE \ \wedge$
$Synchro(current) = KO$
THEN
$b := TRUE \ \|$
$count2 := 0$
END

Les quatre événements précédents représentent les différents cas de vérification de l'octet de synchronisation de chaque paquet. Il permet de gérer un compteur (*count*) comptant le nombre de paquets successifs ayant leur octet de synchronisation défaillant dans le cas où le flux est déjà synchronisé. Symétriquement, si le flux n'est pas correctement synchronisé un compteur *count2* compte le nombre de paquets ayant un octet de synchronisation correct.

retrsync $\widehat{=}$
SELECT
$Synchro(current) = OK \ \wedge$
$b = TRUE \ \wedge$
$count2 \geq 5$
THEN
$TS\_sync\_loss := OK \ ||$
$Sync\_byte\_error := OK \ ||$
$count := 0$
END

errorsync $\widehat{=}$
SELECT
$Sync\_byte\_error = KO \ \wedge$
$count \geq 2$
THEN
$TS\_sync\_loss := KO \ ||$
$Sync\_byte\_error := IND \ ||$
$PAT\_error\_struct := IND \ ||$
$PMT\_error\_struct := IND \ ||$
$count := 0$
END

Ces deux événements permettent la modélisation de l'obtention et de la perte de synchronisation. Quand un nombre suffisant de paquets corrects est reçu, on considère le système comme synchronisé. Inversement, on considère que le système a perdu la synchronisation lorsqu'il reçoit deux paquets successifs ayant un octet de synchronisation erroné. La partie action de l'événement errorsync réinitialise de manière systématique toutes les variables représentant un paramètre. En effet, la préservation de l'invariant et des relations de dépendance impose que les paramètres dépendants de celui-ci soient invalidés signifiant l'absence de sens de ceux-ci. En effet, le paramètre TS_sync_loss est un paramètre évaluant le flux dans sa globalité et sa valeur ne se limite pas à un seul paquet.

Les nouveaux événements du système étant présentés nous allons maintenant passer en revue les versions raffinées des événements abstraits.

uploadedPAT $\widehat{=}$
ANY
t
WHERE
$getType(current) = PAT \ \wedge$
$t \in PID \nrightarrow (PID \nrightarrow TS\_PACKET) \ \wedge$
$PAT\_PID \notin \mathbf{dom}(t) \ \wedge$
$Sync\_byte\_error = OK \ \wedge$
$PAT\_error\_struct = OK \ \wedge$
$PAT\_error\_crypt = OK$
THEN
$table := t \ ||$
$current :\in flow \ ||$
$timePAT := clock + DelayPAT \ ||$
$timeTS := timeTS + Delay \ ||$
$PAT\_error := OK \ ||$
$PAT\_error\_struct := IND \ ||$
$PAT\_error\_crypt := IND \ ||$
$Sync\_byte\_error := IND$
END

testPAT $\widehat{=}$
SELECT
$clock > timePAT \ \wedge$
$PAT\_error \neq KO \ \wedge$
$TS\_sync\_loss = OK \ \wedge$
$getType(current) \neq PAT \ \wedge$
$Unreferenced\_PID = IND$
THEN
$PAT\_error := KO \ ||$
$PMT\_error := IND$
END

Les deux événements précédents sont les raffinements des événements abstraits. Ces nouvelles versions prennent en compte les dépendances ajoutées. Ainsi, les gardes sont renforcées afin de considérer l'absence d'erreur de synchronisation et la cohérence de l'en-tête du paquet (cf. uploadedPAT).

uploadedPMT $\widehat{=}$
ANY
t
WHERE
$t \in PID \nrightarrow TS\_PACKET \wedge$
$getType(current) = PMT \wedge$
$getPID(current) \in \mathbf{dom}(table) \wedge$
$PAT\_error = OK \wedge$
$PMT\_error\_struct = OK \wedge$
$PMT\_error\_crypt = OK \wedge$
$Sync\_byte\_error = OK$
THEN
$\mathbf{table}(getPID(current)) := t \parallel$
$current :\in flow \parallel$
$timePMT := clock + DelayPMT \parallel$
$timeTS := timeTS + Delay \parallel$
$PMT\_error := OK \parallel$
$PMT\_error\_struct := IND \parallel$
$PMT\_error\_crypt := IND \parallel$
$Sync\_byte\_error := IND$
END

testPMT $\widehat{=}$
SELECT
$clock > timePMT \wedge$
$PMT\_error \neq KO \wedge$
$TS\_sync\_loss = OK \wedge$
$getType(current) \neq PMT \wedge$
$Unreferenced\_PID = IND$
THEN
$PMT\_error := KO$
END

De la même manière que précédemment, les versions raffinées de ces événements prennent en compte la précédence des tests de synchronisation sans changer les tests réalisés (cf partie action des événements).

nosystem $\widehat{=}$
SELECT
$getType(current) \neq PAT \wedge$
$getType(current) \neq PMT \wedge$
$getPID(current) \neq PAT\_PID \wedge$
$getPID(current) \notin \mathbf{dom}(table) \wedge$
$Unreferenced\_PID \neq IND \wedge$
$Sync\_byte\_error = OK \wedge$
$TS\_sync\_loss = OK$
THEN
$current :\in flow \parallel$
$timeTS := timeTS + Delay \parallel$
$Sync\_byte\_error := IND \parallel$
$PMT\_error\_struct := IND \parallel$
$PAT\_error\_struct := IND \parallel$
$Unreferenced\_PID := IND$
END

trap $\widehat{=}$
SELECT
$Sync\_byte\_error = KO \vee$
$TS\_sync\_loss = KO$
THEN
$current :\in flow \parallel$
$timeTS := timeTS + Delay \parallel$
$Sync\_byte\_error := IND$
END

L'événement nosystem modélise le traitement des paquets de données une fois la synchronisation vérifiée. Les paramètres ayant une sémantique "limitée au paquet"[10] sont réinitialisés afin de permettre l'évaluation du paquet suivant une fois le traitement en cours terminé. Par ailleurs, l'événement trap a maintenant une justification réelle : il modélise le fait que les paquets non correctement construits (mauvais octet de synchronisation) sont jetés sans que ne soit réalisé aucun autre traitement.

[10]La valeur de ce type de paramètre est valable durant l'analyse d'un paquet. Elle ne se prolonge pas dans le temps.

testPATKO $\widehat{=}$
SELECT
$getType(current) \neq PAT \wedge$
$getPID(current) = PAT\_PID \wedge$
$Sync\_byte\_error = OK$
THEN
$PAT\_error\_struct := KO$
END

testPATOK $\widehat{=}$
SELECT
$getType(current) = PAT \wedge$
$getPID(current) = PAT\_PID \wedge$
$Sync\_byte\_error = OK$
THEN
$PAT\_error\_struct := OK$
END

Ici encore, les nouvelles versions des événements prennent en compte l'importance de la synchronisation. En effet, un paquet quelconque n'a pas à être pris en compte lorsque son octet de synchronisation n'est pas correct. La prise en compte de tels paquets dans les évaluations suivantes risque de provoquer des "cascades d'erreurs" non significatives de la réelle défaillance.

testPIDKO $\widehat{=}$
SELECT
$getType(current) \neq PAT \wedge$
$Sync\_byte\_error = OK \wedge$
$Unreferenced\_PID = IND \wedge$
$TS\_sync\_loss = OK \wedge$
$PAT\_error = OK \wedge$
$PMT\_error = OK \wedge$
$getType(current) \neq PMT \wedge$
$\forall x.(x \in \mathbf{dom}(table) \Rightarrow$
$getPID(current) \notin \mathbf{dom}(table(x)))$
THEN
$Unreferenced\_PID := KO$
END

testPIDOK $\widehat{=}$
ANY
x
WHERE
$Sync\_byte\_error = OK \wedge$
$Unreferenced\_PID = IND \wedge$
$PAT\_error = OK \wedge$
$PMT\_error = OK \wedge$
$TS\_sync\_loss = OK \wedge$
$x \in \mathbf{dom}(table) \wedge$
$getType(current) \neq PAT \wedge$
$getType(current) \neq PMT \wedge$
$getPID(current) \in \mathbf{dom}(table(x))$
THEN
$Unreferenced\_PID := OK$
END

Les événements testPIDKO et testPIDOK ont toujours le même rôle mais leurs gardes sont plus restrictives afin de vérifier la précédence établie par l'invariant. Les redondances apparaissant dans les gardes des événements sont voulues afin d'établir clairement, dans un premier temps, l'ordonnancement des divers événements. Ces redondances seront supprimées dans les futurs raffinements pour obtenir les gardes *optimales* respectant la hiérarchie de paramètres imposée par l'invariant.

4.1.4 Autres raffinements

Nous présentons ici les hiérarchies issues des différents raffinements successifs. Ces raffinements ne présentent aucune difficulté et pas d'intérêt particulier. Nous nous contentons de décrire le système avec de plus en plus de détails et nous introduisons "naturellement" les différents paramètres quand la description du système le permet.

Modèle SYS3

Le modèle **SYS3** ajoute des paramètres liés à la continuité des paquets ainsi que la vérification de l'utilisation effective des différents PID référencés dans les tables (PID_error). La vérification de la continuité des paquets permet d'avoir une information sur une éventuelle perte de paquet. Comme annoncée précédemment, la valeur de ce paramètre est donc extrêmement importante pour l'analyse du décodeur T-STD. La figure 4.5 résume les paramètres présents dans ce modèle et leurs relations.

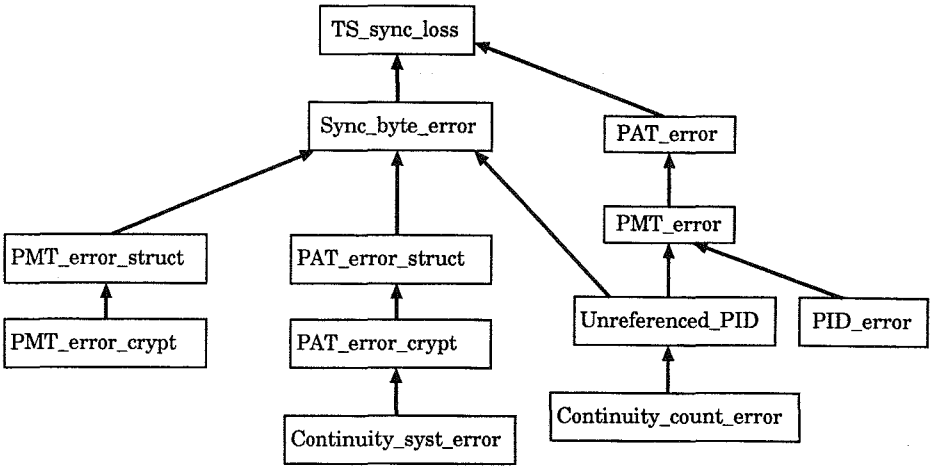


FIG. 4.5 – Hiérarchie issue du modèle **SYS3**

Modèle SYS4

Le modèle **SYS4** ajoute les différentes évaluations liées aux horloges des programmes (voir paragraphe 3.1.4, page 59). Ces paramètres sont relativement complexes :

- *PCR_repetition_error* est chargé de vérifier la régularité de réception de paquets contenant la valeur des horloges.
- Les deux paramètres *PCR_accuracy* et *PCR_discontinuity* sont chargés de vérifier que les valeurs d'horloges reçues ne sont pas trop éloignées des valeurs des horloges locales.

Un autre paramètre important est également pris en compte dans ce modèle. Le paramètre *Transport_error* qui consiste en la lecture d'un octet de validité dans l'en-tête du paquet traité. L'octet de validité indique la correction du paquet en particulier vis-à-vis des codes correcteurs d'erreurs. La position dans la hiérarchie de ce paramètre est assez élevée (voir figure 4.6) malgré sa présence en seconde catégorie dans la norme. Le texte suivant, extrait de la norme, illustre la description du paramètre et le travail de réflexion mené afin de retrouver les dépendances existantes :

Transport_error :
The primary Transport_error indicator is Boolean, but there should be also a resettable binary counter which counts the erroneous TS packets... If an error occurs, no further error indication should be derived from the erroneous packet.

La dernière phrase de description du paramètre sous-entend clairement l'importance de celui-ci. Notre raisonnement, partant de cette description a été de trouver les conditions minimales requises pour pouvoir évaluer le paramètre *Transport_error* : il suffit que le récepteur soit synchronisé, c'est à dire que les paramètres *TS_sync_loss* et *Sync_byte_error* soient corrects. Il existe également une certaine indépendance du paramètre *Transport_error* (qui porte sur la correction d'un paquet) et les paramètres *PAT_error*, *PMT_error*, *PTS_error* ou encore *PCR_repetition_error* qui concernent le flux. Cependant, l'ensemble restant des paramètres ne peut pas, en accord avec la définition proposée par la norme, être évalué en cas d'erreur des codes correcteurs (càd du paramètre *Transport_error*). La figure 4.6 présente l'ajout de ces nouveaux paramètres dans la hiérarchie.

Modèle SYS5

Le modèle **SYS5** rajoute les derniers paramètres non encore traités. Ces derniers paramètres concernent essentiellement des tables secondaires (NIT, EIT, ...). Les traitements et l'ordonnancement de ces paramètres sont similaires. La figure 4.7 présente l'ensemble des paramètres pris en compte dans le modèle.

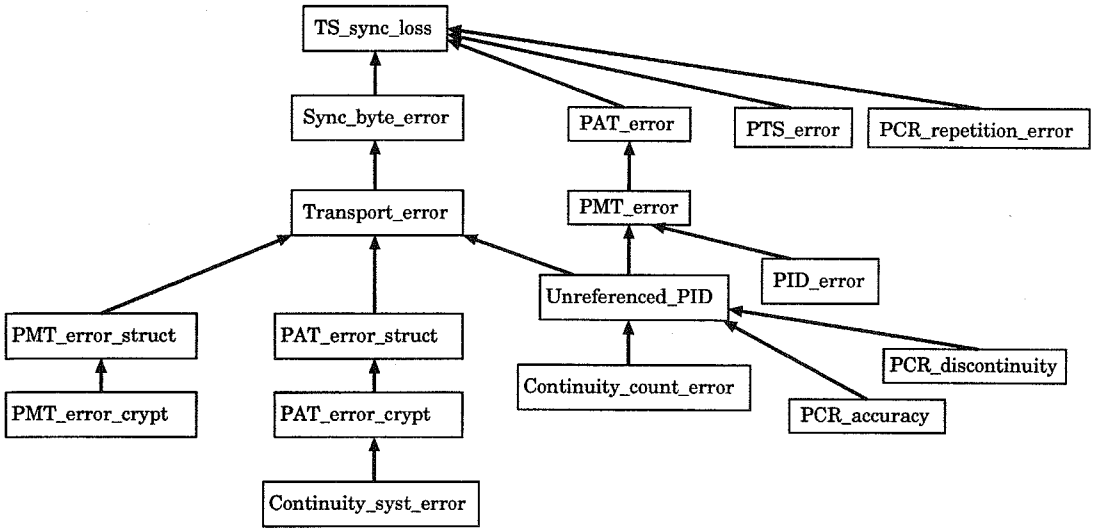


FIG. 4.6 – Hiérarchie issue du modèle **SYS4**

Seuls les paramètres dits de synthèse ou *paramètres de Qualité de Service* (QdS) ne sont pas pris encore en compte dans ce modèle.

4.1.5 Raffinement SYS6 : paramètres QdS

Le dernier raffinement introduit les paramètres QdS, composition d'autres paramètres. Le paragraphe 3.1.6 a présenté dans le détail ces récents paramètres et leurs motivations. Nous avons décidé de considérer les trois paramètres de synthèse dans nos modèles sous la forme de trois variables : SI_error, SD_error et SA_error.

L'invariant de typage suivant indique que les nouvelles variables ajoutées sont des paramètres.

$$\begin{array}{l} CRC\_error \in VALUE \wedge \\ SA\_error \in VALUE \wedge \\ SD\_error \in VALUE \wedge \\ SI\_error \in VALUE \end{array}$$

L'invariant de dépendance exprime le fait que les paramètres de QdS sont des *compositions* d'anciens paramètres *élémentaires*. De ce fait, il existe des dépendances entre la valeur d'un paramètre QdS et les paramètres le constituant. Cependant, les propriétés exprimées ci-après sont beaucoup plus restrictives et précises que les propriétés de dépendances définies dans les modèles précédents.

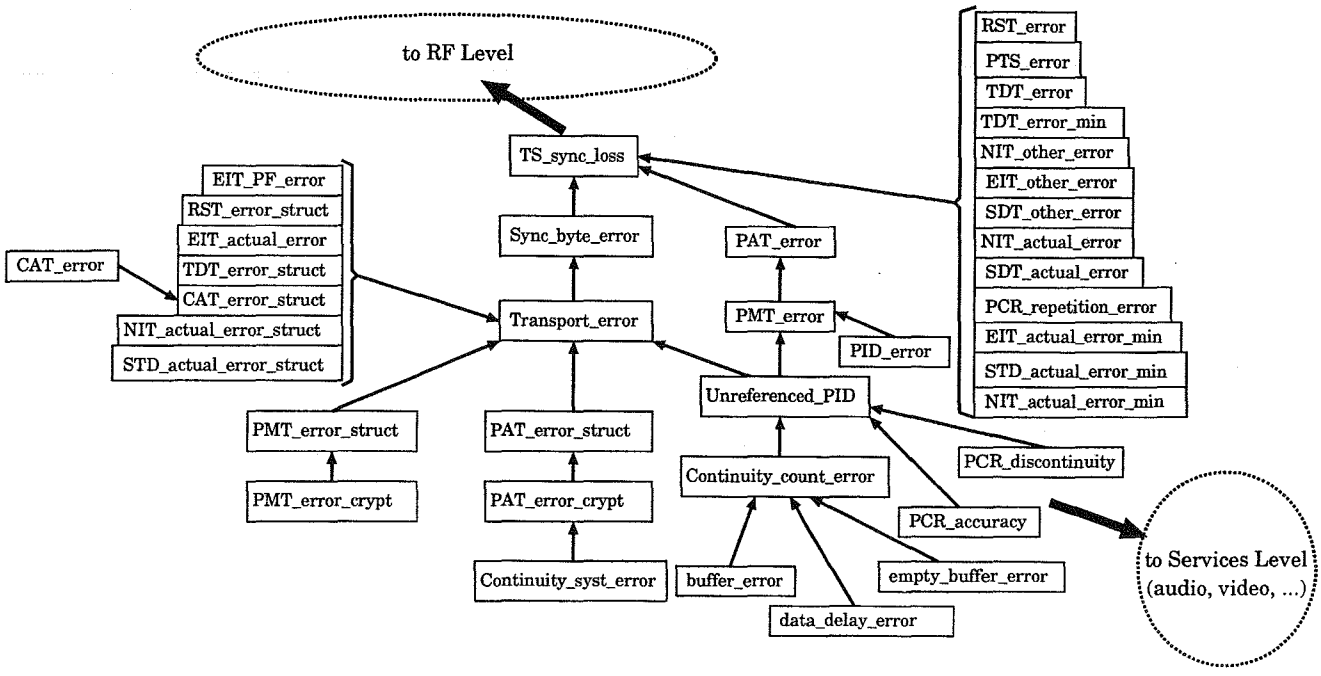


FIG. 4.7 – Hiérarchie issue du modèle **SYS5**

$$
\begin{aligned}
&(TS\_sync\_loss = KO \Rightarrow SA\_error = KO) \wedge \\
&(PAT\_error = KO \Rightarrow SA\_error = KO) \wedge \\
&(PMT\_error = KO \Rightarrow SA\_error = KO) \wedge \\
&(SDT\_actual\_error = KO \Rightarrow SD\_error = KO) \wedge \\
&(CRC\_error = KO \Rightarrow SD\_error = KO) \wedge \\
&(Continuity\_count\_error = KO \Rightarrow SI\_error = KO) \wedge \\
&(Continuity\_syst\_error = KO \Rightarrow SI\_error = KO) \wedge \\
&(Transport\_error = KO \Rightarrow SI\_error = KO) \wedge \\
&(PAT\_error\_struct = KO \Rightarrow CRC\_error = KO) \wedge \\
&(PMT\_error\_struct = KO \Rightarrow CRC\_error = KO) \wedge \\
&(CAT\_error = KO \Rightarrow CRC\_error = KO) \wedge \\
&(NIT\_actual\_error\_struct = KO \Rightarrow CRC\_error = KO) \wedge \\
&(SDT\_actual\_error\_struct = KO \Rightarrow CRC\_error = KO)
\end{aligned}
$$

On peut à l'aide de ces propositions déduire la sémantique des paramètres ainsi proposés. La hiérarchie présentée dans la partie suivante permet de visualiser les buts des différents paramètres de QdS proposés.

Hiérarchie incrémentale

La nouvelle hiérarchie obtenue est une précision de la précédente. On analyse la sémantique des paramètres de QdS de la manière suivante :

Le paramètre *SI_error* est chargé de détecter de légères perturbations du signal. Ces perturbations ne compromettent pas la réception (les paramètres élémentaires composant *SI_error* sont au bas de la hiérarchie) mais sont locales à un programme ou à une/des image(s). Par exemple, des inversions d'ordre dans les paquets peuvent provoquer une réception trop tardive des données, d'où une image incomplète.

Le paramètre *SA_error* est chargé de détecter une perte complète du signal. En effet, les paramètres le constituant sont placés au sommet de la hiérarchie et ont un impact très fort sur le système (acquisition

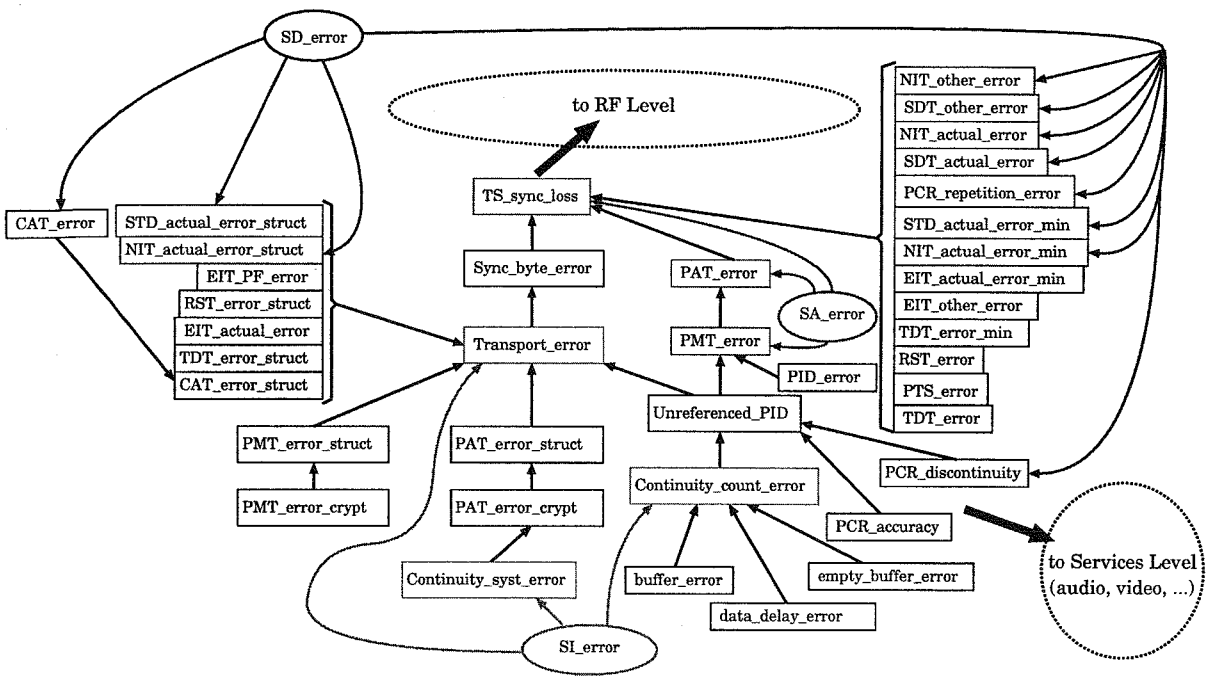


FIG. 4.8 – Hiérarchie finale

de la synchronisation et extraction des tables systèmes). Le paramètre *SA_error* étant une indication "a posteriori" d'un problème grave et le paramètre *SI_error* recherchant des erreurs locales minimes, le paramètre *SD_error* est un paramètre intermédiaire vérifiant un grand nombre de paramètres élémentaires afin de détecter une erreur avant qu'il ne soit trop tard (càd avant le déclenchement du paramètre *SA_error* et donc la perte du signal).

De ce fait *SD_error* vérifie la correction d'un certain nombre de paramètres "secondaires" et en particulier la bonne réception de paquets contenant des tables non indispensables. Ce parcours en largeur de la hiérarchie augmente en effet les chances de découvrir un problème s'il y en a effectivement un. La figure 4.8 résume l'ajout des paramètres de synthèse à la hiérarchie.

Il est important de rappeler que ces trois paramètres dit de *Qualité de Service* ou de *synthèse* sont issus d'une étude psychovisuelle poussée et sont empiriques. La déduction de ces paramètres n'a été faite que sur des raisonnements probabilistiques. Notre modélisation permet cependant de justifier l'existence de ces paramètres et le choix des paramètres de base les composant. En effet, les positions occupées dans la hiérarchie par les différents paramètres (de base) constituant les paramètres de QdS sont révélatrices de leur importance. Les paramètres constituant *SA_error* sont situés au sommet de la hiérarchie, et inspecter ces paramètres pour diagnostiquer la présence du signal semble naturel : il est certain qu'en cas d'erreur sur un de ces paramètres, le signal est absent. De même pour le paramètre *SI_error*, la position des paramètres *Continuity_count_error*, *Continuity_syst_error* et *Transport_error* ainsi que leurs définitions permet de conclure qu'une défaillance de ces paramètres a un impact limité au paquet courant. De ce fait, des erreurs à répétition concernant ces paramètres peuvent provoquer de légères perturbations d'images (parties d'images manquantes suite à une perte ou invalidité de paquets). Nos modélisations et les études empiriques réalisées par des spécialistes du domaine de la DVB permettent d'arriver aux mêmes conclusions, ce qui est, pour nous, une preuve supplémentaire de la puissance du raffinement.

Evénements

Le nouveau raffinement ne nécessite pas de nouveaux événements. Les paramètres QdS ajoutés sont des compositions d'anciens paramètres, certains événements ont donc été raffinés afin de prendre en

compte la mise à jour de ces nouveaux paramètres.

testTransportKO $\widehat{=}$
SELECT
$Sync\_byte\_error = OK \wedge$
$Transport\_error\_indicator(current) = KO$
THEN
$Transport\_error := KO \parallel$
$SI\_error := KO \parallel$
$Transport\_error\_count := Transport\_error\_count + 1$
END

La version raffinée de l'événement testTransportKO prend en compte la mise à jour du paramètre *SI_error* respectant ainsi l'invariant défini.

testContOK $\widehat{=}$
SELECT
$Sync\_byte\_error = OK \wedge$
$PAT\_error\_struct = OK \wedge$
$PAT\_error\_crypt = OK \wedge$
$Transport\_error = OK \wedge$
$getNumber(current) \leq SUCC(memory(PAT\_PID)) \wedge$
$getNumber(current) \geq memory(PAT\_PID)$
THEN
$\mathbf{memory}(PAT\_PID) := getNumber(current) \parallel$
$Continuity\_syst\_error := OK \parallel$
$SI\_error := OK$
END

testContKO $\widehat{=}$
SELECT
$Sync\_byte\_error = OK \wedge$
$PAT\_error\_struct = OK \wedge$
$PAT\_error\_crypt = OK \wedge$
$Transport\_error = OK \wedge$
$(getNumber(current) > SUCC(memory(PAT\_PID)) \vee$
$getNumber(current) < memory(PAT\_PID))$
THEN
$Continuity\_syst\_error := KO \parallel$
$SI\_error := KO$
END

Les deux événements précédents prennent en compte l'existence du paramètre *SI_error* et le mettent ainsi à jour. L'événement testContKO représente le cas d'erreur tandis que l'événement testContOK représente une évaluation correcte des paramètres *Continuity_syst_error* et *SI_error*. En effet, la lecture des propriétés de l'invariant de dépendance concernant *SI_error* nous montre que le paramètre *Continuity_syst_error* est le paramètre "le plus bas" de la hiérarchie intervenant dans le calcul de *SI_error*. Une fois l'évaluation du paramètre *Continuity_syst_error* terminée nous connaissons la valeur du paramètre QdS associé.

testContinuityKO $\widehat{=}$
SELECT
$getType(current) \neq PAT \wedge$
$getType(current) \neq PMT \wedge$
$Sync\_byte\_error = OK \wedge$
$Unreferenced\_PID = OK \wedge$
$getNumber(current) \neq SUCC(memory(getPID(current))) \wedge$
$getNumber(current) \neq memory(getPID(current))$
THEN
$Continuity\_count\_error := KO \parallel$
$SI\_error := KO$
END

testContinuityOK $\widehat{=}$
SELECT
$getType(current) \neq PAT \wedge$
$getType(current) \neq PMT \wedge$
$Sync\_byte\_error = OK \wedge$
$Unreferenced\_PID = OK \wedge$
$(getNumber(current) = SUCC(memory(getPID(current))) \vee$
$getNumber(current) = memory(getPID(current)))$
THEN
$\mathbf{memory}(getPID(current)) := getNumber(current) \parallel$
$Continuity\_count\_error := OK \parallel$
$SI\_error := OK$
END

La remarque précédente s'applique aussi aux événements testContOK et testContKO.

retrsync $\widehat{=}$
SELECT
$Synchro(current) = OK \wedge$
$count \geq 5$
THEN
$TS\_sync\_loss := OK \parallel$
$Sync\_byte\_error := OK \parallel$
$count := 0$
END

Cette version raffinée de l'événement retrsync est désormais épurée des différentes ré-initialisations inutiles au vu des invariants de dépendance. On obtient ainsi un événement *atomique* très simple et conservant les propriétés de dépendances annoncées.

testPAT $\widehat{=}$
SELECT
$clock > timePAT \wedge$
$PAT\_error \neq KO \wedge$
$TS\_sync\_loss = OK \wedge$
$getType(current) \neq PAT \wedge$
$Unreferenced\_PID = IND$
THEN
$PAT\_error := KO \parallel$
$PMT\_error := IND \parallel$
$SA\_error := KO$
END

testPAT2 $\widehat{=}$
SELECT
$getType(current) \neq PAT \wedge$
$getPID(current) = PAT\_PID \wedge$
$Sync\_byte\_error = OK \wedge$
$Transport\_error = OK$
THEN
$PAT\_error\_struct := KO \parallel$
$CRC\_error := KO \parallel$
$SD\_error := KO$
END

Les versions raffinées de ces deux événements se contentent de prendre en compte la mise à jour des variables représentant les paramètres de QdS.

testPMT $\widehat{=}$
SELECT
$clock > timePMT \wedge$
$PMT\_error \neq KO \wedge$
$TS\_sync\_loss = OK \wedge$
$getType(current) \neq PMT \wedge$
$Unreferenced\_PID = IND$
THEN
$PMT\_error := KO \parallel$
$SA\_error := KO$
END

testSDT $\widehat{=}$
SELECT
$clock > timemaxSDT \wedge$
$TS\_sync\_loss = OK \wedge$
$current \notin SdT \wedge$
$SDT\_actual\_error \neq KO$
THEN
$SDT\_actual\_error := KO \parallel$
$SD\_error := KO$
END

De la même manière que dans les événements déjà présentés, les événements modélisant l'évaluation des paramètres composant les paramètres de QdS sont raffinés. Le raffinement en question est très simple et n'est constitué que de l'ajout d'une substitution ($SA\_error := KO$).

testPMT2 $\widehat{=}$
SELECT
$getType(current) \neq PMT \wedge$
$getPID(current) \in \mathbf{dom}(table) \wedge$
$Sync\_byte\_error = OK \wedge$
$Transport\_error = OK$
THEN
$PMT\_error\_struct := KO \parallel$
$CRC\_error := KO \parallel$
$SD\_error := KO$
END

La remarque fait précédemment s'applique également à cet événement raffiné, et une simple affectation du paramètre SD_error est ajoutée.

structNIT1 $\widehat{=}$
SELECT
$Sync\_byte\_error = OK \wedge$
$Transport\_error = OK \wedge$
$getPID(current) = NIT\_actual\_PID \wedge$
$current \notin NiT$
THEN
$NIT\_actual\_error\_struct := KO \parallel$
$CRC\_error := KO \parallel$
$SD\_error := KO$
END

structSDT1 $\widehat{=}$
SELECT
$Sync\_byte\_error = OK \wedge$
$Transport\_error = OK \wedge$
$getPID(current) = SDT\_actual\_PID \wedge$
$current \notin SdT$
THEN
$SDT\_actual\_error\_struct := KO \parallel$
$CRC\_error := KO \parallel$
$SD\_error := KO$
END

nosystem $\widehat{=}$
SELECT
$getType(current) \neq PAT \wedge$
$getType(current) \neq PMT \wedge$
$getPID(current) \neq PAT\_PID \wedge$
$getPID(current) \notin \mathbf{dom}(table) \wedge$
$Unreferenced\_PID \neq IND \wedge$
$Sync\_byte\_error = OK \wedge$
$Transport\_error = OK \wedge$
$TS\_sync\_loss = OK \wedge$
$Continuity\_count\_error \neq IND \wedge$
$(HavePCR(current) = KO \vee boolPCR = OK)$
THEN
$current :\in flow \parallel$
$timeTS := timeTS + Delay \parallel$
$Sync\_byte\_error := IND \parallel$
$Unreferenced\_PID := IND \parallel$
$Continuity\_count\_error := IND \parallel$
$Transport\_error := IND \parallel$
$SI\_error := IND \parallel$
$NIT\_actual\_error\_struct := IND \parallel$
$CRC\_error := IND \parallel$
$CAT\_error := IND \parallel$
$SDT\_actual\_error\_struct := IND \parallel$
$PCR\_accuracy := IND \parallel$
$PCR\_discontinuity := IND \parallel$
$boolPCR := IND \parallel$
$RST\_error\_struct := IND \parallel$
$CAT\_error\_struct := IND \parallel$
$TDT\_error\_struct := IND \parallel$
$EIT\_actual\_error\_struct := IND$
END

La nouvelle version de l'événememnt trap prend en compte le paramètre SI_error défini et réinitialise celui-ci en même temps que les autres paramètres.

```
trap ≙
  SELECT
      Sync_byte_error = KO ∨
      TS_sync_loss = KO
  THEN
      current :∈ flow ||
      timeTS := timeTS + Delay ||
      Sync_byte_error := IND ||
      Unreferenced_PID := IND ||
      Transport_error := IND ||
      SI_error := IND ||
      Continuity_count_error := IND ||
      PCR_accuracy := IND ||
      PCR_discontinuity := IND
  END
```

Da la même façon, la nouvelle version de cet événement permet la réinitialisation du paramètre *SI_error*.

errorsync $\widehat{=}$
SELECT
$Sync\_byte\_error = KO \wedge$
$count \geq 2$
THEN
$TS\_sync\_loss := KO \parallel$
$SA\_error := KO \parallel$
$Sync\_byte\_error := IND \parallel$
$PAT\_error := IND \parallel$
$PMT\_error := IND \parallel$
$PAT\_error\_struct := IND \parallel$
$PMT\_error\_struct := IND \parallel$
$TDT\_error\_min := IND \parallel$
$RST\_error := IND \parallel$
$NIT\_actual\_error := IND \parallel$
$NIT\_other\_error := IND \parallel$
$NIT\_actual\_error\_min := IND \parallel$
$NIT\_actual\_error\_struct := IND \parallel$
$SDT\_actual\_error := IND \parallel$
$SDT\_other\_error := IND \parallel$
$SDT\_actual\_error\_min := IND \parallel$
$SDT\_actual\_error\_struct := IND \parallel$
$PCR\_repetition\_error := IND \parallel$
$PTS\_error := IND \parallel$
$TDT\_error\_struct := IND \parallel$
$TDT\_error := IND \parallel$
$CAT\_error := IND \parallel$
$CRC\_error := IND \parallel$
$RST\_error\_struct := IND \parallel$
$SD\_error := IND \parallel$
$SI\_error := IND \parallel$
$EIT\_actual\_error\_min := IND \parallel$
$EIT\_actual\_error := IND \parallel$
$EIT\_other\_error := IND \parallel$
$count := 0$
END

L'événement errorsync réinitialisant toutes les variables pour cause de perte de synchronisation les paramètres *SD_error* et *SI_error* sont également réinitialisés. Le paramètre *SA_error* n'est lui pas annulé car la perte de synchronisation est un de ses composants élémentaires, il doit donc être positionné tout comme le paramètre *TS_sync_loss*.

L'ensemble des développements présentés représente 7 modèles et 567 obligations de preuves dont environ 79% ont été prouvées de manière automatique. Les obligations de preuves restantes ont été prouvées de manière interactive. La difficulté de ces obligations de preuves n'était pas très élevée et seul le grand nombre d'hypothèses présentes dans les modèles a empêché la réalisation d'une preuve automatique. Le tableau suivant présente dans le détail les obligations de preuves associées à chaque modèle et la nature des preuves réalisées (automatique ou interactive).

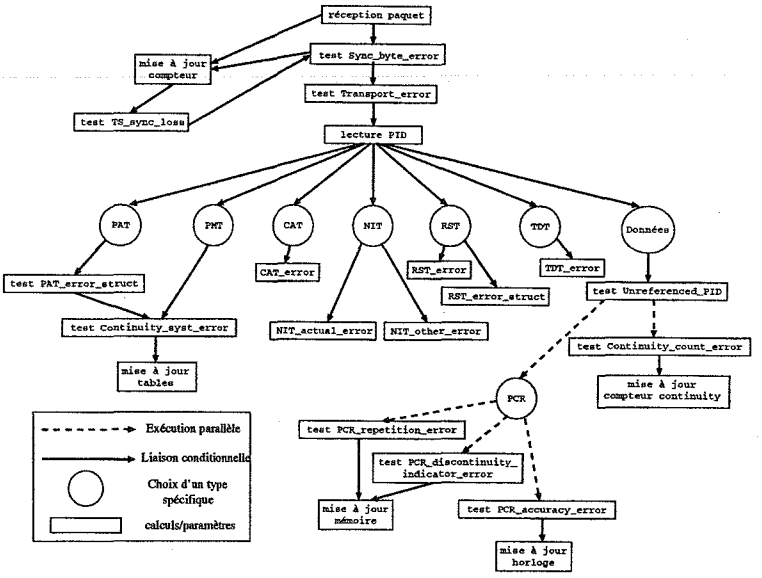


FIG. 4.9 – algorithme de calcul

Modèles B	Preuves Automatiques	Preuves Interactives	% Preuves auto/inter
SYS0	2	0	100/0
SYS1	23	0	100/0
SYS2	76	37	67/33
SYS3	39	19	67/33
SYS4	37	6	86/14
SYS5	137	49	73/27
SYS6	32	19	63/37

4.2 Eléments de conception architecturale

Les modélisations précédemment présentées nous permettent de faire un certain nombre de propositions qui doivent être ensuite présentées aux concepteurs de l'architecture.

4.2.1 Ordonnancement de tâches

La hiérarchie de la figure 4.8 nous permet directement de déduire un ordonnancement. En effet, le principe même de notre hiérarchie est qu'un paramètre donné n'a pas de signification si les paramètres dont il dépend indiquent une erreur. De ce fait, il faut attendre un résultat positif de tous ses parents avant de commencer le calcul du paramètre.

Bien que cette analyse nous permette de conclure sur l'ordre d'exécution des tâches, elle ne nous permet pas pour l'instant de conclure sur un éventuel partitionnement logiciel/matériel. Nous avons cependant une bonne connaissance des dépendances entre calculs et, dans le contexte du temps réel dynamique, cette information peut se révéler précieuse. On peut, en particulier, déduire de manière immédiate un ordre précis dans les calculs à mener. Cet ordre est donné sous forme d'un schéma arborescent figure 4.9 et on retrouve facilement les dépendances soulignées dans la figure 4.8.

Nous pouvons ensuite détailler les traitements à faire pour chaque type de paquet reçu. De fait, les traitements sont entièrement indépendants et peuvent donc être chargés sur le FPGA au besoin ou être regroupés dans des circuits génériques (cf. chapitre suivant).

Ainsi, grâce à une hiérarchisation des erreurs nous pouvons organiser les tests à effectuer sur le signal reçu afin de suivre le cours normal de construction du flux. L'ordonnancement des tests obtenus est

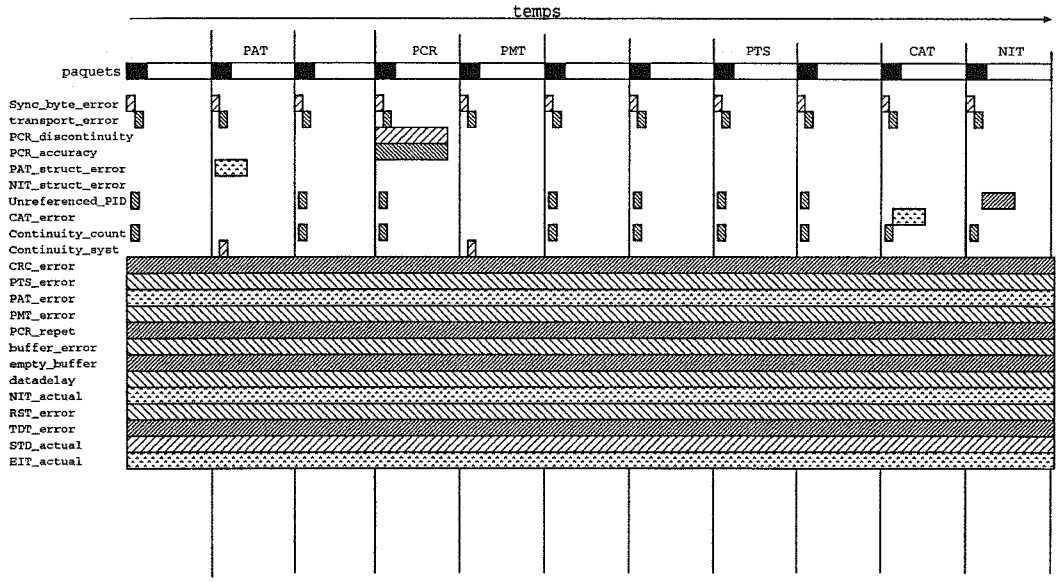


FIG. 4.10 – Durée d'activation des tâches

déduit des propriétés invariantes du système (relations sémantiques entre erreurs) mais aussi des gardes des événements.

4.2.2 Aspect temporel

La technologie utilisée étant largement reconfigurable, un point important est d'estimer la fréquence et la durée de vie des différentes tâches. Ainsi, une tâche s'exécutant périodiquement peut être "chargée" sur le matériel configurable et "supprimée" du matériel une fois son calcul effectué, et ce, jusqu'à la prochaine fois. Notre classification des indicateurs d'erreurs nous permet d'avoir une première idée sur ce sujet. En effet, un certain nombre de tests est dépendant de la valeur de l'indicateur *Sync_byte_error* qui est à tester sur chaque paquet reçu. On a donc une durée de vie pour ces tests qui ne doit pas dépasser le temps séparant deux paquets TS. D'autres paramètres ne dépendent que de *TS_sync_loss* et ont donc une durée d'activation plus longue. Par ailleurs, le code écrit dans les machines B nous permet d'avoir une idée précise du fonctionnement du calcul de certains paramètres. Ainsi, la comparaison entre une horloge et un temps limite (*PAT_error* par exemple) est un processus qui est toujours actif et qui doit toujours être présent. Nos conclusions sont résumées sur le schéma 4.10 qui représente les durées d'activation des paramètres que nous avons pris en compte dans notre modélisation.

Il apparaît très clairement qu'un certain nombre de tâches sont toujours actives mais qu'une bonne partie des calculs se fait ponctuellement lors de la réception de paquets plus ou moins particuliers. Ces calculs sont donc susceptibles d'être mobiles et de ne pas rester "en dur" sur la carte mais au contraire d'être chargés quand cela est nécessaire.

4.2.3 Génération de code

Les développements B qui ont été menés dans le cadre de ce projet, bien qu'ayant pour but principal la découverte d'une structuration intelligente, permettent de se faire une idée précise du code et/ou hardware nécessaire. Ainsi, par une traduction manuelle de certains événements on peut obtenir le code du calcul correspondant.

La définition de règles de traduction automatique de nos modèles vers des schémas électroniques de circuits permettra l'exploitation des propriétés des modèles dans l'implantation électronique de l'application modélisée.

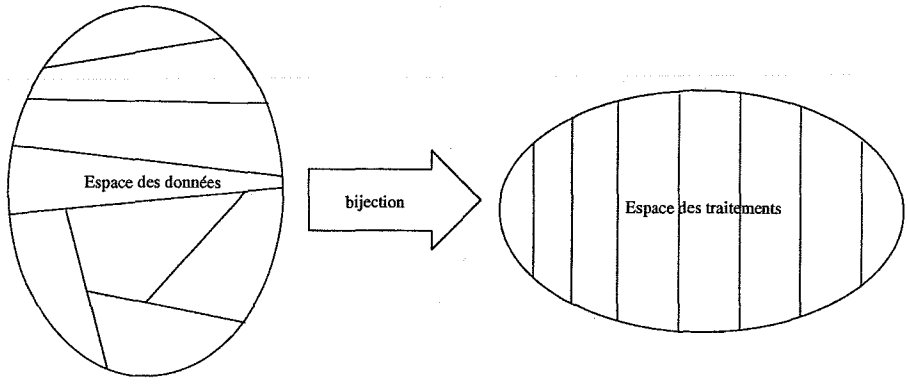


FIG. 4.11 – D'un partitionnement à l'autre

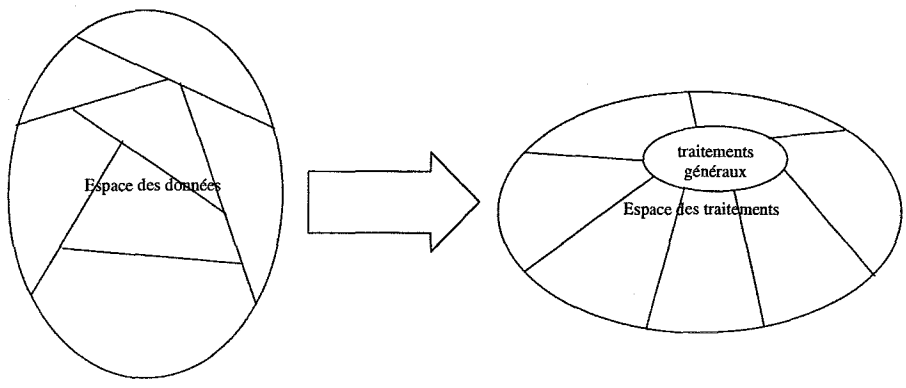


FIG. 4.12 – Partition de notre étude

Nos différents travaux nous permettent de proposer un certain nombre de solutions architecturales avant même la réalisation de l'ensemble des traitements. En effet, nos modèles nous permettent de partitionner l'espace des données en un certain nombre de sous-espaces distincts. Si nous sommes capables d'expliciter les traitements à réaliser sur ces sous-espaces, alors nous réalisons une partition de l'espace des traitements. Cette idée est résumée figure 4.11.

Cependant, dans notre étude de cas, les sous-espaces de données ont un noyau commun de traitements à réaliser (paramètres *Sync_byte_error*, *Transport_error*, etc...). Le partitionnement n'est donc pas tout à fait comme celui présenté figure 4.11 mais plutôt comme à la figure 4.12 : un sous-ensemble des traitements est commun à tous les paquets indépendamment de leur nature.

Avec de telles informations, il est facile d'imaginer un moyen de reconfiguration dynamique :
- l'ensemble des traitements communs est implanté de manière définitive "en front" du FPGA.
- la configuration des traitements spécifiques à chaque type de données est stockée dans une mémoire.
- le FPGA est reconfiguré, en temps-réel, selon le type du paquet à traiter.

Nous avons représenté ce mécanisme figure 4.13.

4.3 Bilan du projet EQUAST

Le projet EQUAST a abouti à la production de deux prototypes industriels. Ces prototypes implantent l'intégralité de la norme et permettent, comme convenu, de mesurer la qualité de transmission d'un réseau. L'importance du projet et sa complexité nous ont permis de confronter la méthode B événementielle à un problème d'envergure et pas seulement à un exemple jouet. Après avoir suivi le processus de conception

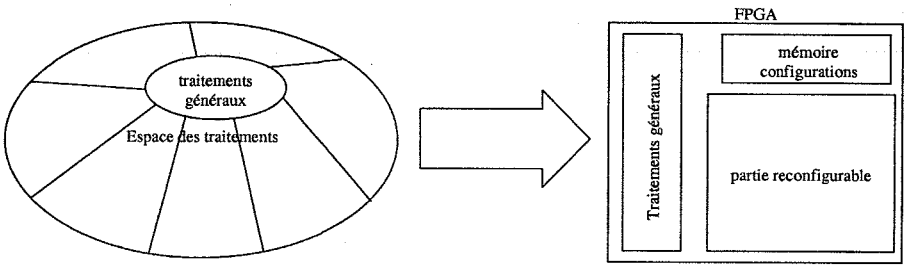


FIG. 4.13 – Vers une répartition sur FPGA

standard et pris conscience des problématiques du monde de l'électronique, nous avons proposé nos propres solutions.

En effet, notre principe de modélisation intégrale de la norme par raffinements successifs permet de suivre le flot de conception orienté modèle des SoC. L'utilisation de la méthode B événementielle a cependant l'avantage de simplifier ce schéma de conception : la preuve garantit le passage d'un modèle à l'autre et le couple abstraction/raffinement minimise l'impact des éventuels allers-retours entre les différents modèles.

La hiérarchisation des différents traitements de l'application conduit à un ordonnancement naturel et à des choix d'implantation en connaissance de cause. De plus, le partitionnement des tâches se fait de manière beaucoup plus aisée et permet une réflexion plus poussée sur l'aspect reconfiguration dynamique du système. En effet, la hiérarchie de tâches issue de nos modèles abstraits exhibe les dépendances et les parallélismes intrinsèques de l'application. De ce fait, c'est un excellent support (garanti par la preuve) pour réaliser le travail d'architecture du système.

Par notre modélisation globale du système nous avons permis l'utilisation d'un flot de conception basé sur la preuve (voir figure 4 page 10) plus rigoureux que le flot initial du projet (figure 3 page 9).

Par ailleurs, nos travaux ont permis de justifier formellement les résultats de travaux empiriques antérieurs. Les hiérarchies incrémentales issues de nos modèles permettent de reconstituer les trois niveaux de priorité de paramètres définis dans la norme et expliquent la constitution des paramètres de QdS [1]. En effet, la position, dans la hiérarchie finale, des différents paramètres constituant les paramètres de QdS permet la compréhension de ceux-ci et justifie leur existence. Les paramètres de QdS permettent un diagnostique simplifié des dysfonctionnements en réalisant la synthèse de paramètres de base. Nos modèles montrent que le choix de ces paramètres de base est judicieux et correspond effectivement à la signification empirique associée à chaque paramètre QdS.

Le choix d'une traduction de modèles B événementiels vers du code SystemC s'est imposé du fait de l'importance prise par ce langage, ces dernières années, dans le cadre de la conception de SoC. Nous avons donc décidé d'étudier plus en détail ce langage de description afin de pouvoir définir une fonction de traduction préservant les avantages d'une modélisation formelle. L'étude de SystemC sera présentée par la suite dans le chapitre 5.

Les principes mis en oeuvre dans le cadre du projet EQUAST restent assez généraux et la question de la réutilisabilité de l'approche s'est posée. Nous avonc donc décidé d'abstraire notre démarche de l'étude de cas afin de voir s'il était possible de "rejouer" nos développements sur d'autres applications. Le chapitre 6 présentera les différentes réflexions et modèles développés dans le but de s'abstraire de l'étude de cas EQUAST.

Chapitre 5

SystemC : modélisation

Parmi les langages de description utilisés pour la conception de SoC, SystemC [88] s'impose comme le plus répandu et le plus abouti. Bibliothèque C++, ce "langage" permet la description, à plusieurs niveaux (abstraction TLM jusqu'au RTL), de modules (unité matérielle) et de processus (unité fonctionnelle) communiquant. L'actuelle version de SystemC 2.0.1 propose de nombreux outils pour simuler le fonctionnement d'un SoC et en particulier un ordonnanceur (scheduler) qui gère l'exécution des processus en fonction des différents événements du système.

La sémantique de SystemC n'est absolument pas formelle et le comportement d'un programme SystemC est défini par l'ordonnanceur fourni. Le langage SystemC suscite de nombreux travaux [57, 100] visant à fixer sa sémantique de manière formelle [58, 65] éventuellement par le biais de son ordonnanceur comme Mueller [97] qui propose une modélisation formelle du scheduler de SystemC 1.0 à l'aide d'ASM[11] mais SystemC a beaucoup évolué depuis.

Nous allons donc étudier l'ordonnanceur de cette nouvelle version de SystemC dans le but d'être capable de produire un code SystemC en accord avec nos exigences. Nous allons introduire les différents principes de SystemC au fur et à mesure de nos modèles B événementiels. De cette manière, nous allons définir une sémantique opérationnelle de SystemC.

5.1 SystemC : un langage haut-niveau de description Hardware ?

L'énorme avantage de SystemC est la possibilité d'utiliser un unique langage pour l'ensemble de la conception d'un System-on-Chip. En effet, la partie TLM (*Transactionnal Level Modeling*) du langage permet de considérer l'application d'un point de vue communication abstraite alors que la partie RTL (*Register Transfert Level*) décrit un système à un niveau extrêmement bas. La coexistence, dans un même langage et outil, de niveaux d'abstraction si différents pose un certain nombre de problèmes quant à la cohérence des architectures décrites et leur viabilité.

En effet, une application décrite en SystemC TLM et simulable n'est pas toujours synthétisable et permet essentiellement une réflexion sur la communication entre les différents blocs architecturaux. Ce niveau de description permet d'étudier la communication entre les différents blocs sans fixer la nature (logicielle ou hardware) de ceux-ci. Par ailleurs, la description RTL d'une architecture est la réalisation d'une analyse ainsi que d'une étude poussée de différentes solutions architecturales et, est destinée à une implantation matérielle.

5.1.1 SystemC : une bibliothèque C++

SystemC est une bibliothèque C++ dédiée à la description de systèmes électroniques. Cette bibliothèque contient un certain nombre d'objets et de macros prédéfinis qui permettent de manipuler facilement

[11] Un travail similaire des mêmes auteurs a également été réalisé sur SpecC [87].

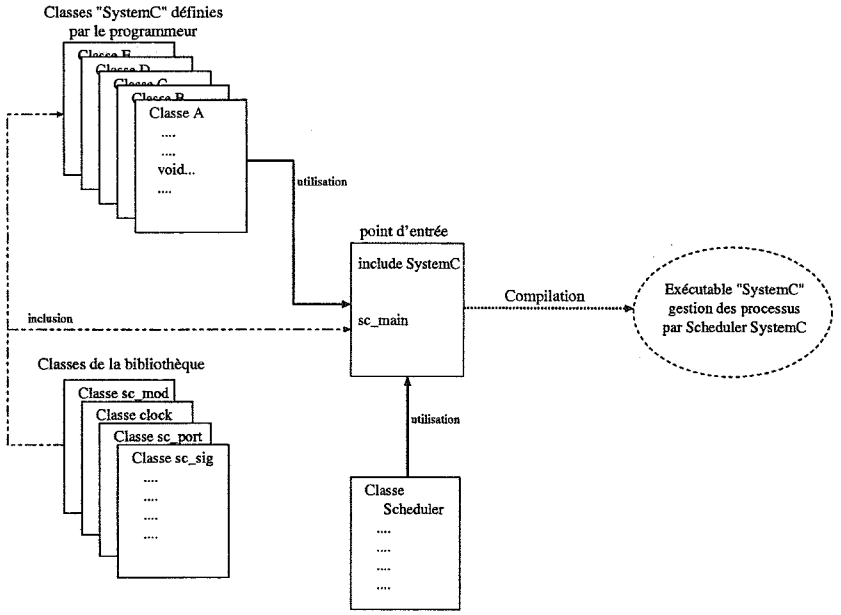


FIG. 5.1 – Création d'un programme SystemC

les différents éléments constituant un circuit électronique : modules, interfaces, canaux de communication ... Le programmeur SystemC écrit un programme C++ en utilisant les classes prédéfinies dans la bibliothèque SystemC ou en redéfinissant par héritage ses propres classes.

Comme en C++, le programmeur décrit des composants (ici dédiés à la conception de circuits) sous forme de classes avec des attributs et des méthodes propres ayant une portée réduite à la classe. Pour pouvoir utiliser ces composants, le programmeur doit créer des instances de ces classes. Il écrit donc un point d'entrée du programme qui est chargé d'instancier les différents objets utilisés. L'avantage essentiel de cette approche est le bénéfice apporté par la notion d'objet et par la méthodologie objet qui s'appliquent bien au cadre de la description de SoC. Des exemples de programmes SystemC typiques sont présentés dans l'annexe B.

Lors de la description de ce point d'entrée, l'utilisateur doit également instancier un ordonnanceur afin de simuler l'exécution de son programme. En effet, SystemC n'est pas qu'un ensemble de classes prédéfinies : la bibliothèque SystemC contient également un programme d'ordonnancement chargé de simuler l'exécution (parallèle) d'un programme. C'est le fonctionnement de cet ordonnanceur qui détermine toute la sémantique de la simulation SystemC, il est donc important de bien comprendre son fonctionnement afin d'être capable de raisonner sur des programmes SystemC.

La figure 5.1 résume la création d'un programme SystemC. La compilation d'un programme SystemC a pour unique but la simulation de son comportement et par conséquent la vérification de certaines propriétés.

Comme en VHDL, une sous-partie de SystemC peut être utilisée pour synthétiser des circuits [74]. La simulation de ces programmes reste possible et a l'énorme avantage d'être beaucoup plus rapide qu'une simulation VHDL. Dans le cadre d'une production de circuits, le schéma de compilation décrit n'est plus valable : seuls les modules décrits en SystemC "synthétisable" sont utilisés pour la synthèse du circuit.

5.1.2 Syntaxe de base

Afin de suivre les différentes étapes de raffinement de notre développement, nous allons introduire au fur et à mesure la syntaxe et les règles de programmation SystemC. La notion de module est la brique de base des programmes SystemC. Un module contient :

– des ports pour communiquer avec "l'extérieur" du module,

```
#include ''systemc.h''

SC_MODULE(my_module) {

  sc_in<bool> port1;
  sc_out<bool> port2;

  int count; // variable locale

  void proc1() {
     if (count < 10) {
       count++;
       ...
     } else {
       ...
     }
  }

  void proc2() {
     if (count < 11) {
       count = count+2;
       ...
     } else {
       ...
     }
  }

  void proc3() {
     count = 0;
     ...
  }

  SC_CTOR(my_module) {
    count = 0;
    SC_METHOD(proc1);
    ...
    SC_METHOD(proc2);
    ...
    dont_initialize();
    SC_METHOD(proc3);
    ...
    dont_initialize();
  }
};
```

FIG. 5.2 – Exemple de code SystemC (incomplet)

- des variables locales (attributs),
- des processus réalisant les traitements du module (méthodes).

La figure 5.2 présente un exemple incomplet de déclaration de module SystemC. Dans cet exemple, un module `my_module` ayant un port d'entrée (mot clé `sc_in`) `port1`, un port de sortie (mot clé `sc_out`) `port2` et un attribut `count` est déclaré. Le module contient également trois processus `proc1`, `proc2` et `proc3`. Les processus `proc1` et `proc2` incrémentent le compteur `count` alors que le processus `proc3` le réinitialise.

L'instanciation du module est décrite dans la macro `SC_CTOR` qui permet l'initialisation des attributs (on utilise un langage objet) du module et la déclaration des processus du module (mot-clé `SC_METHOD` ou `SC_THREAD`)

Le mot-clé `SC_METHOD` permet de déclarer un type de processus particulier : ce type de processus, lorsqu'il se déclenche, exécute son code sans interruption.

Le mot-clé `SC_THREAD` permet la déclaration de processus un peu différents : le processus ne s'exécute qu'une unique fois. Ce type de processus contient généralement des boucles ouvertes (`while (true) {...}`) et qui sont mises en attente afin de "rendre la main" par la méthode `wait()`.

L'instruction `dont_initialize()` suivant certaines déclarations de processus, a un impact sur l'initialisation du système lors de l'utilisation du module défini. La section suivante présentera plus en détail la phase d'initialisation d'un programme SystemC.

5.1.3 Exécution et ordonnancement en deux phases

La première étape de l'exécution d'un programme SystemC est l'élaboration. L'élaboration consiste en la création des différents objets, constituants du programme considéré, et de leurs connexions (ports, signaux, etc...). Cette étape est décrite dans le point d'entrée du programme (la fonction `main()`). Lors de l'exécution d'un programme, ce point d'entrée est d'abord exécuté permettant la création des différents objets manipulés.

Vient ensuite l'exécution proprement dite avec la phase d'initialisation : tous les processus déclarés sont exécutés sauf mention explicite du contraire (utilisation du mot-clé `dont_initialize()`). Ceci signifie qu'il est impossible de prédire le comportement du système lors de son initialisation. Ce comportement reflète assez bien la réalité électronique des choses : lors du branchement d'un circuit, il y a toujours une latence avant que le fonctionnement de celui-ci soit cohérent. Ceci est du à l'arrivée "brusque" de l'électricité dans l'ensemble du circuit qui provoque des calculs erronés. Ce phénomène est représenté par le mécanisme d'initialisation de SystemC qui exécute dans un ordre quelconque l'ensemble des processus.

Durant l'exécution, tous les processus du programme sont considérés de la même manière. Il n'y a pas de hiérarchie entre les processus ou de localisation des processus. Ainsi, le processus p d'un module spécifique m, sous-module d'un module général M, est considéré lors de l'ordonnancement de la même manière que les processus de M et, plus généralement, de la même manière que les autres processus du programme.

Après la phase d'initialisation, le fonctionnement de l'ordonnanceur SystemC se découpe en deux phases comme dans la plupart des langages synchrones. La première phase est une phase d'évaluation où les processus considérés comme exécutables sont exécutés les uns après les autres. Une fois la phase d'évaluation terminée, une phase de mise à jour a lieu. Elle consiste en l'actualisation des valeurs des variables calculées lors de la phase d'évaluation précédente et en la construction d'une nouvelle liste de processus exécutables. Une fois cette liste construite on effectue une nouvelle phase d'évaluation, etc. L'alternance d'une phase d'évaluation et d'une phase de mise à jour est appelée un δ-cycle.

Pour notre exemple, les processus `proc2` et `proc3` ne sont pas exécutés durant l'initialisation (cf figure 5.2, utilisation de `dont_initialize()` dans la déclaration de ces deux processus). Seul le processus `proc1` est considéré comme exécutable à l'initialisation du système.

5.1.4 Premier modèle abstrait : manipulation de processus

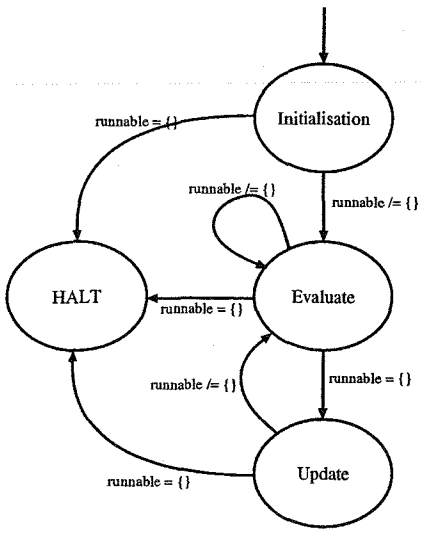
Nous nous proposons de partir de cette première description abstraite, voire simpliste, du comportement de l'ordonnanceur SystemC. L'aspect fondamental de l'ordonnanceur SystemC est donc l'alternance plus ou moins rapide entre phases d'évaluation (*evaluate*) et phases de mise à jour (*update*). Dans un premier temps, nous décrirons l'ordonnanceur comme un système manipulant des processus en ne nous souciant pas de plus de détails. Nous limiterons nos modèles à des processus de type `SC_METHOD`. La figure 5.3 présente l'automate associé à notre modèle qui considère le fonctionnement et l'arrêt du système après son initialisation. L'alternance des phases d'évaluation et de mise à jour est représentée par les deux états Evaluate et Update. L'automate de la figure 5.3 montre bien l'existence d'une boucle entre ces deux états. L'exécution d'un programme SystemC n'étant pas infinie, l'événement HALT modélise l'arrêt de l'exécution du système.

De manière plus technique, le modèle est construit de la façon suivante : on se munit d'un ensemble *PROCESSUS* qui représente l'ensemble des processus définis dans le système. Les éléments de cet ensemble sont donc les méthodes définies par le programmeur SystemC à l'aide du mot-clé `METHOD`. On se munit d'une variable *runnable* qui est le sous-ensemble des processus exécutables à un instant donné. L'ensemble *runnable* des processus exécutables est modifié par l'exécution des processus qu'il contient :

- lorsqu'un processus p est exécuté, il est supprimé de l'ensemble *runnable*,
- l'exécution de p rend de nouveaux processus exécutables ; ils sont donc ajoutés à l'ensemble *runnable*.

On se donne également une variable *phase* permettant de simuler l'état du programme :

- *init*, signifie que le système est en phase d'initialisation,
- *run*, signifie que le système est en cours d'exécution, càd dans une phase d'évaluation ou de mise à jour,
- *stop*, signifie que le système est arrêté, son exécution est terminée.



MODEL
Abstract_Scheduler
SETS
$PROCESSES$;
$STATE = \{init, run, stop\}$
VARIABLES
$runnable, phase$
INVARIANT
$runnable \subseteq PROCESSES \wedge$
$phase \in STATE \wedge$
$(phase = stop \Rightarrow runnable = \emptyset)$
INITIALISATION
$runnable :\in \mathbb{P}(PROCESSES) \parallel$
$phase := init$

FIG. 5.3 – Automate et en-tête du modèle abstrait

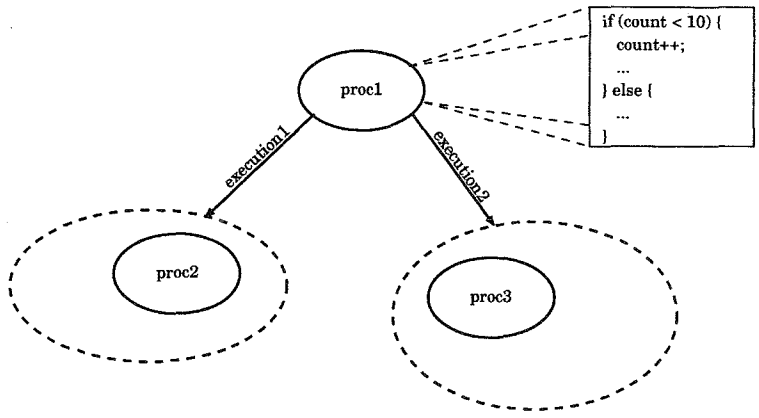


FIG. 5.4 – Différentes exécutions possibles d'un processus

Une propriété de sûreté intéressante présentée à la figure 5.3 est $phase = stop \Rightarrow runnable = \emptyset$. Cette propriété signifie que la simulation ne s'arrête que lorsque le système n'est plus actif et qu'aucun processus ne peut encore s'exécuter.

La figure 5.4 reprend l'exemple de la figure 5.2, page 123. Le processus `proc1` a plusieurs exécutions possibles en raison de la présence d'une conditionnelle dans le code de `proc1`.

Les deux événements suivants permettent la modélisation du système. L'événement Evaluate représente la phase d'exécution de l'ordonnanceur. Lorsque l'ensemble *runnable* n'est pas vide, les processus le composant s'exécutent et une nouvelle liste de processus exécutables est créée ($runnable :\in \mathbb{P}(PROCESSUS)$). La nouvelle liste de processus construite est la conséquence de l'exécution des processus précédents mais le niveau d'abstraction du modèle ne permet pas de détailler sa construction. L'événement Evaluate peut se produire de manière répétée tant que l'ensemble *runnable* n'est pas vide et, *runnable* étant systématiquement reconstruit, l'événement peut même prendre indéfiniment la main.

Si après une ou plusieurs exécutions de processus, la liste *runnable* devient vide, l'événement Update peut prendre la main : une nouvelle liste est ainsi construite. Une fois encore, l'abstraction ne permet pas de détailler les règles de construction de cette liste et l'événement Update montre la prise de contrôle du système par l'ordonnanceur qui construit une nouvelle liste de processus exécutables une fois la précédente épuisée. Comme dit précédemment, la méthode utilisée par l'ordonnanceur pour reconstruire cette liste

n'est, pour l'instant, pas précisée. Les prochains raffinements auront pour but d'expliciter l'algorithme utilisé.

```
Evaluate ≙
  SELECT
    runnable ≠ ∅ ∧
    phase ≠ stop
  THEN
    runnable := ℙ(PROCESSUS) ||
    phase := run
  END
```

```
Update ≙
  SELECT
    phase = run ∧
    runnable = ∅
  THEN
    runnable :∈ ℙ(PROCESSUS)
  END
```

Enfin, la simulation d'un programme SystemC doit pouvoir s'arrêter. Par conséquent, nous avons défini un événement B représentant l'arrêt du système, une fois les processus exécutables complètement consommés. Une fois l'événement HALT produit, les autres événements ne peuvent plus se produire (*phase* := *stop*).

```
HALT ≙
  SELECT
    phase ≠ stop ∧
    runnable = ∅
  THEN
    phase := stop
  END
```

Le modèle abstrait du système exhibe les grandes phases de l'ordonnanceur SystemC et en particulier l'alternance des phases d'évaluation et de mise à jour représentées par les deux événements Evaluate et Update. Le haut niveau d'abstraction ne permet pas d'expliciter les détails de cette alternance de phases. Cependant, le modèle abstrait permet de montrer l'existence d'une boucle "indésirable" dans la simulation. En effet, on peut très bien imaginer qu'un processus p exécutable ($p \in runnable$) soit exécuté et systématiquement replacé, suite à son exécution dans la liste des processus exécutables. De ce fait, cette liste n'est jamais vide et le système ne passe jamais en phase de mise à jour. Cette boucle fait partie de la définition de l'ordonnanceur et la simulation d'un programme présentant ce genre "d'anomalies" n'est pas problématique.

Les prochains raffinements vont consister à préciser l'événement Update afin de déterminer clairement les règles de construction de la liste de processus exécutables.

5.2 SystemC : Ordonnancement à base d'événements

Une notion importante de SystemC est l'ordonnancement à base d'événements. Les événements permettent de représenter des situations importantes du système et, par un mécanisme de liste de sensibilité, de déclencher les processus.

Une autre notion importante à définir est le traitement du temps durant une simulation SystemC. Là encore, c'est par le biais de l'ordonnanceur et des notifications événementielles que le temps s'écoule. On peut cependant préciser tout de suite que l'ordonnanceur gère une référence temporelle interne qui représente l'horloge de référence du système.

5.2.1 Ordonnanceur à base d'événements

L'ordonnanceur SystemC est un ordonnanceur fonctionnant sur la notion d'événement. Il convient de préciser ce concept et d'expliquer la gestion de l'ordonnanceur ainsi que le principe de notification.

Un événement est un objet à part entière (au sens C++). L'utilisateur peut donc créer ses propres instances d'événements. Les événements sont les outils de synchronisation de SystemC. Durant l'exécution

d'un programme, les processus agissent sur le système et leurs actions provoquent des événements par le biais d'un mécanisme de notification. La notification d'un événement peut se faire de trois manières distinctes :

- immédiate, l'événement a eu lieu à l'instant courant.
- δ, l'événement aura lieu au prochain δ cycle.
- temporelle, l'événement aura lieu à un instant futur $time + t$.

Nous préciserons par la suite les comportements provoqués pas les différents types de notifications cependant, toute notification non-immédiate est considérée comme "en attente". Un événement donné ne peut conserver qu'une notification en attente de traitement, c'est pourquoi il convient de définir un ordre de priorité entre les trois types de notifications distinctes.

Les processus décrits dans le code SystemC sont sensibles à l'occurrence de certains événements. Chaque processus possède sa propre liste de sensibilités décrivant l'ensemble des événements le concernant (voir figure 5.5 pour la syntaxe). L'ordonnanceur a la charge de sélectionner les processus à exécuter en fonction des événements ayant lieu. Les notifications permettent d'indiquer l'occurrence des événements.

Les trois types de notifications distinctes permettent la construction de l'ensemble des processus exécutables de différentes manières :

- La notification immédiate d'un événement e a pour conséquence l'ajout immédiat (dans la phase d'évaluation courante et sans passer par une phase de mise à jour préalable) de processus sensibles à l'occurrence de l'événement e.
- La δ notification d'un événement e indique que e aura lieu au δ cycle suivant. Ainsi, une fois la phase d'évaluation terminée, la liste de processus exécutables construite est constituée de l'ensemble des processus sensibles à l'apparition de l'événement e. Tous les événements ayant une δ notification en attente ont également lieu et les différents processus concernés sont ajoutés à la liste des processus exécutables.
- La notification temporelle d'un événement e permet son déclenchement à un instant précis. Le principe est le même que pour les δ notifications (construction d'une liste en phase de mise à jour), cependant une notion de temps est ajoutée. Un événement e étant concerné par une notification temporelle est censé se produire à un instant précis $time + t$ (temps relatif). Si, lors de la phase de mise à jour, aucune *delta* notification n'est en attente, l'ordonnanceur avance son horloge interne $time$ jusqu'à l'instant de notification le plus proche.

5.2.2 Priorité de notification

En SystemC, le choix de priorité des notifications se fait par rapport au temps et à l'instant courant. Nous allons lister les différentes règles décrites dans le manuel de référence SystemC :

- Une notification immédiate est prioritaire sur tout autre type de notification.
- Une δ notification est prioritaire sur une notification temporelle.
- La plus petite notification temporelle affectant un événement est toujours conservée.

On peut résumer les priorités de notification de la manière suivante :

$$timed \prec \delta \prec immediate$$

L'exemple de code présenté à la figure 5.2 était incomplet. La figure 5.5 présente le code complet du module `my_module` avec différents types de notification. Un processus ne s'exécute pas n'importe quand mais lors de l'occurrence d'événements particuliers. Chaque processus a une liste de sensibilités (mot clé `sensitive`) indiquant les événements déclenchant son exécution. Des exemples de liste de sensibilités sont également présentés dans l'exemple de la figure 5.5.

5.2.3 Premier raffinement : introduction d'événements SystemC

Dans ce premier raffinement, nous allons ajouter la notion d'événement et les différents types de notifications possibles. Il convient de respecter scrupuleusement les spécifications SystemC de l'ordonnanceur. L'ajout de ces nouveaux concepts et des fonctionnalités leur étant associés entraîne une séparation de l'événement Update en plusieurs événements distincts raffinant l'événement abstrait. Il en est de même

```
#include ''systemc.h''

SC_MODULE(my_module) {

  sc_in<bool> port1;
  sc_out<bool> port2;

  event e1,e2,e3; // declaration d' evenements

  int count; // variable locale

  void proc1() {
     if (count < 10) {
       count++;
       e2.notify(); // notification immediate
     } else {
       e3.notify(5,SC_NS); // notification temporelle
     }
  }

  void proc2() {
     if (count < 11) {
       count = count+2;
       e1.notify(SC_ZERO_TIME); // delta notification
     } else {
       e3.notify(4,SC_NS); // notification temporelle
     }
  }

  void proc3() {
     count = 0;
     e1.notify(SC_ZERO_TIME); // delta notification
  }

  SC_CTOR(my_module) {
    count = 0;
    SC_METHOD(proc1);
    sensitive << e1;
    SC_METHOD(proc2);
    sensitive << e2;
    dont_initialize();
    SC_METHOD(proc3);
    sensitive << e3;
    dont_initialize();
  }
};
```

FIG. 5.5 – Code complet du module `my_module`

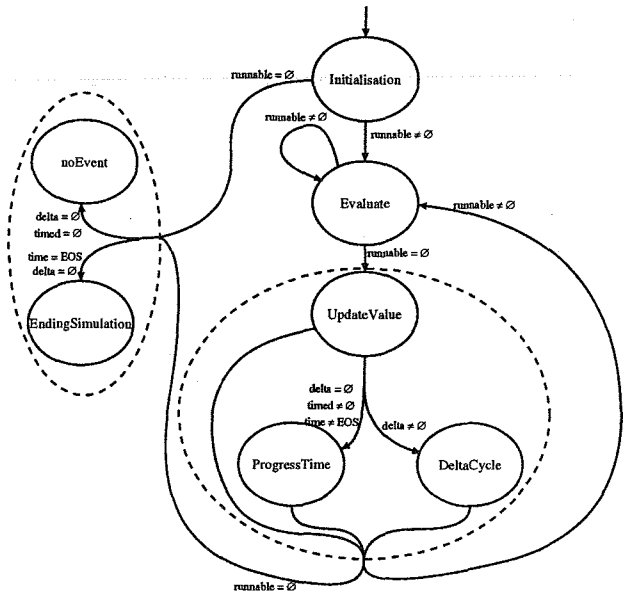


FIG. 5.6 – Automate concret de l'ordonnanceur

pour l'événement HALT qui est alors séparé pour prendre en compte deux cas de terminaisons possibles. La figure 5.6 présente le nouvel automate du système qui résulte de "l'éclatement" des anciens états en plusieurs états concrets distincts.

On introduit tout d'abord la notion importante d'événement sous la forme de l'ensemble SC_EVENTS et de la relation *sensitivity*. En effet, l'ordonnanceur SystemC est un ordonnanceur à événements et les conditions d'activation des processus reposent sur l'occurrence des événements du système.

On définit de nouvelles constantes *trigger* et *sensitivity* qui permettent de considérer l'occurrence d'événements dans l'environnement du système. La relation *trigger* introduit les notifications événementielles lors des différentes exécutions d'un processus. La figure 5.4 reste d'actualité, les différentes exécutions d'un processus provoquent différentes combinaisons d'événements qui rendent activables différents ensembles de processus.

La constante *sensitivity* est une relation entre les $PROCESSUS$ et les SC_EVENTS. La relation *sensitivity* représente les listes de sensibilités de chaque processus. Un processus peut être sensible à plusieurs événements distincts c'est pourquoi la constante *sensitivity* est définie comme une relation. SystemC propose la possibilité de créer des listes de sensibilités dynamiques, évoluant au cours du temps. A un instant t donné un processus est sensible à certains événements et l'instant suivant $t+1$, la liste de sensibilité du dit processus a été modifiée et il n'est plus sensible aux mêmes événements. Nous avons fait le choix de ne pas modéliser cette fonctionnalité de SystemC, et les listes de sensibilités sont donc *constantes*.

Les notions d'événements et de sensibilités étant introduites, il est nécessaire d'introduire les différents types de notifications. Pour cela, il faut encore ajouter la notion de temps à notre système. La constante EOS représente le temps maximum de simulation du système. Une fois ce temps atteint, la simulation est terminée [12].

[12] Une fois les événements ayant lieu à cet instant s'étant déroulés.

REFINEMENT
Event_Scheduler
REFINES
Abstract_Scheduler
SETS
SC_EVENTS
CONSTANTS
$sensitivity, trigger, EOS$
PROPERTIES
$sensitivity \in PROCESSUS \leftrightarrow SC\_EVENTS \wedge$
$trigger \in PROCESSUS \leftrightarrow \mathbb{P}(SC\_EVENTS) \wedge$
$EOS \in \mathbb{N}$
VARIABLES
$runnable, time, phase,$
$timed, \delta$

INVARIANT
$time \in \mathbb{N} \wedge$
$EOS \geq time \wedge$
$timed \in SC\_EVENTS \nrightarrow \mathbb{N} \wedge$
$\forall t.(t \in ran(timed) \Rightarrow t > time) \wedge$
$\delta \subseteq SC\_EVENTS \wedge$
$dom(timed) \cap \delta = \emptyset \wedge$
$(phase = stop \Rightarrow \delta = \emptyset) \wedge$
$(timed \neq \emptyset \wedge min(ran(timed)) \leq EOS$
$\Rightarrow phase \neq stop)$
INITIALISATION
$time := 0 \parallel$
$phase := init \parallel$
$runnable :\in \mathbb{P}(PROCESSUS) \parallel$
$timed := \emptyset \parallel$
$\delta := \emptyset$

Les nouvelles variables introduites permettent de préciser le fonctionnement de l'ordonnanceur. Les deux variables *timed* et δ représentent deux types de notifications événementielles distinctes. δ est l'ensemble des événements devant notifier le système dans le prochain δ-cycle et *timed* représente l'ensemble des événements ayant une notification temporelle. Une propriété importante du modèle est la disjonction de ces deux sous-ensembles du fait des règles de priorité établies en SystemC. Par ailleurs, le système et les événements n'ayant pas de mémoire du passé, les notifications temporelles sont toujours dans l'avenir. L'initialisation du système est conforme au manuel de référence : le temps est nul et aucune notification n'a lieu.

L'événement concret Evaluate doit maintenant préciser le comportement des événements déclenchés par l'exécution d'un processus p choisi dans la liste des processus exécutables. On considère l'ensemble E représentant l'ensemble des événements issus d'une exécution possible du processus p ($E \in trigger[\{p\}]$). Il convient de distinguer les différents types de notifications associés aux éléments de E. Ce "tas" d'événements est donc partitionné en plusieurs sous-ensembles distincts :

- i est l'ensemble des événements à notification immédiate,
- d est l'ensemble des événements à δ notification,
- t est une fonction qui a pour domaine de définition l'ensemble des événements de E à notification temporelle.

D'où la partie de garde de l'événement Evaluate qui représente la partition de E :

$E \in trigger[\{p\}] \wedge$
$t \in SC\_EVENTS \nrightarrow \mathbb{N} \wedge$
$dom(t) \subseteq E \ \wedge \ d \subseteq E \ \wedge i \subseteq E \wedge$
$dom(t) \cap d = \emptyset \ \wedge dom(t) \cap i = \emptyset \wedge$
$d \cap i = \emptyset \ \wedge \ dom(t) \cup d \cup i = E \wedge$

Le respect des propriétés invariantes implique également que les notifications temporelles représentées par t ne concernent aucun événement déjà δ-notifié durant l'exécution d'un processus p' précédent.

$newTimed \in SC\_EVENTS \nrightarrow \mathbb{N} \wedge$
$dom(newTimed) = dom(timed \ \lhd\!\!\!- \ t) - (d \cup i) \wedge$
$\forall e.(e \in (dom(timed) \cap dom(t)) \Rightarrow newTimed(e) = min(\{timed(e), t(e)\})) \wedge$
$\forall e.(e \in dom(newTimed) \wedge e \notin dom(t) \Rightarrow newTimed(e) = timed(e)) \wedge$
$\forall e.(e \in dom(newTimed) \wedge e \notin dom(timed) \Rightarrow newTimed(e) = t(e))$

La variable locale *newTimed* représente le nouvel ensemble des notifications temporelles. Cet ensemble est construit de manière à préserver les règles précédemment énoncées. La figure 5.7 illustre la construction

de cette fonction à partir de la variable *timed* et des notifications temporelles t, issues de l'exécution de p. La construction de la fonction *newTimed* est une sorte de surcharge qui, en respect avec les règles de priorité de notifications, ne conserve que les notifications temporelles les plus proches de l'instant courant. Ainsi, si un événement e donné avait déjà une notification temporelle n en attente pour l'instant $time+t$ et qu'une nouvelle notification temporelle n' pour l'instant $time+t'$ le concernant apparaît, la fonction *newTimed* contiendra la notification temporelle de e la plus proche dans le temps.

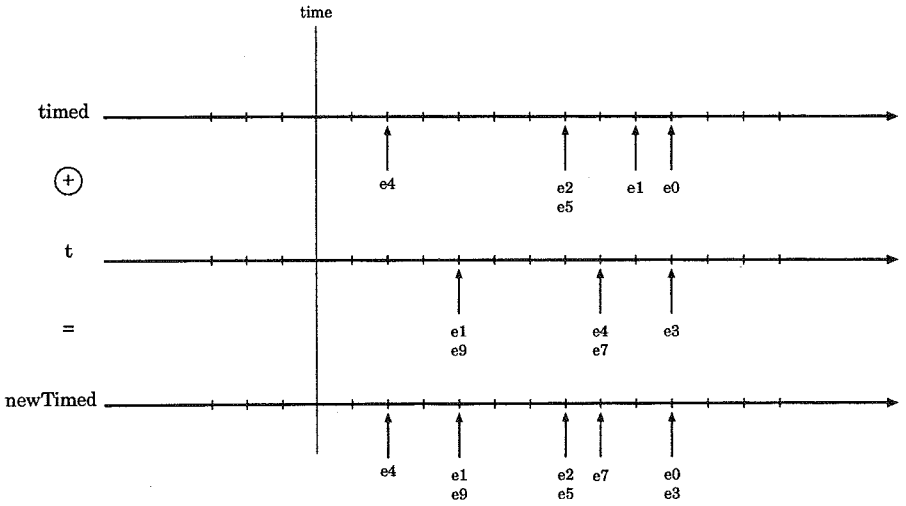


FIG. 5.7 – Construction de *newTimed*

De même, la variable δ est mise à jour en respectant les règles d'ordonnancements : les éventuelles notifications immédiates d'événements précédemment δ notifiés soient prioritaires ($\delta := \delta \cup d - i$).

Enfin, l'ensemble *runnable* est mis à jour par la suppression du processus p exécuté et par l'ajout de l'ensemble des processus sensibles à la notification immédiate des événements éléments de i, issus de l'exécution de p.

La version concrète de l'événement Evaluate est donc :

Evaluate $\widehat{=}$
ANY
$p, E, t, d, i, newTimed$
WHERE
$p \in runnable \wedge$
$t \in SC\_EVENTS \nrightarrow \mathbb{N} \wedge$
$E \in trigger[\{p\}] \wedge$
$dom(t) \subseteq E \wedge$
$newTimed \in SC\_EVENTS \nrightarrow \mathbb{N} \wedge$
$d \subseteq E \wedge i \subseteq E \wedge$
$dom(newTimed) = dom(timed \lhd\!\!- t) - (d \cup i) \wedge$
$\forall e.(e \in (dom(timed) \cap dom(t)) \Rightarrow newTimed(e) = min(\{timed(e), t(e)\})) \wedge$
$\forall e.(e \in dom(newTimed) \cap dom(timed) \Rightarrow newTimed(e) = timed(e)) \wedge$
$\forall e.(e \in dom(newTimed) \cap dom(t) \Rightarrow newTimed(e) = t(e)) \wedge$
$dom(t) \cap d = \emptyset \wedge dom(t) \cap i = \emptyset \wedge$
$dom(t) \cap \delta = \emptyset \wedge d \cap i = \emptyset \wedge$
$dom(t) \cup d \cup i = E \wedge$
$\forall x.(x \in ran(newTimed) \Rightarrow time < x)$
THEN
$runnable := (runnable - \{p\}) \cup sensitivity^{-1}[i] \parallel$
$timed := newTimed \parallel$
$\delta := \delta \cup d - i \parallel$
$phase := run$
END

On va maintenant préciser la phase de mise à jour réalisée par l'ordonnanceur. Selon les différents types de notifications, la phase de mise à jour se comporte différemment. La séparation de l'événement abstrait Update a produit trois événements concrets, updateValue, DeltaCycle, ProgressTime.

L'événement concret updateValue est un événement non-déterministe qui ajoute un ensemble S d'événements à l'ensemble des événements δ ayant une δ-notification en attente c'est-à-dire devant se produire au prochain δ cycle. Ceci signifie que, parfois, l'ordonnanceur produit de nouvelles notifications événementielles lors de la phase de mise à jour. Les détails de ce comportement seront ajoutés dans le prochain raffinement. L'invariant du modèle reste préservé par l'activation de ce nouvel événement.

updateValue $\widehat{=}$
ANY
S
WHERE
$S \subseteq SC\_EVENTS \wedge$
$runnable = \emptyset \wedge$
$S \cap dom(timed) = \emptyset \wedge$
$phase = run$
THEN
$\delta := \delta \cup S$
END

L'événement DeltaCycle représente une mise à jour du système due à des δ notifications (cf garde, $\delta \neq \emptyset$). A ce niveau d'abstraction, la mise à jour permet d'expliquer la construction d'une nouvelle liste de processus exécutables : cette liste est l'ensemble des processus sensibles aux événements contenus par δ. Une fois cette nouvelle liste construite, l'ordonnanceur considère que les événements de δ ont eu lieu et les supprime ; l'ensemble δ devient donc vide.

L'événement ProgressTime modélise la phase de mise à jour due à des notifications temporelles. Ce type de mise à jour n'a lieu que si il n'existe aucune δ notification dans le système et que l'horloge de simulation n'a pas encore atteint l'instant de terminaison de la simulation. Dans ce cas, l'horloge *time* du système est avancée jusqu'à la notification la plus proche. Les événements ayant lieu à cet instant $time + n$ sont supprimés et la liste des processus exécutables est construite à l'aide du mécanisme de sensibilité.

DeltaCycle $\widehat{=}$
SELECT
$runnable = \emptyset \wedge$
$\delta \neq \emptyset \wedge$
$phase = run$
THEN
$runnable := sensitivity^{-1}[\delta] \parallel$
$\delta := \emptyset$
END

ProgressTime $\widehat{=}$
SELECT
$runnable = \emptyset \wedge$
$\delta = \emptyset \wedge$
$timed \neq \emptyset \wedge$
$EOS \geq min(ran(timed)) \wedge$
$phase = run$
THEN
$runnable :=$
$sensitivity^{-1}[timed^{-1}[\{min(ran(timed))\}]] \parallel$
$time := min(ran(timed)) \parallel$
$timed := timed \rhd \{min(ran(timed))\}$
END

Enfin, la simulation doit s'arrêter quand le temps de simulation est écoulé ou que le système n'a plus d'activité. En effet, le temps ne progressant que par le biais des notifications temporelles, il se peut que l'horloge de notre système n'atteigne jamais le temps de fin de simulation. Ce cas d'arrêt est modélisé par l'événement noEvent. Sinon, lorsque l'horloge du système atteint le temps de fin de simulation, l'exécution est terminée. Il y a donc deux cas de terminaison possibles qui sont issus de l'événement abstrait HALT.

EndingSimulation =
SELECT
$runnable = \emptyset \wedge$
$\delta = \emptyset \wedge$
$time = EOS \wedge$
$phase \neq stop$
THEN
$phase := stop$
END

noEvent =
SELECT
$runnable = \emptyset \wedge$
$\delta = \emptyset \wedge$
$timed = \emptyset \wedge$
$phase \neq stop$
THEN
$phase := stop$
END

La figure 5.8 présente une animation d'une partie du programme SystemC de la figure 5.5 (page 128) par notre modèle. On peut voir évoluer l'ensemble δ et la fonction *timed* selon l'exécution des processus particuliers du programme. La partie gauche de la figure montre l'état du système au moment de l'exécution du processus `proc2` :

- le processus `proc2` appartient à *runnable*.
- la valeur de la variable `count` est 8. A la fin de l'exécution de `proc2`, sa valeur sera 10.
- l'exécution de `proc2` provoque l'ajout de l'événement `e1` à l'ensemble δ (δ notification de `e1`).

La partie droite de la figure présente l'état du système au δ cycle suivant. Le processus `proc1` va maintenant s'exécuter. Son exécution va provoquer la notification temporelle de `e3` qui est donc ajoutée au domaine de *timed*. La simulation se poursuit par la mise à jour de l'horloge du système et par l'exécution du processus `proc3`.

Ce premier raffinement introduit l'ensemble des règles régissant l'ordonnanceur SystemC. Il peut donc servir de base pour raisonner sur les programmes et pour détecter d'éventuels programmes "qui bouclent" en phase d'évaluation. Ce modèle reste cependant un peu vague sur la phase de mise à jour notamment en ce qui concerne l'événement updateValue. La prochaine section et le prochain raffinement vont permettre de lever les dernières ambiguïtés du modèle en introduisant les principes de communication entre modules de SystemC.

5.3 SystemC : communications entre modules

SystemC permet de décrire de vastes systèmes complexes sous la forme de modules, pouvant éventuellement comprendre d'autres modules, qui encapsulent des processus chargés de réaliser les différents traitements du système. Le découpage d'une application en différents modules implique des moyens de

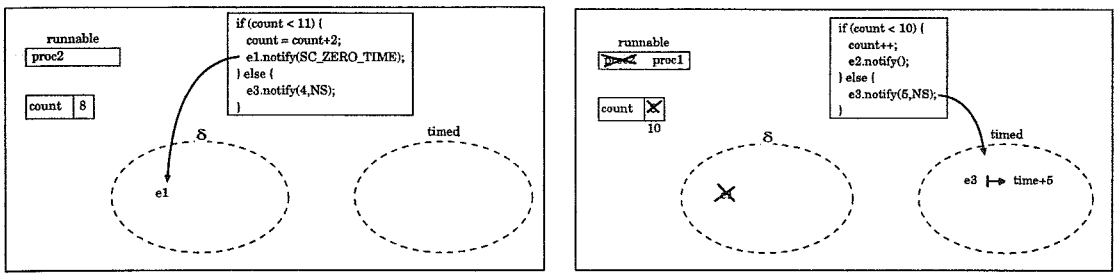


FIG. 5.8 – Modélisation de l'exécution

communication. Les outils proposé par SystemC afin de simuler la communication interne et externe des modules sont les signaux et les événements SystemC. Nous allons présenter plus en détail les signaux, objets de communication.

5.3.1 Signaux : le média de communication

Un signal est un objet C++ (instance de la classe `signal`) chargé de représenter un média de communication. La lecture de la valeur d'un signal se fait par l'intermédiaire de la méthode `read()`. L'écriture directe d'une valeur sur un signal se fait par l'intermédiaire de la méthode `write()` ayant pour paramètre la valeur écrite.

Les objets de type `signal` sont également affectés par la simulation en deux phases de SystemC. La valeur d'un signal n'est renouvelée, lors de la simulation, qu'en phase de mise à jour, une fois l'ensemble des processus exécutables vide. Ceci signifie que les consultations de la valeur d'un signal durant la phase d'évaluation ne prennent pas en compte les éventuelles modifications de valeur ayant eu lieu durant cette phase d'évaluation. En cas de multiples affectations ou modifications d'un signal durant la même phase d'évaluation, seule la dernière modification sera considérée lors de la phase de mise à jour.

Les signaux permettant la communication de modules sont reliés à un port de sortie et un port d'entrée. La partie gauche de la figure 5.9 présente un exemple de cette utilisation avec le signal `canal1` qui relie les deux modules présentés. La connexion d'un signal à un port d'entrée et de sortie n'est pas obligatoire et lorsque les règles de visibilité et de portée des variables le permettent, les processus peuvent utiliser les signaux de manière directe. La partie droite de la figure 5.9 présente ce cas de figure. Dans ce cas, le signal est considéré comme une variable locale du module et sa portée est limitée à celui-ci. Seuls les processus (les méthodes de la classe) de ce module peuvent utiliser le signal en question.

5.3.2 Sensibilité des processus

La description de liste de sensibilités peut être plus complexe qu'une simple liste d'événements. En effet, les processus SystemC peuvent être également sensibles à la modification de valeur de signaux. Dans ce cas, la sensibilité indique que le processus doit être exécuté lors d'un changement de valeur de l'objet listé. Dans ce contexte, il existe trois types de sensibilité distincts :

- sensibilité générale avec le même mot-clé que précédemment `sensitive`. Ce mot-clé permet de mélanger différents types d'objets (événements, signaux et ports). Le processus doit être exécuté lors d'un changement de valeur de l'objet concerné (signal).
- sensibilité sur front montant. Ce type de sensibilité ne s'applique que sur des objets (ports, canaux) de type booléen. Le processus doit être exécuté lors d'un changement de valeur impliquant un front montant (passage de false à true ou de 0 à 1). Le mot-clé utilisé est alors `sensitive_pos`.
- sensibilité sur front descendant. Idem que précédemment mais pour un passage de true à false. Le mot-clé utilisé est `sensitive_neg`.

La figure 5.9 présente les liens existants entre les signaux, les événements et les processus SystemC. La modification de la valeur d'un signal est associée à un événement : toute modification de ce signal provoque la δ notification de celui-ci. La partie gauche de la figure 5.9 montre l'exemple de deux processus

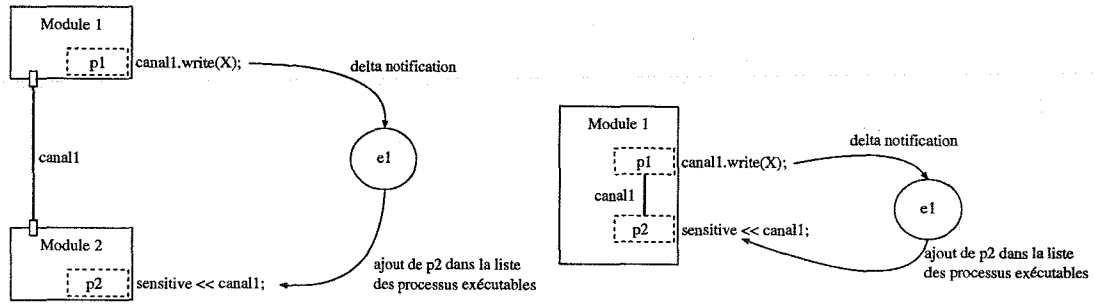


FIG. 5.9 – Modification de valeur d'un signal

```
#include ''systemc.h''

SC_MODULE(my_module) {

  sc_in<bool> port1;
  sc_out<bool> port2;

  event e2,e3; // declaration d' evenements

  sc_signal<int> count; // signal local

  void proc1() {
     if (count.read() < 10) {
       count.write(count.read()+1);
       e2.notify(); // notification immediate
     } else {
       e3.notify(5,SC_NS); // notification temporelle
     }
  }

  void proc2() {
     if (count.read() < 11) {
       count.write(count.read()+2);
     } else {
       e3.notify(4,SC_NS); // notification temporelle
     }
  }

  void proc3() {count.write(0);}
}

  SC_CTOR(my_module) {
    count.write(0);
    SC_METHOD(proc1);
    sensitive << count;
    SC_METHOD(proc2);
    sensitive << e2;
    dont_initialize();
    SC_METHOD(proc3);
    sensitive << e3;
    dont_initialize();
  }
};
```

FIG. 5.10 – Exemple de code SystemC avec signaux

qui sont définis dans des modules distincts mais le principe de notification d'un changement de valeur de signal reste inchangé pour différents processus d'un même module connectés par des signaux internes, voir partie droite de la figure 5.9.

On peut reprendre l'exemple présenté figure 5.2 et ré-écrire un nouveau code SystemC ayant un comportement *relativement proche* mais utilisant des listes de sensibilités incluant des signaux. Les dangers de ce genre d'approximation sont décrits plus en détail par la suite. En appliquant une telle transformation on obtient le code présenté à la figure 5.10.

5.3.3 Remarques

Il est important de souligner que les deux codes présentés n'ont pas exactement le même comportement lors de leur simulation : les deux codes ne sont pas équivalents. La variable locale `count` est maintenant un signal (entier) et sa modification entraîne des δ notifications. Le processus `proc1` est désormais sensible à la modification de valeur de ce signal, et il est donc averti de tout changement. Ce nouveau programme a un comportement différent du précédent lors de l'exécution de la partie `then` du processus `proc1`. En effet, l'instruction `count.write(count.read()+1)` ; provoque un changement de valeur du signal `count` qui a pour conséquence la création d'une δ-notification. Au prochain δ-cycle tous les processus sensibles à une modification de `count` seront exécutés et en particulier `proc1`. Dans ce programme SystemC les deux processus `proc1` et `proc2` se passent la main pour incrémenter `count` mais le processus `proc1` n'a pas besoin de l'exécution du processus `proc2` pour reprendre la main (contrairement au premier programme).

Ces remarques prennent une importance considérable quand on se place dans le cadre d'un développement réel de SoC. Lors de la conception du SoC, les concepteurs décrivent le système en SystemC de manière abstraite sans décider a priori des parties matérielles et logicielles de l'application. Une fois cette description conforme au cahier des charges, les concepteurs partitionnent le système en blocs logiciels et matériels. La description SystemC abstraite des blocs matériels ne permet pas la synthèse immédiate des circuits. Les concepteurs doivent donc "raffiner" leur description SystemC et ré-écrire celle-ci avec un sous-langage SystemC "synthétisable". Ils opèrent par conséquent un certain nombre de transformations sur le programme initial sans garantie.

5.3.4 Algorithme complet de l'ordonnanceur

L'algorithme d'ordonnancement de SystemC tel qu'il apparaît dans le manuel de référence SystemC [88] est le suivant :

1. *Phase d'initialisation :* exécution de l'ensemble des processus (sauf mention explicite du contraire).
2. *Phase d'évaluation :* Choisir un processus dans l'ensemble des processus exécutables et l'exécuter. L'ordre dans lequel les processus sont choisis n'est pas spécifié.
   - l'exécution du processus choisi peut provoquer des notifications immédiates d'événements pouvant entraîner un ajout de processus dans l'ensemble des processus exécutables.
   - supprimer le processus sélectionné.
3. Répéter l'étape 2 tant qu'il existe des processus exécutables.
4. *Phase de mise à jour :* Exécuter tous les appels à la fonction `update()` provenant de la phase d'évaluation (concernant les signaux).
5. S'il existe des δ notifications, déterminer quels processus sont exécutables et retourner à l'étape 2.
6. S'il n'y a pas de notifications temporelles ou que la prochaine notification temporelle se situe après le temps de fin de simulation (EOS) alors la simulation est finie.
7. Sinon, avancer le temps de simulation jusqu'à l'instant de notification temporelle le plus proche.
8. Déterminer l'ensemble des processus exécutables et retourner à l'étape 2.

5.3.5 Dernier modèle : prise en compte des signaux

Les signaux sont introduits dans ce raffinement terminal. Le nouvel automate du système n'est pas présenté car il reste très similaire au précédent : seules quelques transitions sont renforcées (voir gardes des événements) afin de considérer l'ajout des signaux et de leurs valeurs.

De nouvelles constantes sont introduites dans ce raffinement. Le nouvel ensemble $CHANNELS$ représente l'ensemble des signaux utilisés dans un programme SystemC. Un autre ensemble $VALUE$ représente l'ensemble des valeurs (abstraites) prises par les signaux considérés. Enfin, notre modèle introduit le sous-ensemble C_EVENTS qui représente l'ensemble des événements implicites du programme. Un événement implicite est un événement SystemC non manipulable directement par le programmeur et qui indique une modification de valeur d'un signal. Nous introduisons les propriétés suivantes sur ces constantes et ensembles :

$$C\_EVENTS \subseteq SC\_EVENTS \wedge$$
$$\forall S.(S \in ran(trigger) \Rightarrow S \cap C\_EVENTS = \emptyset)$$

D'autres constantes sont également introduites dans ce modèle. La fonction *produce* est une fonction totale qui représente le déclenchement des événements implicites lors de la modification de valeur des signaux. Notre modèle utilise des signaux abstraits et ne détaille pas la génération des événements implicites pour les listes de sensibilités négatives et positives construites à partir des mots-clés `sensitive_pos` et `sensitive_neg`. Ce type de liste de sensibilités concerne les signaux de type booléen et nous préciserons la modélisation de ce type d'objet lors de notre exemple d'instanciation.

Une nouvelle variable *value* est ajoutée et représente la valeur courante des signaux. Une autre variable *newValue* est également introduite afin de représenter la nouvelle valuation de l'ensemble des signaux avant la phase de mise à jour. Ces deux variables sont donc des fonctions totales de $CHANNELS$ dans $VALUE$.

La version concrète de l'événement Evaluate prend en compte les signaux. Une fonction partielle f est introduite et représente les modifications de valeurs apportées aux signaux lors de l'exécution du processus p. La fonction *newValue* est construite à l'aide de cette fonction f durant la phase d'évaluation, reflétant ainsi les modifications apportées aux signaux par les exécutions des différents processus exécutables. Il est facile de montrer que la nouvelle version de l'événement Evaluate raffine la version abstraite.

Evaluate =
ANY
$p, E, t, d, i, newTimed, f$
WHERE
$p \in runnable \wedge$
$t \in EVENTS \nrightarrow \mathbb{N} \wedge$
$E \in trigger[\{p\}] \wedge$
$dom(t) \subseteq E \wedge$
$newTimed \in EVENTS \nrightarrow \mathbb{N} \wedge$
$d \subseteq E \wedge i \subseteq E \wedge$
...
$f \in CHANNELS \nrightarrow VALUE$
THEN
$runnable := (runnable - p) \cup sensitivity^{-1}[i] \parallel$
$timed := newTimed \parallel$
$\delta := \delta \cup d - i \parallel$
$newValue := newValue \leftarrow f \parallel$
$phase := run$
END

Enfin, la nouvelle version de l'événement updateValue est déterministe et explique l'ajout de δ notifications durant la phase de mise à jour. Lorsque les valeurs des canaux sont mises à jour, des δ notifications apparaissent si la nouvelle valeur des signaux est différente de l'ancienne. Les processus sensibles à la modification de valeur des signaux peuvent être activés. Cet événement explique le comportement de SystemC en cas d'affectation multiple d'un même signal durant la phase d'évaluation : seule la dernière affectation est conservée.

updateValue =
SELECT
$runnable = \emptyset \wedge$
$value \neq newValue \wedge$
$phase = run$
THEN
$value := newValue \parallel$
$\delta := \delta \cup produce\left[\left\{ c \mid c \in CHANNELS \wedge value(c) \neq newValue(c) \right\}\right]$
END

Les autres événements ne sont pas sensiblement différents des versions abstraites c'est pourquoi nous

ne les présenterons pas ici. Ce dernier raffinement permet la prise en compte de l'ensemble des mécanismes de synchronisation et de simulation de SystemC. Notre développement nous a permis d'écarter un certain nombre de fausses interprétations et nous permet de fixer clairement la sémantique de simulation SystemC. Le fonctionnement de l'ordonnanceur, basé sur l'occurrence d'événements, est explicité et nous pouvons raisonner sur les programmes SystemC. En effet, notre développement est générique et peut être instancié pour l'étude d'un programme spécifique. La section suivante présentera les intérêts de l'instanciation dans ce cadre ainsi qu'un exemple-jouet d'une telle instanciation.

Nous allons maintenant comparer les deux programmes SystemC servant d'illustrations à nos propos à l'aide de ce nouveau modèle de l'ordonnanceur. Les tableaux ci-dessous présentent, en détaillant à l'aide des nouvelles variables du modèle, l'exécution des deux programmes SystemC de la figure 5.5 et 5.10. Le premier tableau décrit l'exécution simulée du code de la figure 5.5 et le second l'exécution simulée du code de la figure 5.10.

Le premier tableau présente clairement l'alternance des deux processus `proc1` et `proc2` ainsi que l'incrémentation de la variable `count` jusqu'à sa réinitialisation par le processus `proc3`. L'ensemble δ ne peut contenir que l'événement $e1$ durant l'exécution : les seules δ-notifications apparaissant dans le programme concernent $e1$. De même, la fonction *timed* évolue au cours de l'exécution : l'événement $e3$ appartient au domaine de la fonction lorsque `proc3` a exécuté la partie else de son code. L'exécution de ce programme se termine lorsque la notification temporelle de $e3$ est postérieure à l'instant de fin de simulation.

Phase	*runnable*	δ	*timed*	*count*	*time*
Init	$\{proc1\}$	$\emptyset$	$\emptyset$	0	0
Evaluate	$\{proc2\}$	$\emptyset$	$\emptyset$	1	0
Evaluate	$\emptyset$	$\{e1\}$	$\emptyset$	3	0
DeltaCycle	$\{proc1\}$	$\emptyset$	$\emptyset$	3	0
Evaluate	$\{proc2\}$	$\emptyset$	$\emptyset$	4	0
Evaluate	$\emptyset$	$\{e1\}$	$\emptyset$	6	0
DeltaCycle	$\{proc1\}$	$\emptyset$	$\emptyset$	6	0
Evaluate	$\{proc2\}$	$\emptyset$	$\emptyset$	7	0
Evaluate	$\emptyset$	$\{e1\}$	$\emptyset$	9	0
DeltaCycle	$\{proc1\}$	$\emptyset$	$\emptyset$	9	0
Evaluate	$\{proc2\}$	$\emptyset$	$\emptyset$	10	0
Evaluate	$\emptyset$	$\{e1\}$	$\emptyset$	12	0
DeltaCycle	$\{proc1\}$	$\emptyset$	$\emptyset$	12	0
Evaluate	$\emptyset$	$\emptyset$	$\{e3 \mapsto time + 5\}$	12	0
ProgressTime	$\{proc3\}$	$\emptyset$	$\emptyset$	12	5
Evaluate	$\emptyset$	$\{e1\}$	$\emptyset$	0	5
...	...	...	...	...	...
HALT	$\emptyset$	$\emptyset$	$\{e3 \mapsto time + 5\}$	12	??

Une trace d'exécution de programme SystemC de la figure 5.10 est présentée ci-dessous. La déclaration de `count` comme un signal particulier a un impact important sur la sémantique du programme.

Phase	*runnable*	δ	*timed*	*value(count)*	*newValue(count)*	*time*
Init	$\{proc1\}$	$\emptyset$	$\emptyset$	0	0	0
Evaluate	$\{proc2\}$	$\{e\}$	$\emptyset$	0	1	0
Evaluate	$\emptyset$	$\{e\}$	$\emptyset$	0	2	0
DeltaCycle	$\{proc1\}$	$\emptyset$	$\emptyset$	2	2	0
Evaluate	$\{proc2\}$	$\{e\}$	$\emptyset$	2	3	0
Evaluate	$\emptyset$	$\{e\}$	$\emptyset$	2	4	0
DeltaCycle	$\{proc1\}$	$\emptyset$	$\emptyset$	4	4	0
Evaluate	$\{proc2\}$	$\{e\}$	$\emptyset$	4	5	0
Evaluate	$\emptyset$	$\{e\}$	$\emptyset$	4	6	0
DeltaCycle	$\{proc1\}$	$\emptyset$	$\emptyset$	6	6	0
Evaluate	$\{proc2\}$	$\{e\}$	$\emptyset$	6	7	0
Evaluate	$\emptyset$	$\{e\}$	$\emptyset$	6	8	0
DeltaCycle	$\{proc1\}$	$\emptyset$	$\emptyset$	8	8	0
Evaluate	$\{proc2\}$	$\{e\}$	$\emptyset$	8	9	0
Evaluate	$\emptyset$	$\{e\}$	$\emptyset$	8	10	0
DeltaCycle	$\{proc1\}$	$\emptyset$	$\emptyset$	10	10	0
Evaluate	$\emptyset$	$\emptyset$	$\{e3 \mapsto time + 5\}$	10	10	0
ProgressTime	$\{proc3\}$	$\emptyset$	$\emptyset$	10	10	5
Evaluate	$\emptyset$	$\{e1\}$	$\emptyset$	10	0	5
...	...	...	...	...	...	...
HALT	$\emptyset$	$\emptyset$	$\{e3 \mapsto time + 5\}$	10	10	??

Du fait de la déclaration de `count` en tant que signal, la sémantique des deux programmes SystemC est très différente : les bornes des variables ne sont pas les mêmes et le cycle défini par l'alternance des deux processus `proc1` et `proc2` est plus long.

5.4 Raisonner sur les programmes SystemC

Les modèles, que nous avons établis, fixent clairement la sémantique de simulation de SystemC. Notre idée est de nous servir de ces modèles pour raisonner sur les programmes. En effet, SystemC s'est imposé comme le langage de description de SoC de référence permettant de décrire des modèles plus ou moins abstraits du SoC (TLM, RTL, etc...) et de les simuler. Cependant le lien entre les différentes descriptions du même SoC n'est pas fait de manière formelle. Dans [74], la méthodologie à suivre pour obtenir du code SystemC synthétisable depuis une description plus abstraite (niveau TLM par exemple) est :

- rendre synthétisable une construction qui ne l'est pas en ré-écrivant celle-ci,
- simuler le nouveau programme,
- vérifier la "compatibilité" de la nouvelle version avec l'ancienne,
- recommencer pour chaque construction non-synthétisable.

Nous considérons cette méthodologie comme insuffisante et ne permettant pas de garantir une cohérence réelle et suffisante entre les modèles au vu de l'imprécision de la sémantique SystemC.

A partir d'un programme SystemC, nous nous proposons de construire un modèle B événementiel qui représente l'exécution de ce programme par l'ordonnanceur. Ce modèle B est une simple instanciation de nos modèles abstraits de l'ordonnanceur. Pour deux programmes SystemC différents, nous pouvons alors construire des modèles B différents et comparer ces modèles (raffinement ou autres) afin de conclure sur les différences de comportement entre les programmes.

5.4.1 Instanciation de modèles

Nous avons présenté dans le chapitre 1, section 2.4 le principe de l'instanciation de projet B événementiel. Dans notre cas actuel, l'instanciation peut se réaliser de manière extrêmement simple : le contexte abstrait (ensembles, constantes, propriétés) est spécifié pour devenir le contexte effectif, et l'événement abstrait Evaluate est séparé en différents événements simulant l'exécution des différents processus du programme.

Pour chaque processus SystemC d'un programme spécifique nous pouvons produire des événements (au moins un) qui représentent l'exécution du processus. L'ensemble des événements produits doit raffiner l'événement abstrait Evaluate qui représente l'exécution abstraite des processus. Les autres événements des modèles abstraits restent inchangés car ils reflètent le fonctionnement de l'ordonnanceur qui ne varie pas avec le programme simulé.

Les ensembles abstraits et les constantes sont concrétisés avec les valeurs particulières du programme modélisé. Comme annoncé dans la section 2.4, les valeurs concrètes du modèle instancié doivent préserver les propriétés des ensembles et constants abstraits. Par exemple, l'ensemble concret $CHANNELS$ va être composé de l'ensemble des signaux utilisés dans le programme SystemC étudié.

5.4.2 Exemple jouet

Pour illustrer notre propos, nous allons utiliser l'exemple jouet de la figure 5.10 page 135. Nous allons proposer une instanciation du modèle générique de l'ordonnanceur SystemC représentant l'exécution du programme exemple. La construction de cette instanciation se fait manuellement et est détaillée plus en détails dans la suite de ce chapitre. Nous allons instancier l'ordonnanceur abstrait avec les ensembles concrets suivants :

$$
\begin{array}{l}
PROCESSES = \{proc1, proc2, proc3\} \\
SC\_EVENTS = \{e, e2, e3\} \\
CHANNELS = \{count\} \\
VALUE == \mathbb{N}
\end{array}
$$

L'ensemble $PROCESSUS$ est instancié avec les trois processus `proc1, proc2, proc3` de l'exemple. L'ensemble abstrait SC_EVENTS est instancié avec les événements $e2$ et $e3$, définis par le programmeur, et avec l'événement e qui est un événement implicite. De la même façon, l'ensemble abstrait $CHANNELS$ est instancié avec l'unique signal *count*. L'événement implicite e est relié au signal *count* par la relation *produce*. Enfin, l'ensemble abstrait $VALUE$ est instancié avec l'ensemble des naturels.

Les constantes sont instanciées pour le programme spécifique considéré. Il est facile de montrer que les constantes d'instanciation ont les mêmes propriétés que les constantes abstraites. En effet, la majeure partie des propriétés des constantes sont de simples propriétés de typage que les constantes instanciées préservent trivialement.

$$
\begin{array}{l}
C\_EVENTS = \{e\} \\
produce = \{count \mapsto e\} \\
trigger = \{proc1 \mapsto \{e2\}, \\
\qquad\qquad proc1 \mapsto \{e3\}, \\
\qquad\qquad proc2 \mapsto \emptyset, \\
\qquad\qquad proc2 \mapsto \{e3\}, \\
\qquad\qquad proc3 \mapsto \emptyset\} \\
sensitivity = \{proc1 \mapsto e, proc2 \mapsto e2, proc3 \mapsto e3\}
\end{array}
$$

VARIABLES
$runnable, time, phase,$
$\delta, timed, value, newValue$
INVARIANT
$\mathbf{dom}(timed) \subseteq \{e3\} \ \wedge \ e3 \notin \delta \wedge e2 \notin \delta$
INITIALISATION
$time := 0 \ \|$
$runnable := \{\text{proc1}\} \ \|$
$phase := init \ \|$
$timed := \emptyset \ \| \delta := \emptyset \ \|$
$value := \{count \mapsto 0\} \ \|$
$newValue := \{count \mapsto 0\}$

L'invariant du modèle instancié permet de caractériser les variables. Nous pouvons donc prouver un invariant spécifique à l'exécution du programme étudié. Dans notre exemple jouet, l'invariant du modèle précise que seul l'événement $e3$ est concerné par des notifications temporelles. Ceci se traduit par le prédicat $\mathbf{dom}(timed) \subseteq \{e3\}$.

La mécanique de l'ordonnanceur SystemC ne change pas avec les programmes simulés. Dans nos précédents modèles, seul l'événement B Evaluate a un rapport direct avec le code simulé. Lors d'une instanciation de l'ordonnanceur abstrait pour un programme particulier, l'événement B Evaluate doit être

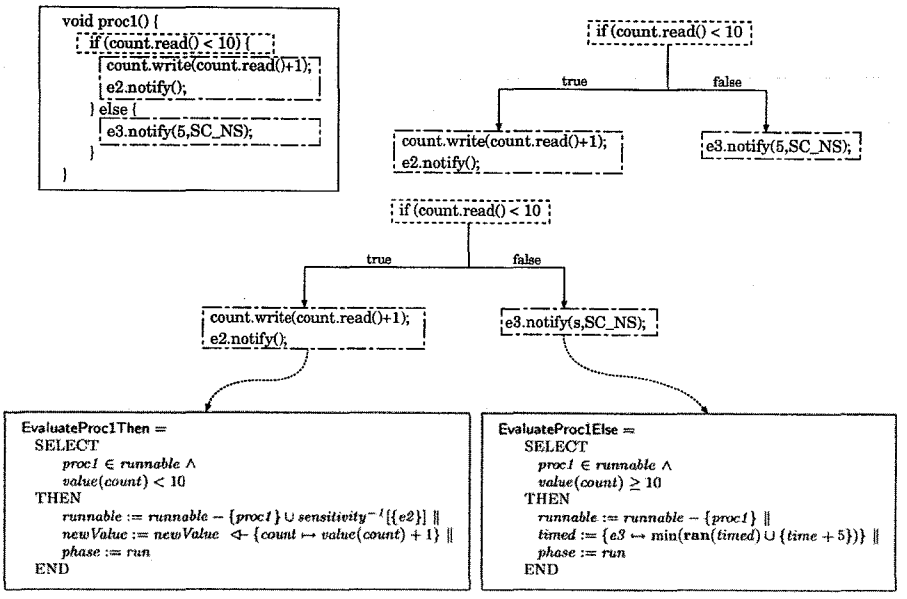


FIG. 5.11 – Génération d'événements B depuis du code SystemC

"raffiné" afin de modéliser l'exécution spécifique des processus définis. Pour cela, il convient d'analyser le code SystemC par une "analyse de bloc" simpliste afin d'extraire les différentes exécutions possibles de chaque processus SystemC défini. Un événement B modélise une exécution particulière d'un processus du code SystemC simulé.

Nous pouvons déduire de la structure des processus de l'exemple, page 135, cinq événements B :

- `proc1` est constitué d'une conditionnelle :
  - `count.read() < 10`, le signal `count` est incrémenté de 1 provoquant une δ-notification $\Rightarrow$ **1 événement B.**
  - `count.read() >= 10`, une notification temporelle de `e3` est postée $\Rightarrow$ **1 événement B.**
- `proc2` est également constitué d'une conditionnelle :
  - `count.read() < 11`, le signal `count` est incrémenté de 2 provoquant une δ-notification $\Rightarrow$ **1 événement B.**
  - `count.read() >= 11`, une notification temporelle de `e3` est postée $\Rightarrow$ **1 événement B.**
- `proc3` remet le signal count à 0 provoquant une δ-notification $\Rightarrow$ **1 événement B.**

Il convient donc de découper l'événement abstrait Evaluate en 5 événements concrets distincts modélisant les exécutions des processus de notre exemple. La figure 5.11 illustre l'analyse de flot simple que nous réalisons sur le code SystemC afin de construire les événements B. L'analyse effectuée peut être vue comme l'inverse des merging rules de Jean-Raymond Abrial. Au lieu de reconstruire un algorithme à partir des différents événements d'un modèle, nous retrouvons les événements décrivant les différents comportements d'un "algorithme".

EvaluateProc1Then =
SELECT
proc1 ∈ *runnable* ∧
value(*count*) < 10
THEN
runnable := *runnable* − {*proc1*}∪
sensitivity^{-1}[{*e2*}] ||
newValue := *newValue* ⩤
{*count* ↦ *value*(*count*) + 1} ||
phase := *run*
END

EvaluateProc1Else =
SELECT
proc1 ∈ *runnable* ∧
value(*count*) ≥ 10
THEN
runnable := *runnable* − {*proc1*} ||
timed := {*e3* ↦
min(**ran**(*timed*) ∪ {*time* + 5})} ||
phase := *run*
END

Les deux événements précédents simulent l'exécution du processus `proc1` de l'exemple. Les instructions des deux blocs de la conditionnelle sont "traduits" dans l'événement correspondant. Les gardes des deux événements représentent le test réalisé dans la conditionnelle et le contexte de simulation. La règle de priorité entre notifications temporelles est bien respectée et les modifications de valeur du signal `count` se répercutent dans la variable *newValue*.

De la même façon, les deux événements suivants modélisent le comportement du processus `proc2` durant la simulation du programme. Ces deux événements définissent une sémantique opérationnelle du processus `proc2`.

EvaluateProc2Then =
SELECT
proc2 ∈ *runnable* ∧
value(*count*) < 15
THEN
runnable := *runnable* − {*proc2*} ||
newValue := *newValue* ⩤
{*count* ↦ *value*(*count*) + 2} ||
phase := *run*
END

EvaluateProc2Else =
SELECT
proc2 ∈ *runnable* ∧
value(*count*) ≥ 15
THEN
runnable := *runnable* − {*proc2*} ||
timed := {*e3* ↦
min(**ran**(*timed*) ∪ {*time* + 4})} ||
phase := *run*
END

Enfin, l'événement EvaluateProc3 représente l'exécution du processus `proc3` qui réinitialise le signal `count` à la valeur 0.

EvaluateProc3 =
SELECT
proc3 ∈ *runnable*
THEN
runnable := *runnable* − {*proc3*} ||
newValue := *newValue* ⩤ {*count* ↦ 0} ||
phase := *run*
END

Les différents événements présentés sont tous des raffinements de l'événement abstrait Evaluate et simulent l'exécution des processus du module `my_module` durant une simulation SystemC. Le comportement du module est ainsi clairement explicité et un invariant qui lui est propre peut être spécifié afin de garantir son fonctionnement.

5.5 Conclusions

Le langage SystemC, outil important dans la construction de SoC, permet l'écriture de descriptions plus ou moins abstraites (raffinement) de systèmes. La sémantique des programmes SystemC est donnée

par l'ordonnanceur SystemC qui gère l'exécution (ou simulation) des programmes. Le principal avantage de SystemC est son temps de simulation très court (c'est une simple exécution logicielle) qui permet, dans un temps donné, une plus grande couverture de l'espace de tests. Cependant, la sémantique du langage n'est définie que par la simulation de la description SystemC par l'ordonnanceur SystemC.

Notre modélisation de l'ordonnanceur a tout d'abord rempli le rôle pédagogique "traditionnel" de la plupart des développements B. Nous avons, en particulier grâce au raffinement, mieux compris et expliqué le fonctionnement et les principes de la simulation SystemC. La simulation de programmes SystemC est basée sur la notification événementielle, et la compréhension fine du mécanisme de δ cycle apportée par la modélisation nous permet un meilleur contrôle de la sémantique des programmes SystemC. En effet, la bibliographie du langage n'est pas très explicite sur ce genre de comportement.

Le tableau ci-dessous résume le nombre et la nature des obligations de preuves associés aux différents modèles du développement. La modélisation réalisée, ainsi que l'utilisation d'un *theorem prover*, permet de démontrer automatiquement un grand nombre d'obligations de preuves. Comme le montre le tableau ci-dessous, c'est seulement lors de l'instanciation que les obligations de preuves deviennent plus techniques. Dans ce cas, le prouveur automatique échoue à cause du nombre important de propriétés qui bruitent la preuve automatique. Ces obligations de preuves restent, en général, assez simples et quelques interactions suffisent à réaliser la preuve.

Modèles B	**Preuves Automatiques**	**Preuves Interactives**	**% Preuves auto/inter**
AbstractScheduler	4	0	100/0
Scheduler1	22	5	82/18
Scheduler2	10	2	84/16
Instanciation	20	10	66/34
Total	56	17	77/23

Par ailleurs, nos modèles nous permettent de raisonner sur les programmes SystemC. En construisant un modèle B événementiel représentant le programme SystemC étudié nous pouvons prouver des propriétés propres aux programmes étudiés.

Nous pouvons définir ainsi un lien plus formel entre deux programmes SystemC. En effet, nous pouvons vérifier que les modèles des programmes comparés respectent les mêmes propriétés invariantes. En comparant ainsi deux programmes SystemC nous pouvons établir un lien formel, sorte de "raffinement", de ceux-ci en garantissant que deux programmes préservent un certain nombre de propriétés invariantes communes.

Enfin, la définition précise d'une sémantique opérationnelle (sémantique de simulation) d'un programme SystemC permet de considérer l'éventualité d'une traduction de modèle B événementiel vers un ou des modules SystemC. Ayant un modèle formel du comportement de l'ordonnanceur, nous pouvons vérifier que la traduction conserve les propriétés du modèle source.

Chapitre 6

Modèles généraux

Le présent chapitre aborde une généralisation des principes de modélisation mis en oeuvre au sein du projet EQUAST dans le but de les rendre appliquables à d'autres études de cas. Nous présenterons essentiellement des modèles B événementiels abstraits décrivant une hiérarchie et étudierons cette classe de modèles ainsi que ses différents raffinements.

La formalisation d'une hiérarchie sous forme de modèles généraux peut permettre d'envisager leurs réutilisations pour des applications particulières. En effet, l'étude de cas EQUAST était un exemple bien spécifique d'application et il convient de s'en abstraire afin de raisonner plus généralement.

Par ailleurs, l'étude précise des différentes natures de hiérarchie susceptibles d'être rencontrées permet de mieux envisager une éventuelle traduction ou implantation de celles-ci. En effet, la définition d'une fonction de traduction, la plus générale possible, de modèles B vers du code SystemC, demande de s'abstraire de l'étude de cas et de considérer les caractéristiques essentielles des modèles développés.

6.1 Hiérarchie de dépendance

Nous allons considérer des applications constituées d'un ensemble de tâches à exécuter. Ces tâches doivent avoir une relative indépendance pour permettre de les considérer de manière autonome mais elles doivent cependant constituer un tout cohérent et imposer certaines contraintes d'évaluation (temps, ordre, etc...).

Nous pouvons représenter les dépendances entre tâches d'une même application, comme un graphe ayant pour noeud les différentes tâches de l'application. Les transitions de ce graphe expriment la notion de dépendance. Les relations de dépendances entre tâches sont dictées par le "bon sens" ou par des dépendances de données : la tâche B a besoin du résultat de la tâche A pour pouvoir s'exécuter. Cette définition de la dépendance nous permet de considérer une vaste classe d'applications : toutes les applications pouvant être décrite sous la forme d'un graphe de flot de données peuvent a priori être considéré par notre approche.

Selon la nature de l'application, les tâches peuvent être réalisées plusieurs fois. Dans le cas de notre étude, l'outil de mesure fonctionne sans interruption et doit analyser chaque paquet en transit dans le réseau. De ce fait, les différentes tâches sont "rejouées" sur chaque paquet. De manière plus générale, toutes les applications de surveillance ou de type démon fonctionnent en continue, répétant les mêmes tâches régulièrement. Implicitement, il existe dans ce type d'application un mécanisme signalant l'exécution d'une nouvelle passe. Notre modélisation devra également prendre en compte ce type de détails.

6.1.1 Un graphe valué

L'aspect purement statique d'un graphe ne permet pas d'exprimer la dynamique d'une application. En effet, lors de l'exécution d'une application, les tâches d'une application peuvent être dans différents états selon l'avancement de leur exécution. Nous proposons de représenter l'exécution de l'application par un graphe valué, les valuations des noeuds du graphe traduisant l'état des tâches correspondantes :

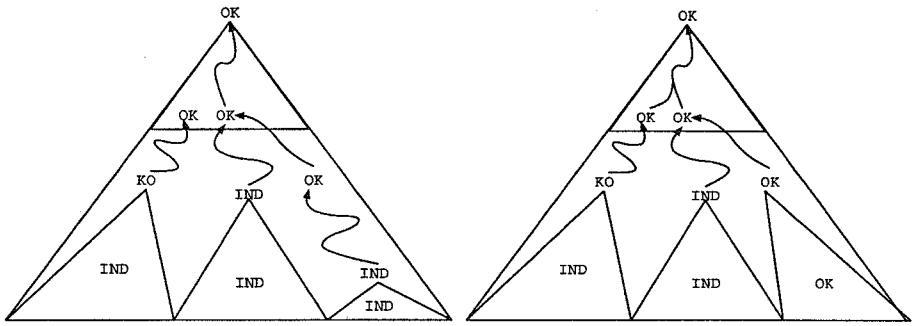
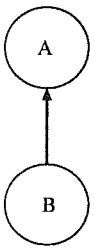


FIG. 6.1 – Etapes de parcours d'un graphe

- la tâche s'est exécutée et donne un résultat correct. Nous attribuerons la valeur OK à une tâche dans ce type d'état.
- la tâche est terminée et fournit un résultat incorrect. Nous attribuerons la valeur KO à une tâche dans cet état.
- la tâche n'est pas terminée et/ou n'est pas en mesure de fournir un résultat. Nous attribuerons la valeur IND à ce type de tâche.

Plusieurs stratégies peuvent être définies en fonction de l'état des tâches mais on peut considérer que lorsque le résultat fourni par une tâche n'est pas correct, les tâches dépendantes ne peuvent pas l'utiliser. Par ailleurs, une tâche ne peut commencer son exécution qu'une fois toutes les données reçues (notion de dépendance). La définition 3 suivante explicite la notion de dépendance et est une simple généralisation de la définition présentée dans le cadre du projet EQUAST s'appliquant aux paramètres d'évaluation[13].

Définition 3 (Dépendance de tâches) *Soit deux variables (noeuds) A et B représentant deux tâches distinctes T_A et T_B, on dit que la tâche T_B* **dépend** *de la tâche T_A (et l'on note $T_B \prec T_A$) si et seulement si l'un des prédicats suivants est présent dans l'invariant du modèle considéré :*
$B \neq IND \Rightarrow A = OK$ *ou*
$A \neq OK \Rightarrow B = IND$
Une notation graphique d'une telle relation est :



La figure 6.1 montre la progression de l'évaluation d'une application sur un graphe composé d'une racine et de n feuilles (arbre). La partie gauche de la figure représente l'état de l'application à un instant t et la partie droite représente l'état de la même application à l'instant $t+1$. Certaines tâches de l'application ont terminé leur exécution, d'autres n'ont pas encore commencé.

Nous allons par la suite décrire ce type de hiérarchie sous la forme de modèles B événementiels qui permettront de caractériser les principes énoncés ici de manière informelle et d'expliciter un peu plus les applications visées.

6.1.2 Propriétés de la relation de dépendance

La relation de dépendance que nous définissons entre les tâches d'une application a plusieurs propriétés intéressantes. En effet, des propriétés comme la transitivité, par exemple, permettent d'insérer des tâches

[13]Dans le cadre du projet EQUAST, les différents paramètres ont finalement été considérés comme des tâches spécifiques.

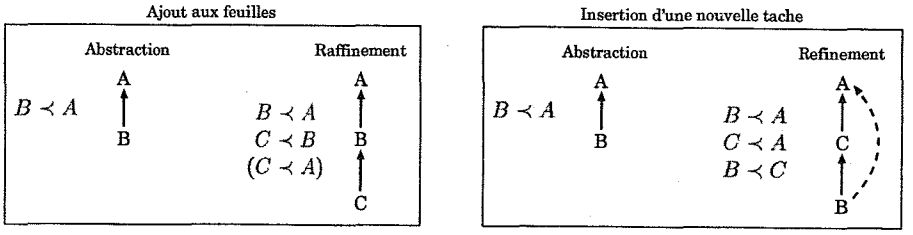


FIG. 6.2 – Hiérarchie incrémentale obtenue par raffinement

à l'intérieur de la hiérarchie au fur et à mesure des raffinements et pas seulement comme feuilles de l'arbre.

Transitivité de $\prec$

Nous pouvons montrer, à l'aide de modèles formels, que la relation de dépendance $\prec$ entre tâches est transitive. Nous avons prouvé facilement cette propriété de la relation : la preuve est entièrement réalisée automatiquement et nous ne détaillerons pas vraiment la construction de cette preuve. La transitivité de la relation de dépendance s'exprime par la propriété suivante :

$$T_C \prec T_B \wedge T_B \prec T_A \Rightarrow T_C \prec T_A$$

Raffinement et relation de dépendance

La notion de raffinement est centrale à la méthode B événementielle et nous permet de décrire un système de manière incrémentale. De ce fait, la construction de la hiérarchie de dépendances d'une application est également incrémentale. La question de la conservation de la cohérence de celle-ci au fur et à mesure des raffinements successifs peut se poser.

Une solution permettant l'ajout de nouvelles tâches et de nouvelles dépendances avec le raffinement est l'introduction de ces tâches comme feuilles de la hiérarchie incrémentale (voir partie gauche de la figure 6.2). L'ajout de feuilles à la hiérarchie ne contrarie pas les relations de dépendance déjà définies et consiste en un ajout de propriétés de dépendance entre la nouvelle feuille et l'ancienne feuille.

Avec la preuve de la transitivité de la relation de dépendance, l'insertion de nouvelles tâches intermédiaires dans la hiérarchie conserve la cohérence de celle-ci. En effet, comme le montre la partie droite de la figure 6.2, la transitivité de la relation de raffinement permet de préserver l'invariant abstrait c'est à dire de conserver les dépendances existant entre paramètres.

Si la relation de dépendance n'était pas transitive, les nouvelles tâches devraient toujours être ajoutées (au fur et à mesure des raffinements successifs) comme racines ou feuilles de la hiérarchie en construction. La plupart du temps il est difficile de prévoir à l'avance l'ordonnancement des tâches et c'est justement la hiérarchisation de celles-ci qui doit permettre de renseigner le concepteur sur un éventuel ordre d'exécution. Dans notre étude de cas, nous avons rencontré l'exemple d'un paramètre qui était décrit comme une vérification optionnelle et qui s'est révélé être d'une importance majeure et situé en plein coeur de la hiérarchie [14].

Détection de cycle

Supposons l'existence d'un cycle dans la hiérarchie représentant les tâches d'une application. Quelles déductions pouvons nous faire sur l'application en question et sur l'ordonnancement des tâches ? Nous allons étudier les conséquences de l'existence d'un cycle sur l'exemple suivant :

[14] Ce paramètre est le paramètre *Transport_error* qui indique une erreur des codes correcteurs.

INVARIANT
$(B \neq IND \Rightarrow A = OK) \ \wedge \ /* T_B \prec T_A */$
$(C \neq IND \Rightarrow B = OK) \ \wedge \ /* T_C \prec T_B */$
$(B \neq IND \Rightarrow C = OK) \ \wedge \ /* T_B \prec T_C */$
...

L'invariant précédent présente les dépendances existantes entre trois tâches T_A, T_B et T_C. Les deux tâches T_B et T_C forment un cycle de dépendance, celles-ci étant mutuellement dépendantes. L'existence d'un tel cycle dans l'invariant implique des conséquences sur les valuations possibles des valeurs des variables. Le tableau ci-dessous résume les différentes valuations possibles des variables B et C et le respect des relations de dépendances définies par l'invariant du modèle.

B	C	$T_C \prec T_B$	$T_B \prec T_C$
IND	**IND**	**vrai**	**vrai**
IND	OK	faux	vrai
IND	KO	faux	vrai
OK	IND	vrai	faux
OK	**OK**	**vrai**	**vrai**
OK	KO	vrai	faux
KO	IND	vrai	faux
KO	OK	faux	vrai
KO	KO	faux	faux

Il apparaît clairement que seul deux valeurs du couple de variables B et C sont cohérentes avec les relations de dépendance définies. Plus précisément les deux variables B et C doivent avoir la même valeur, toujours différente de KO.

Ceci signifie que les deux tâches T_B et T_C ne produisent jamais d'erreurs et sont toujours calculées exactement en même temps. La présence de cycles dans une hiérarchie de tâches est donc un signe assez clair d'un problème de définition de l'application. En effet, la transitivité de la relation permet, en cas de cycle, de déduire la relation $T_B \prec T_B$ qui est incohérente par définition.

La détection d'un cycle dans une hiérarchie peut assez facilement se faire en vérifiant que toutes les variables représentant des tâches peuvent prendre la valeur KO. Si l'on arrive à prouver qu'une tâche ne peut jamais provoquer une erreur (valeur à KO) cela signifie que la tâche en question est impliqué dans un cycle. Ainsi dans le cas de notre exemple, il est trivial de prouver que la variable B ne peut jamais prendre la valeur KO :

INVARIANT
$(B \neq IND \Rightarrow A = OK) \ \wedge \ /* T_B \prec T_A */$
$(C \neq IND \Rightarrow B = OK) \ \wedge \ /* T_C \prec T_B */$
$(B \neq IND \Rightarrow C = OK) \ \wedge \ /* T_B \prec T_C */$
$B \neq KO$
ASSERTIONS
$B = C$

Si les événements du modèle préservent les propriétés invariantes de dépendances définies, les obligations de preuves générées par le modèle permettront de prouver que la variable B ne prends jamais la valeur KO.

6.2 Modèles B de hiérarchie de dépendance

Les deux modèles suivant décrivent la forme générale d'une hiérarchie ainsi que la progression de l'évaluation des noeuds représentant les tâches de l'application. Nous nous concentrons ici sur la caractérisation des hiérarchies c'est pourquoi les modèles suivant ne présentent pas de propriétés invariantes de

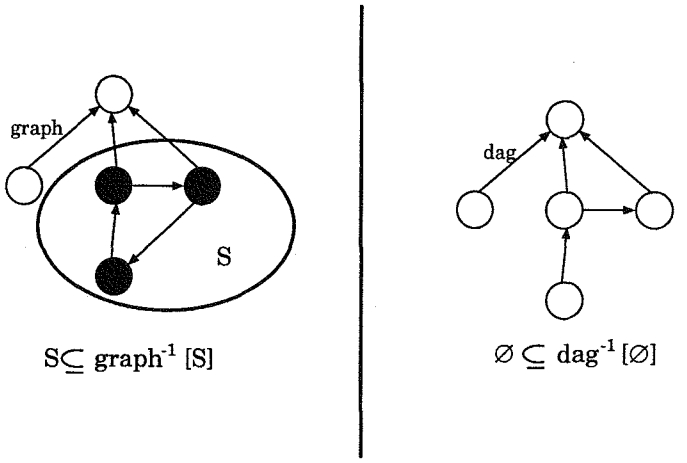


FIG. 6.3 – Illustration de l'acyclicité d'une relation *dag*

dépendance telles qu'elles sont présentées dans la définition 3. Le premier modèle abstrait a pour but l'explicitation des principales caractéristiques de la hiérarchie et du mode d'évaluation des différentes tâches la constituant. La motivation du second modèle, raffinement du premier, est de lever l'indéterminisme présent dans l'abstraction initiale. En effet, la réinitialisation d'une hiérarchie n'est pas entièrement explicitée dans le premier modèle et la réalisation d'un raffinement permet, tout en conservant les comportements de l'abstraction, de présenter une solution déterministe.

6.2.1 Modèle abstrait

Notre premier modèle se contente de décrire de manière très générale une hiérarchie de tâches. On va donc décrire un graphe acyclique *dag*, en se munissant d'un ensemble de sommet $VERTEX$. La structure du graphe est décrite par la relation *dag* qui est la relation de paternité sur le graphe. Si deux noeuds x et y sont tels que $y \in dag[\{x\}]$ alors la tâche représentée par le noeud x dépend de la tâche représentée par le noeud y.

Comme nous définissons un DAG, cette relation *dag* a certaines propriétés qui sont décrites dans le code B suivant. En premier lieu, un DAG est un graphe acyclique et la propriété $\forall S.(S \subseteq VERTEX \wedge S \subseteq dag^{-1}[S] \Rightarrow S = \emptyset)$ de la relation *dag* impose le respect de cette contrainte : à part l'ensemble vide, il n'existe pas d'ensemble S étant inclus dans leur image inverse par *dag*. Il n'existe pas d'ensemble de paramètres S contenant des tâches réalisant un cycle de dépendance. La figure 6.3 illustre cette définition.

MODEL
Hierarchie
SETS
$VALUE = \{OK, KO, IND\}$;
$VERTEX$
CONSTANTS
$dag, root$
PROPERTIES
$root \in VERTEX \wedge$
$dag \in VERTEX - \{root\} \leftrightarrow VERTEX \wedge$
$\mathbf{dom}(dag) = VERTEX - \{root\} \wedge$
$\forall S.(S \subseteq VERTEX \wedge S \subseteq dag^{-1}[S] \Rightarrow S = \emptyset)$

Les valuations du graphe sont décrites à l'aide des trois ensembles OK, KO et IND qui réalisent une partition des noeuds du graphe. Ces trois ensembles ont une signification tout à fait similaire à celle

donnée précédemment sur les tâches :

- tout noeud appartenant à l'ensemble OK est un noeud évalué et correct.
- tout noeud appartenant à l'ensemble KO est un noeud évalué et incorrect.
- tout noeud appartenant à l'ensemble IND est un noeud non évalué.

Initialement tous les noeuds sont non-évalués, les sous-ensembles OK et KO sont donc vides. Nous définissons également une fonction *predi* chargée de représenter, pour chaque tâche associée aux noeuds, des vérifications permettant de conclure sur le résultat de l'exécution.

VARIABLES
$IND, OK, KO, predi$
INVARIANT
$IND \subseteq VERTEX \wedge$
$OK \subseteq VERTEX \wedge$
$KO \subseteq VERTEX \wedge$
$OK \cap KO = \emptyset \wedge$
$OK \cap IND = \emptyset \wedge$
$KO \cap IND = \emptyset \wedge$
$OK \cup KO \cup IND = VERTEX \wedge$
$predi \in VERTEX \rightarrow \mathbf{BOOL}$
INITIALISATION
$OK := \emptyset \parallel$
$KO := \emptyset \parallel$
$IND := VERTEX \parallel$
$predi :\in VERTEX \rightarrow \mathbf{BOOL}$

Le couple d'événements suivant permet de modéliser la progression de l'évaluation au sein du graphe. Un noeud non-évalué ($n \in IND$) peut être évalué si tous ses parents directs ont déjà été évalués et sont corrects. De ce fait, la relation de dépendance modélisée par le graphe *dag* est bien respectée. En fonction du prédicat d'évaluation associé au noeud considéré, le noeud en question est ajouté à l'un des ensembles OK ou KO et retiré de l'ensemble IND.

progressOK $\widehat{=}$
ANY
n
WHERE
$n \in VERTEX \wedge$
$n \in IND \wedge$
$\forall p.(p \in VERTEX \wedge n, p \in dag$
$\Rightarrow p \in OK) \wedge$
$predi(n) = \mathbf{TRUE}$
THEN
$OK := OK \cup \{n\} \parallel$
$IND := IND - \{n\}$
END

progressKO $\widehat{=}$
ANY
n
WHERE
$n \in VERTEX \wedge$
$n \in IND \wedge$
$\forall p.(p \in VERTEX \wedge n, p \in dag$
$\Rightarrow p \in OK) \wedge$
$predi(n) = \mathbf{FALSE}$
THEN
$KO := KO \cup \{n\} \parallel$
$IND := IND - \{n\}$
END

Nous décrivons ainsi très simplement la hiérarchie de tâches d'une application, la relation *dag* ainsi que l'évaluation de cette hiérarchie, c'est-à-dire l'exécution de l'application, par le déclenchement du couple d'événements progressOK et progressKO. Les noeuds sont évalués par l'activation d'un de ces deux événements. Il reste encore un événement non présenté dans notre modèle. L'événement reset représente la réinitialisation de la hiérarchie à la fin d'une passe. La réinitialisation doit se déclencher lorsque l'ensemble des noeuds a été correctement évalué ou bien lorsqu'une erreur s'est produite. Cette version de l'événement reset est clairement indéterministe.

```
reset ≙
  SELECT
    OK = VERTEX∨
    KO ≠ ∅
  THEN
    IND := VERTEX ||
    OK := ∅ ||
    KO := ∅ ||
    predi :∈ VERTEX → BOOL ||
  END
```

En effet, l'événement reset présenté n'explicite pas le comportement du système lorsqu'une ou des erreurs se sont produites : les tâches pouvant être évaluées (car ne dépendant des résultats erronés) doivent elles être évaluées ou, au contraire, la réinitialisation doit-elle être immédiate ?

Nous nous proposons de raffiner ce modèle abstrait afin d'obtenir un modèle plus concret utilisant 4 sous-ensembles de noeuds pour lever l'indéterminisme de l'événement reset. La quatrième valeur doit représenter la propagation d'une erreur jusqu'aux feuilles afin de permettre une réinitialisation en ne considérant que celles-ci et en réalisant les tâches pouvant l'être. Ce choix est arbitraire mais semble relativement judicieux car il permet de conserver un état cohérent dans la hiérarchie.

6.2.2 Implantation à l'aide du raffinement

On ajoute dans ce raffinement, les 4 nouveaux ensembles formant une partition de l'ensemble des noeuds du graphe. Les deux ensembles concrets *ok* et *ko* sont absolument identiques aux ensembles abstraits OK et KO (cf. invariant de collage). Par contre, les ensembles *oko* et *ind* réalisent une partition de l'ancien ensemble abstrait IND. Comme annoncé précédemment, ce raffinement permet l'implantation de la hiérarchie en levant l'indéterminisme de l'événement reset. En effet, l'introduction d'une quatrième valeur va permettre de continuer l'évaluation des tâches jusqu'aux feuilles de la hiérarchie. De cette manière, l'ensemble de la hiérarchie est toujours évalué et le problème de l'instant de réinitialisation ne se pose plus : il suffit d'attendre la fin de l'évaluation complète de la hiérarchie.

L'exécution d'une tâche peut donc consister à propager une erreur qui a eu lieu durant l'exécution d'une tâche précédente. Il suffit de se munir d'un codage adéquat permettant de bien implanter les 4 valeurs *ind*, *ok*, *ko* et *oko*.

La dernière propriété de l'invariant est une propriété intéressante et nécessaire pour réaliser la preuve de raffinement : si l'ensemble *oko* n'est pas vide alors l'ensemble *ko* non plus. Autrement dit, on ne peut propager les erreurs que si une erreur s'est déjà produite. Comme dans le modèle abstrait, lors de l'initialisation, tous les ensembles sont vides sauf l'ensemble *ind*.

REFINEMENT
Hierarchie4V
REFINES
Hierarchie
VARIABLES
predi, *ok*, *ko*, *oko*, *ind*
INVARIANT
$ok \subseteq VERTEX \wedge$
$ok = OK \wedge$
$ko \subseteq VERTEX \wedge$
$ko = KO \wedge$
$ind \subseteq VERTEX \wedge$
$oko \subseteq VERTEX \wedge$
$ind \cup oko = IND \wedge$
$ind \cap oko = \emptyset \wedge$
$(oko \neq \emptyset \Rightarrow ko \neq \emptyset)$
INITIALISATION
$ok := \emptyset \parallel$
$ko := \emptyset \parallel$
$ind := VERTEX \parallel$
$oko := \emptyset \parallel$
$predi :\in VERTEX \rightarrow \mathbf{BOOL}$

Les versions raffinées des événements progressOK et progressKO sont identiques aux précédentes aux noms près. En effet, les ensembles abstraits OK et KO sont raffinés par les ensembles concrets ok et ko qui ont exactement le même rôle et dont les éléments n'ont pas plus de contraintes, l'égalité des ensembles étant spécifiée dans l'invariant de collage du modèle.

progressOK $\widehat{=}$
ANY
n
WHERE
$n \in VERTEX \wedge$
$n \in ind \wedge$
$\forall p.(p \in VERTEX \wedge n, p \in dag$
$\Rightarrow p \in ok) \wedge$
$predi(n) = \mathbf{TRUE}$
THEN
$ok := ok \cup \{n\} \parallel$
$ind := ind - \{n\}$
END

progressKO $\widehat{=}$
ANY
n
WHERE
$n \in VERTEX \wedge$
$n \in ind \wedge$
$\forall p.(p \in VERTEX \wedge n, p \in dag$
$\Rightarrow p \in ok) \wedge$
$predi(n) = \mathbf{FALSE}$
THEN
$ko := ko \cup \{n\} \parallel$
$ind := ind - \{n\}$
END

Le nouvel événement progressOKO permet la propagation des erreurs jusqu'aux feuilles. Comme pour les autres événements permettant la progression de l'évaluation, il faut attendre que tous les nœuds parents aient été évalués. La nouvelle valeur *oko* est également une évaluation possible et seule la valeur *ind* indique une non-évaluation d'un noeud (c'est-à-dire de la tâche lui correspondant). Ainsi, la garde de l'événement progressOKO est plus souple que les gardes du couple d'événements progressOK et progressKO. Les noeuds parents doivent être tous évalués pour permettre l'évaluation d'un nouveau noeud ($p \notin ind$) alors que dans le cas d'une progression "traditionnelle" les noeuds parents doivent être *correctement* évalués. Il doit cependant exister au moins un noeud m, dans les parents du noeud courant n, qui est en état d'erreur ou qui propage déjà une erreur.

L'ajout de l'événement progressOKO et de l'ensemble *oko* permet de rendre l'événement reset déterministe. En effet, dans ce modèle encore abstrait, la réinitialisation a désormais uniquement lieu lorsque

tous les noeuds ont été évalués ($ind = \emptyset$).

progressOKO $\widehat{=}$
ANY
n, m
WHERE
$n \in VERTEX \wedge$
$m \in VERTEX \wedge$
$n \in ind \wedge$
$n, m \in dag \wedge$
$\forall p.(p \in VERTEX \wedge n, p \in dag \Rightarrow p \notin ind) \wedge$
$m \in ko \cup oko$
THEN
$oko := oko \cup \{n\} \parallel$
$ind := ind - \{n\}$
END

reset $\widehat{=}$
SELECT
$ind = \emptyset$
THEN
$ok := \emptyset \parallel$
$ko := \emptyset \parallel$
$ind := VERTEX \parallel$
$oko := \emptyset \parallel$
$predi :\in VERTEX \rightarrow \mathbf{BOOL}$
END

Ce raffinement permet de décrire une hiérarchie de dépendances sous la forme d'un graphe acyclique, chaque noeud du graphe représentant une tâche du système modélisé. Les différentes valuations du graphe représentent l'état du système et l'évaluation des tâches le composant. Les événements du modèle simulent l'exécution des tâches en imposant des contraintes d'ordonnancement : une tâche ne peut s'exécuter que si l'ensemble de ses parents s'est exécuté. Le parallélisme d'exécution des tâches est implicite dans notre modèle et l'indéterminisme lié à la réinitialisation a été levé. A partir de ce modèle, nous pouvons exprimer une implantation (par raffinements successifs) et montrer qu'elle conserve les propriétés de la hiérarchie formelle.

6.3 Hiérarchie : raffinement vers le matériel

Nous nous proposons de montrer ici comment implanter, en SystemC, une hiérarchie formelle. Pour cela nous allons utiliser les modèles généraux présentés dans la section précédente et poursuivre le raffinement en introduisant avec les raffinements successifs les constructions et les structures SystemC. Nous avons proposé des modèles formels génériques de l'exécution d'un programme SystemC, nous sommes donc en mesure de décrire clairement l'exécution d'un programme. La preuve de raffinement nous garantira la préservation des propriétés abstraites de dépendance.

Le développement complet de cette implantation représente 6 modèles (1 modèle abstrait et 5 raffinements) ainsi qu'un ensemble de 178 obligations de preuves. Le modèle abstrait initial (*Hierarchie*) est le modèle présenté dans la section précédente et non le raffinement *Hierarchie4V*. En effet, le raffinement *Hierarchie4V* impose une méthode de réinitialisation (propagation des erreurs et attente de l'évaluation des feuilles) dont nous avons voulu nous abstraire. Ce choix de réinitialisation n'est cependant pas un problème et ne contredit pas les développements que nous avons menés. Cependant, nous préférons dans ce développement nous focaliser sur la transformation incrémentale d'une hiérarchie en un ensemble de processus qui conserve l'ordonnancement initialement défini.

Afin de se rapprocher d'une description SystemC, nous devons introduire des notions nouvelles comme les processus et autres canaux de communication puis "distribuer" les traitements aux différents processus. Le premier raffinement (*Wave*) réalise la distribution de la réinitialisation en explicitant deux phases d'évaluation distinctes et en introduisant un nouvel événement réalisant la réinitialisation "locale" des tâches.

Le second raffinement (*Processus*) introduit la notion de processus et lie les tâches de la hiérarchie aux processus les réalisant. Le but de ce raffinement est de se rapprocher d'une description SystemC et du modèle de calcul utilisé par celle-ci.

Le troisième raffinement (*Channels*) a également pour motivations l'introduction progressive des éléments constitutifs d'un programme SystemC. Ce modèle introduit donc les canaux de communication

permettant la communication et le codage de l'état de chaque processus (càd de chaque tâche) à ses descendants.

Le raffinement suivant (*toArchi*) supprime la vue globale du système représentée par l'utilisation de la constante *dag*. Ce raffinement introduit une relation *input* qui permet de représenter les entrées des différents processus : l'activation d'un processus se fait par la consultation de valeurs locales.

Enfin le dernier raffinememnt (*Architecture*) est un raffinement de convenance qui permet de se rapprocher de la modélisation B réalisée sur l'ordonnanceur SystemC. La principale modification de ce raffinement est le changement de représentation des valeurs des signaux.

6.3.1 Modèle abstrait et premier raffinement

Notre abstraction de départ est exactement le modèle *Hierarchie* présenté dans la section précédente. Ce modèle décrit de manière générale une hiérarchie de dépendance et son animation. Dans le cadre de cette étude, le premier raffinement est un peu différent du modèle $Hierarchie4V$. Nous avons décidé de considérer que le système avait deux phases distinctes :

- phase d'évaluation : les tâches s'exécutent en utilisant des données et sont contraintes par la relation de dépendance.
- phase de réinitialisation : la hiérarchie n'est pas réinitialisée globalement de manière "magique", chaque tâche est chargée de se repositionner dans un état permettant le traitement des nouvelles données. La relation de dépendance n'a pas de raison d'être dans cette phase et toutes les tâches peuvent travailler parallèlement.

Nous définissons donc le raffinement $Wave$ de la manière suivante :

REFINEMENT
$Wave$
REFINES
$Hierarchie$
SETS
$PHASE = \{eval, re\}$
VARIABLES
$predi, rOK, rKO, rIND, phase$
INVARIANT
$rOK \subseteq VERTEX \wedge$
$rKO \subseteq VERTEX \wedge$
$rIND \subseteq VERTEX \wedge$
$rIND \cup rOK \cup rKO = VERTEX \wedge$
$rIND \cap rKO = \emptyset \wedge rIND \cap rOK = \emptyset \wedge$
$rKO \cap rOK = \emptyset \wedge$
$phase \in PHASE \wedge$
$rOK \subseteq OK \wedge rKO \subseteq KO \wedge$
$(phase \neq re \Rightarrow rIND \subseteq IND) \wedge$
$(phase = re \Rightarrow (OK = VERTEX \vee KO \neq \emptyset))$
INITIALISATION
$rIND := VERTEX \parallel$
$rOK := \emptyset \parallel$
$rKO := \emptyset \parallel$
$phase := eval \parallel$
$predi :\in VERTEX \rightarrow BOOL$

L'ensemble $PHASE$ ainsi que la variable *phase* permettent de modéliser les deux phases distinctes du comportement du système. Les variables concrètes $rIND$, rOK et rKO représentent les valeurs concrètes des noeuds de la hiérarchie. Comme la réinitialisation du système est faite localement par chaque tâche, les variables concrètes sont incluses dans les variables abstraites (voir l'invariant de collage). En phase d'évaluation, les variables concrètes contiennent exactement les mêmes noeuds que les variables abstraites.

Evénements raffinés

Le couple d'événements progressOK et progressKO est raffiné et manipule les variables concrètes. Les gardes des événements sont renforcées afin de spécifier que ceux-ci n'ont lieu qu'en phase d'évaluation.

progressOK $\widehat{=}$
ANY
v
WHERE
$phase = eval \wedge$
$v \in rIND \wedge$
$predi(v) = TRUE \wedge$
$\forall y.(y \in VERTEX \wedge v, y \in dag \Rightarrow y \in rOK)$
THEN
$rOK := rOK \cup \{v\} \parallel$
$rIND := rIND - \{v\}$
END

progressKO $\widehat{=}$
ANY
v
WHERE
$phase = eval \wedge$
$v \in rIND \wedge$
$predi(v) = FALSE \wedge$
$\forall y.(y \in VERTEX \wedge v, y \in dag \Rightarrow y \in rOK)$
THEN
$rKO := rKO \cup \{v\} \parallel$
$rIND := rIND - \{v\}$
END

La version raffinée de l'événement reset est différente de l'abstraction. L'événement abstrait réinitialise l'ensemble des noeuds de la hiérarchie alors que dans cette version concrète ce travail a déjà été fait $rIND = VERTEX$ (localement pour chaque noeud). L'événement concret reset se contente donc de signaler que le système est à nouveau prêt à traiter des données ($phase := eval$) et symbolise l'arrivée d'une nouvelle donnée en réaffectant la fonction *predi*.

reset $\widehat{=}$
SELECT
$phase = re \wedge$
$rIND = VERTEX$
THEN
$phase := eval \parallel$
$predi :\in VERTEX \rightarrow BOOL$
END

Nouveaux événements

De nouveaux événements apparaissent dans ce raffinement. Un premier événement toReset explicite le passage de la phase d'évaluation à la phase de réinitialisation. Cet événement ne manipule aucune variable de l'abstraction mais reprend une partie de la garde de l'événement abstrait reset avec les variables concrètes.

Un second événement progressIND modélise la réinitialisation locale des tâches. L'activation de cet événement provoque la réinitialisation d'un noeud v déjà évalué. A nouveau, cet événement ne manipule aucune variable de l'abstraction préservant ainsi l'invariant et permettant la preuve de raffinement.

```
toReset ≙
  SELECT
    phase = eval ∧
    (rOK = VERTEX ∨
    rKO ≠ ∅)
  THEN
    phase := re
  END
```

```
progressIND ≙
  ANY
    v
  WHERE
    v ∈ rOK ∪ rKO ∧
    phase = re
  THEN
    rOK := rOK − {v} ||
    rKO := rKO − {v} ||
    rIND := rIND ∪ {v}
  END
```

Ce premier raffinement nous a permis de passer d'une réinitialisation abstraite à une réinitialisation implantable où chaque tâche a la charge de se préparer. Le système concret se rapproche déjà d'une implantation matérielle réaliste. En effet, dans un circuit, la réinitialisation des canaux de communication se fait simplement : les objets (modules, portes logiques etc...) émettant sur le média de communication cessent d'émettre provoquant ainsi une réinitialisation.

6.3.2 Second raffinement : introduction de processus

Nous proposons d'implanter notre hiérarchie sous la forme d'un ensemble de processus concurrents dont l'ordonnancement respecte les contraintes définies par la relation de dépendance de tâches. De ce fait, nous allons devoir introduire un certain nombre de structures permettant l'implantation des différents éléments abstraits (tâches, relation de dépendance). Les raffinements suivants vont permettre d'expliciter les choix d'implantation.

La première étape de l'implantation consiste à lier les tâches de l'application à leur implantation. Nous avons considéré (après une démarche similaire dans le projet EQUAST) qu'un bon choix était d'implanter chaque tâche comme un processus indépendant. Ce premier raffinement est donc assez simple et est chargé d'introduire les processus implantant les tâches (éléments de l'ensemble abstrait $VERTEX$). L'en-tête du modèle est présenté ci-dessous :

REFINEMENT
Processus
REFINES
Wave
SETS
$PROCESSUS$
CONSTANTS
$TRAD, RESET$
PROPERTIES
$RESET \in PROCESSUS \wedge$
$TRAD \in VERTEX \rightarrowtail\!\!\!\!\!\!\rightarrow PROCESSUS - \{RESET\}$
VARIABLES
$predi, phase,$
$pOK, pKO, pIND, predicat$
INVARIANT
$pOK \subseteq PROCESSUS - \{RESET\} \wedge$
$pKO \subseteq PROCESSUS - \{RESET\} \wedge$
$pIND \subseteq PROCESSUS - \{RESET\} \wedge$
$pOK = TRAD[rOK] \wedge pKO = TRAD[rKO] \wedge$
$pIND = TRAD[rIND] \wedge$
$predicat \in PROCESSUS - \{RESET\} \rightarrow BOOL \wedge$
$\forall m.(m \in PROCESSUS \Rightarrow predicat(m) = predi(TRAD^{-1}(m)))$

L'ensemble des processus de l'implantation $PROCESSUS$ est introduit. Cet ensemble contient un processus particulier $RESET$. Une *fonction de traduction* $TRAD$ est définie : elle modélise l'implantation de chaque tâche de l'application en un processus unique, de ce fait $TRAD$ est une bijection.

Afin de conserver l'état des tâches, l'ensemble $PROCESSUS$ est partitionné en trois sous-ensembles pOK, pKO et $pIND$. Ces trois ensembles sont liés aux ensembles abstraits rOK, rKO et $rIND$ par l'intermédiaire de la bijection $TRAD$. L'invariant de collage explicite la signification de ces trois ensembles.

Evénements raffinés

Les événements du modèle sont raffinés afin de manipuler les processus implantant les tâches. En effet, les événements progressOK et progressKO manipulent désormais un processus n'ayant pas encore produit de résultat $p \in pIND$ et dont les processus parents (c'est-à-dire les processus implantant les tâches parentes) ont tous terminé correctement leurs évaluations $\forall y.(y \in PROCESSUS - \{RESET\} \wedge TRAD^{-1}(p), TRAD^{-1}(y) \in dag \Rightarrow y \in pOK)$.

progressOK $\widehat{=}$
ANY
p
WHERE
$phase = eval \wedge$
$p \in pIND \wedge$
$predicat(p) = TRUE \wedge$
$\forall y.(y \in PROCESSUS - \{RESET\} \wedge TRAD^{-1}(p), TRAD^{-1}(y) \in dag \Rightarrow y \in pOK)$
THEN
$pOK := pOK \cup \{p\} \parallel$
$pIND := pIND - \{p\}$
END

progressKO $\widehat{=}$
ANY
p
WHERE
$phase = eval \wedge$
$p \in pIND \wedge$
$predicat(p) = FALSE \wedge$
$\forall y.(y : PROCESSUS - \{RESET\} \wedge TRAD^{-1}(p), TRAD^{-1}(y) \in dag \Rightarrow y \in pOK)$
THEN
$pKO := pKO \cup \{p\} \parallel$
$pIND := pIND - \{p\}$
END

Le changement de phase du système est toujours modélisé par les deux événements reset et toReset. De la même façon que précédemment, ces deux événements concrets manipulent désormais les processus implantant les tâches de l'application.

reset $\widehat{=}$
SELECT
$phase = re \wedge$
$pIND = PROCESSUS - \{RESET\}$
THEN
$phase := eval \parallel$
$predi, predicat \in ($
$predi : VERTEX \rightarrow BOOL \wedge$
$predicat \in PROCESSUS - \{RESET\} \rightarrow BOOL \wedge$
$\forall m.(m \in PROCESSUS \Rightarrow predicat(m) = predi(TRAD^{-1}(m))))$
END

toReset $\widehat{=}$
SELECT
$phase = eval \wedge$
$(pOK = PROCESSUS - \{RESET\} \vee$
$pKO \neq \emptyset)$
THEN
$phase := re$
END

L'événement progressIND est raffiné de la même manière que les autres événements du modèle c'est pourquoi il n'est pas présenté ici.

Ce second raffinement permet de détailler quelque peu une implantation possible de hiérarchie de dépendance (et par conséquent de l'application qu'elle représente) en considérant que chaque tâche est réalisée par un processus. Les différents processus sont concurrents (en effet les différents événements du modèle permettent l'exécution de plusieurs branches de la hiérarchie en parallèle) mais sont contraints par la relation de dépendance. Le raffinement présenté n'est pas encore une implantation de la hiérarchie qui reste utilisée par les gardes des événements pour contraindre leurs activations. Les prochains raffinements devront expliciter une implantation possible de la hiérarchie formelle.

6.3.3 Raffinements suivants : introduction de signaux

Les deux raffinements suivants introduisent les signaux de communications permettant l'implantation de la hiérarchie. Avec l'introduction de ces signaux, les ensembles servant à modéliser l'état des processus du système vont disparaître au profit de valeurs booléennes en transit sur les signaux.

Pour éviter une lecture trop fastidieuse nous allons présenter rapidement les principales caractéristiques des deux modèles.

Raffinement *Channels* : signaux

Nous introduisons la constante $CHANNELS$, l'ensemble des signaux utiles à l'implantation de la hiérarchie. Nous le partitionnons en deux sous-ensembles $Cerr$ et $Cend$ qui représentent des signaux de fonctionnalités différentes. Ces ensembles représentent les signaux permettant aux processus de communiquer leur état :

- l'ensemble $Cend$ est le co-domaine de la bijection *ending* associant à tout processus implantant une tâche un signal indiquant la terminaison du processus.
- l'ensemble $Cerr$ est le co-domaine de la bijection *error* associant à tout processus implantant une tâche un signal indiquant une erreur du processus.

Un signal particulier est présent dans le système : la constante *Creset* représente ce signal qui n'appartient pas aux deux sous-ensembles *Cerr* et *Cend*. Les fonctions *ending* et *error* sont des bijections vers des sous-ensembles disjoints de signaux.

REFINEMENT
Channels
REFINES
Processus
SETS
$CHANNELS$
CONSTANTS
$Cend, Cerr, ending, error, Creset$
PROPERTIES
$Cend \subseteq CHANNELS \wedge$
$Cerr \subseteq CHANNELS \wedge$
$Cend \cap Cerr = \emptyset \wedge$
$Creset \in CHANNELS \wedge$
$Creset \notin (Cend \cup Cerr) \wedge$
$Cend \cup Cerr \cup \{Creset\} = CHANNELS \wedge$
$ending \in PROCESSUS - \{RESET\} \rightarrowtail\!\!\!\!\!\!\rightarrow Cend \wedge$
$error \in PROCESSUS - \{RESET\} \rightarrowtail\!\!\!\!\!\!\rightarrow Cerr \wedge$
$ending \cap error = \emptyset$

On introduit deux nouvelles variables dans ce raffinement : les ensembles $High$ et Low. Ils permettent de modéliser la valuation des éléments de l'ensemble $CHANNELS$, et en réalisent une partition. Ces deux ensembles représentent les signaux booléens ayant pour valeur *true* (présence d'un courant électrique) et *false* (absence de tout courant). Les variables abstraites du raffinement précédent ont, pour la plupart, disparu comme la variable *phase*. Les différentes phases de l'application sont désormais *codées* à l'aide du signal *Creset*. L'invariant de collage ci-dessous exprime clairement le codage utilisé. De même les valuations des signaux permettent de *coder* les ensembles abstraits $pIND$, pOK et pKO, l'invariant de collage permettant de prouver la correction du raffinement.

VARIABLES
$predi, predicat,$
$High, Low$
INVARIANT
$High \subseteq CHANNELS \wedge$
$Low \subseteq CHANNELS \wedge$
$High \cap Low = \emptyset \wedge$
$High \cup Low = CHANNELS \wedge$
$(Creset \in Low \Rightarrow phase = eval) \wedge$
$(Creset \in High \Rightarrow phase = re) \wedge$
$pIND = ending^{-1}[Low] \cap error^{-1}[Low] \wedge$
$pOK = ending^{-1}[High] \wedge pKO = error^{-1}[High]$
DEFINITIONS
$outputs \triangleq ending \cup error$

On a donc le codage suivant :
- $pIND = ending^{-1}[Low] \cap error^{-1}[Low]$, les signaux de terminaison et d'erreur des tâches non encore évaluées ne sont pas positionnés.
- $pOK = ending^{-1}[High]$, lorsqu'une tâche s'est correctement terminée, le signal de sortie *ending* du processus l'implantant est positionné à *true*.

- $pKO = error^{-1}[High]$, lorsqu'une tâche produit une erreur, le signal de sortie *error* du processus l'implantant est positionné à *true*.

Les événements sont désormais raffinés pour prendre en compte cette implantation de l'état des tâches. L'événement concret progressOK est présenté ci-dessous, à titre d'exemple, afin de montrer la "conversion" réalisée :

progressOK $\widehat{=}$
ANY
p
WHERE
$Creset \in Low \wedge$
$p \in PROCESSUS - \{RESET\} \wedge$
$outputs[\{p\}] \subseteq Low \wedge$
$predicat(p) = TRUE \wedge$
$\forall y.(y \in PROCESSUS - \{RESET\} \wedge TRAD^{-1}(p), TRAD^{-1}(y) \in dag \Rightarrow$
$ending(y) \in High)$
THEN
$High := High \cup \{ending(p)\} \parallel$
$Low := Low - \{ending(p)\}$
END

Raffinement *toArchi* : implanter la hiérarchie

Jusqu'au raffinement précédent, la hiérarchie n'était pas implantée et les processus parents d'un processus étaient trouvés par l'intermédiaire de la relation abstraite *dag*. Ce raffinement supprime cette constante et en introduit : *inputs* qui représente les signaux d'entrée des processus. La constante *inputs* est une implantation de la constante abstraite *dag*. Les propriétés suivantes expriment cette implantation en indiquant que les signaux de terminaison des processus pères sont des entrées des processus fils permettant ainsi l'implantation de la relation de dépendance.

CONSTANTS
$inputs$
PROPERTIES
$inputs \in PROCESSUS \leftrightarrow CHANNELS \wedge$
$(ending; inputs^{-1}) = (TRAD^{-1}; dag; TRAD) \wedge$
$\forall (x, y).(x \in VERTEX \wedge y \in VERTEX \wedge y \in dag[\{x\}] \Rightarrow$
$ending(TRAD(y)) \in inputs[\{TRAD(x)\}]) \wedge$
$inputs[\{RESET\}] = Cend \cup Cerr$

Une propriété importante exprimée ci-dessus est que lorsque l'on compose la hiérarchie *dag* avec la bijection $TRAD$ afin d'obtenir les processus parents des processus $((TRAD^{-1}; dag; TRAD))$, on a exactement les mêmes processus que lorsque l'on compose la relation *inputs* décrivant l'ensemble des signaux d'entrée des processus avec la fonction *ending* permettant d'associer un processus à son signal de terminaison.

Les événements du modèle sont raffinés afin de considérer l'implantation de la hiérarchie proposée. Une vision globale du système (par l'intermédiaire de la relation *dag*) est remplacée par la vision locale de chaque processus. En effet, les gardes des deux événements progressOK et progressKO ne manipulent que des informations locales en relation avec le processus p exécuté : ses entrées ($inputs[\{p\}]$) et ses sorties ($outputs[\{p\}]$). Grâce aux propriétés de la constante *inputs* il est facile de démontrer que ces événements raffinent bien les événements abstraits et par conséquent que l'implantation définie de manière abstraite ici respecte bien l'ordonnancement défini par la hiérarchie abstraite.

progressOK $\widehat{=}$
ANY
p
WHERE
$Creset \in Low \wedge$
$p \in PROCESSUS - \{RESET\} \wedge$
$outputs[\{p\}] \subseteq Low \wedge$
$predicat(p) = TRUE \wedge$
$\forall i.(i \in inputs[\{p\}] - \{Creset\} \Rightarrow i \in High)$
THEN
$High := High \cup \{ending(p)\} \parallel$
$Low := Low - \{ending(p)\}$
END

progressKO $\widehat{=}$
ANY
p WHERE
$Creset \in Low \wedge$
$p \in PROCESSUS - \{RESET\} \wedge$
$outputs[\{p\}] \subseteq Low \wedge$
$predicat(p) = FALSE \wedge$
$\forall i.(i \in inputs[\{p\}] - \{Creset\} \Rightarrow i \in High)$
THEN
$High := High \cup \{error(p)\} \parallel$
$Low := Low - \{error(p)\}$
END

L'événement concret reset simule l'exécution du processus particulier $RESET$ défini et permet d'indiquer aux différents processus que le système est à nouveau prêt. De la même manière que précédemment, les informations manipulées par cet événement sont d'ordre local et sont toutes rattachées au processus $RESET$.

reset $\widehat{=}$
SELECT
$Creset \in High \wedge$
$inputs[\{RESET\}] = Low$
THEN
$High := High - \{Creset\} \parallel$
$Low := Low \cup \{Creset\} \parallel$
$predi, predicat \in ($
$predi \in VERTEX \rightarrow BOOL \wedge$
$predicat \in PROCESSUS - \{RESET\} \rightarrow BOOL \wedge$
$\forall m.(m \in PROCESSUS \Rightarrow predicat(m) = predi(TRAD^{-1}(m)))$
$)$
END

Le présent raffinement explicite entièrement l'implantation de la relation de dépendance. Nous avons ainsi explicité une implantation possible de la hiérarchie formelle, liant par une relation de dépendance les tâches d'une application donnée. Pour des raisons techniques (difficulté des preuves), nous avons modélisé la valuation des signaux de communication à l'aide d'ensembles. Le raffinement suivant permet d'exprimer, dans un formalisme plus proche de celui employé dans le chapitre 5 (page 121), l'évaluation d'une hiérarchie.

6.3.4 Hiérarchie : implantation proposée

Le dernier modèle proposé est une description abstraite d'une architecture matérielle implantant la hiérarchie formelle. Les anciennes variables abstraites sont toutes supprimées (sauf les deux variables *predi* et *predicat* exprimant les conditions de terminaison des tâches) et une nouvelle variable $getValue$ apparaît. La fonction $getValue$ représente les valeurs des différents signaux du système et permet un ordonnancement respectant les contraintes abstraites.

Il est possible de trouver un invariant de collage reliant la fonction $getValue$ aux variables abstraites du modèle précédent. De même, les assertions définies dans l'en-tête du modèle sont aisément prouvables.

REFINEMENT
Architecture
REFINES
toArchi
SETS
$VALUE = \{high, low\}$
VARIABLES
predi, *predicat*,
$getValue$
INVARIANT
$getValue \in CHANNELS \rightarrow VALUE \wedge$
$getValue\ [\{low\}] = Low \wedge$
$getValue\ [\{high\}] = High$

DEFINITIONS
$outputs \triangleq (ending \cup error)$
ASSERTIONS
$\forall c.(c \in CHANNELS \wedge getValue(c) = high \Rightarrow c \in High);$
$\forall c.(c \in CHANNELS \wedge getValue(c) = low \Rightarrow c \in Low)$
INITIALISATION
$getValue := CHANNELS * \{low\} \parallel$
$predi, predicat \in ($
$predi \in VERTEX \rightarrow BOOL \wedge$
$predicat \in PROCESSUS - \{RESET\} \rightarrow BOOL \wedge$
...)

Les versions terminales des événements modélisent l'exécution des processus en ne manipulant que des informations locales aux processus. Il n'est plus fait référence aux variables abstraites, la fonction $getValue$ photographiant l'état du système à un instant donné.

progressOK $\widehat{=}$
ANY
p
WHERE
$getValue(Creset) = low \wedge$
$p \in PROCESSUS - \{RESET\} \wedge$
$getValue[outputs[\{p\}]] = \{low\} \wedge$
$predicat(p) = TRUE \wedge$
$\forall i.(i \in inputs[\{p\}] - \{Creset\} \Rightarrow getValue(i) = high)$
THEN
$getValue := getValue \ \lhd\!\!- \{ending(p) \mapsto high\}$
END

progressKO $\widehat{=}$
ANY
p
WHERE
$getValue(Creset) = low \wedge$
$p \in PROCESSUS - \{RESET\} \wedge$
$getValue[outputs[\{p\}]] = \{low\} \wedge$
$predicat(p) = FALSE \wedge$
$\forall i.(i \in inputs[\{p\}] - \{Creset\} \Rightarrow getValue(i) = high)$
THEN
$getValue := getValue \lhd\!\!- \{error(p) \mapsto high\}$
END

La réinitialisation de l'application est distribuée dans les différents processus qui sont chargés de réinitialiser leurs sorties quand ils en sont informés ($getValue(Creset) = high$).

progressIND $\widehat{=}$
ANY
p
WHERE
$p \in PROCESSUS - \{RESET\} \wedge$
$(getValue(error(p)) = high \vee$
$getValue(ending(p)) = high) \wedge$
$getValue(Creset) = high$
THEN
$getValue := getValue \lhd\!\!- \{ending(p) \mapsto low, error(p) \mapsto low\}$
END

Le retour en phase d'évaluation, une fois la réinitialisation terminée, est toujours représenté par l'événement reset qui modélise de manière abstraite l'exécution du processus $RESET$. De même, l'événement toReset modélise l'entrée en phase de réinitialisation par une exécution du processus $RESET$ une fois l'ensemble des processus exécutés. L'entrée en phase de réinitialisation se traduit par l'émission de la valeur *true* sur le signal *Creset*.

reset $\widehat{=}$
SELECT
$getValue(Creset) = high \wedge$
$getValue[inputs[\{RESET\}]] = \{low\}$
THEN
$getValue := getValue \lhd\!\!- \{Creset \mapsto low\} \parallel$
$predi, predicat \in ($
$predi \in VERTEX \rightarrow BOOL \wedge$
$predicat \in PROCESSUS - \{RESET\} \rightarrow BOOL \wedge$
$\forall m.(m \in PROCESSUS \Rightarrow predicat(m) = predi(TRAD^{-1}(m)))$
$)$
END

toReset $\widehat{=}$
SELECT
$getValue(Creset) = low \wedge$
$(getValue[Cend] = \{high\} \vee$
$getValue[Cerr] \cap \{high\} \neq \emptyset)$
THEN
$getValue := getValue \lhd\!\!- \{Creset \mapsto high\}$
END

Ce raffinement décrit de manière simple une implantation (abstraite) d'une hiérarchie de dépendance. Ce dernier modèle est le résultat d'une série de développements introduisant progressivement des éléments implantatoires propres à l'électronique tout en conservant, par la relation de raffinement, les comportements et les spécificités de l'abstraction.

Ayant travaillé sur des modèles généraux de hiérarchie, les modèles représentant l'implantation restent assez généraux mais montrent clairement comment dériver une implantation imposant un ordonnancement des processus respectant la hiérarchie définie sur les tâches. Ce développement a encore une fois des vertus pédagogiques et permet une bonne compréhension des mécanismes et du codage permettant la synchronisation des processus.

6.4 Abstraction : conclusions

Les considérations générales sur la relation de dépendance entre tâches nous ont amené à considérer de manière détaillée la notion de hiérarchie. Nous avons ainsi montré qu'une hiérarchie de tâches était basée sur une relation transitive de dépendance et qu'une telle hiérarchie était un graphe acyclique. L'existence d'un cycle dans le graphe de dépendance est la preuve d'une spécification incohérente car entraînant une impossibilité d'évaluer les tâches impliquées dans le cycle.

Nous avons, par la suite, formalisé ces considérations générales en des modèles B événementiels permettant de raisonner et de prouver les propriétés supposées. Ces modèles B abstraits nous ont servis de support pour dériver par raffinements successifs un modèle d'implantation d'une hiérarchie. Ce travail nous a convaincu de la possibilité de définir des règles de traduction permettant de produire, à partir d'un modèle B événementiel décrivant la hiérarchie de tâches d'une application, une architecture électronique décrite en SystemC. La figure 6.4 résume la démarche générale que nous avons suivie. Pour une application particulière, la description de la hiérarchie est réalisée de manière incrémentale (comme pour le projet EQUAST) depuis la spécification initiale. Les modèles généraux présentés précédemment dressent une trame de raffinements permettant l'implantation du système.

En effet, nos précédents travaux sur SystemC nous ont permis d'acquérir une bonne maîtrise de ce langage et l'abstraction décrivant l'implantation d'une hiérarchie manipule des "composants" immédiatement disponibles avec SystemC (signaux, processus, ...). La définition d'une fonction de traduction est présentée dans le chapitre suivant, s'accompagnant également de modèles expliquant le lien entre le code SystemC produit et la hiérarchie formelle définie.

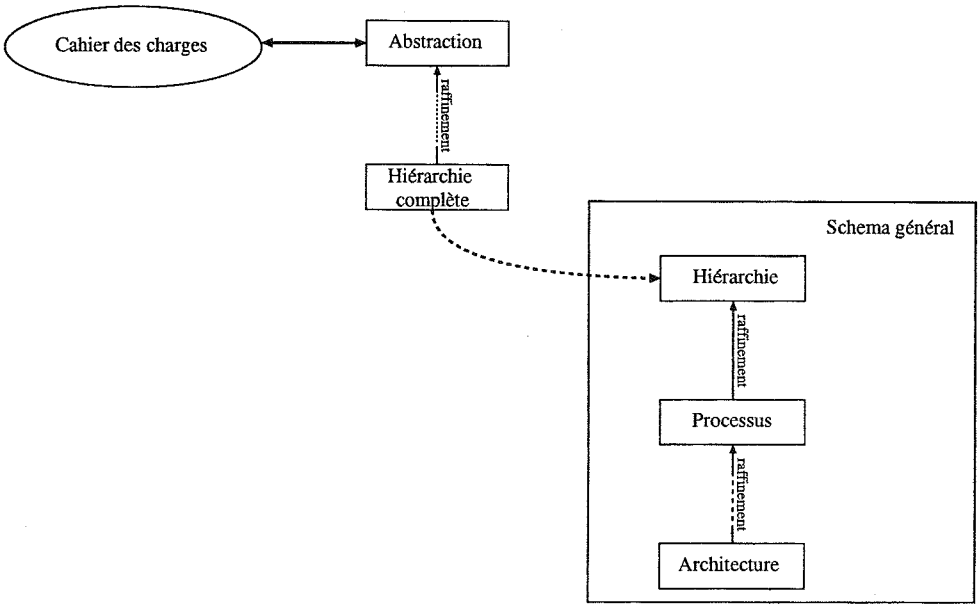


FIG. 6.4 – Illustration du raffinement de hiérarchie

Chapitre 7

Conception de SoC à partir de modèles B

La méthode B événementielle nous a permis de comprendre et de formaliser les problèmes mais n'est pas une solution exploitable directement en électronique. En effet, un certain nombre de problèmes ne peuvent être considérés facilement dans des modèles abstraits et dépendent de la technologie d'implantation : temps de communication et de propagation des signaux, surface de silicone, consommation électrique, etc ... Ces considérations purement électroniques sont très importantes dans la réalisation d'un SoC et doivent être étudiées à l'aide d'outils de simulation et d'évaluation de performances propres aux différentes technologies et fabriquants (Xilinks, Virtex,...). Afin de ne pas rompre la chaîne de raffinements et conserver les garanties apportées par les preuves réalisées, nous avons proposé de fournir une description, manipulable et exécutable, directement issue des modèles.

Nous avons choisi le langage SystemC pour langage cible. Ce choix s'est imposé au vu des outils disponibles et des possibilités de tests de SystemC.

7.1 Principes généraux

Pour définir clairement une fonction de traduction, il convient de préciser la nature des modèles B événementiels traduisibles ainsi que le résultat donné par la fonction de traduction. Ayant déjà étudié, de manière abstraite et générale, la forme et les propriétés d'une hiérarchie de dépendance nous sommes en mesure de définir la classe des modèles B traduisibles. De plus, le code SystemC produit doit être exploitable et implanter correctement la relation de dépendance. L'étude déjà présentée de l'ordonnanceur SystemC et du comportement des processus lors de la simulation SystemC va également nous permettre de vérifier que la traduction conserve bien les propriétés du modèle source. Nous allons utiliser l'instanciation de modèle générique afin de construire un modèle B de l'exécution du module produit et montrer qu'il a les mêmes comportements que le modèle abstrait source.

7.1.1 Classe de modèle B

Nous allons décrire en détail la classe des modèles B événementiels candidats à une traduction. Notre traduction se propose de transformer un modèle B appartenant à une classe spécifique de modèles en une description SystemC. La traduction proposée produit un code SystemC destiné à la partie matérielle du SoC à partir d'un modèle B concret et sans indéterminisme. Le modèle B, source de notre traduction, comporte quelques contraintes qui sont décrites de manière générale dans la définition suivante :

Définition 4 *la classe C des modèles B traduisibles par la fonction de traduction TRAD est l'ensemble des modèles M(dag,V,v,events) contenant :*

- *VALUE : un ensemble chargé de représenter l'état d'évaluation d'une tâche et contenant trois valeurs OK, KO et IND.*

- *DATA : un ensemble abstrait représentant les données traitées par l'application (dans le cas du projet EQUAST, les paquets TS).*
- *current : une variable décrivant la donnée courante.*
- *V : les variables de contrôle du modèle de type VALUE et liées par des propriétés de dépendance.*
- *dag : le graphe acyclique déduit de l'ensemble des relations de dépendances décrites dans l'invariant du modèle sous forme de prédicat du type* $B \neq IND \Rightarrow A = OK$.
- *v : les variables d'implantation.*
- *c : les constantes du modèle permettant la caractérisation des données.*
- *events : la liste des événements du modèle.*

Le résultat de la traduction est un module SystemC constitué de processus et de signaux implantant le modèle source.

La définition donnée précise la nature des modèles sources et indique que l'invariant du modèle doit contenir la description d'une hiérarchie *dag* entre les tâches. Cette description n'est pas faite sous la forme d'une constante (c'était le cas dans les modèles généraux afin de pouvoir raisonner) mais d'un ensemble de prédicats contenus dans l'invariant du modèle. Les variables V impliquées dans les relations de dépendance sont des variables de type $VALUE$. A chaque sommet s du graphe (tâche) est associé un couple d'événements modélisant son exécution. Un prédicat permet de distinguer une exécution correcte et incorrecte des tâches. Ce prédicat (ou sa négation) est présent dans les gardes des événements associés aux différentes tâches modélisées.

Les variables d'implantation

Définition 5 *Toute variable V, n'étant ni la variable current, ni une variable d'ordonnancement, est une variable d'implantation. Elle sert aux traitements du système modélisé et peut être de différents types.*

Ainsi, toutes les variables ne servant pas à la synchronisation sont considérées comme des variables propres au système décrit et servant aux calculs effectués par celui-ci. On a donc une en-tête du type :

VARIABLES
$current, A, B, C, count$
INVARIANT
$/* \ typage \ */$
$current \in DATA \wedge$
$count \in \mathbb{N} \wedge$
$A \in VALUE \wedge$
$B \in VALUE \wedge$
$C \in VALUE \wedge$
$/* \ graphe \ de \ taches \ */$
$B \neq IND \Rightarrow A = OK \quad (B \prec A) \wedge$
$C \neq IND \Rightarrow A = OK \quad (C \prec A) \wedge$
$C \neq IND \Rightarrow B = IND \quad (C \sharp B)$

L'exemple précédent contient toutes les possibilités de variables offertes par la classe de modèles B étudiés. De même, le choix du type des variables d'implantation est libre et ne présente aucune restriction. Afin de permettre une traduction complète, il convient cependant d'utiliser des types "simples" : les naturels $(\mathbb{N}, \mathbb{N}_1)$, les booléens $\mathbb{B}$, les tableaux bornés d'entiers ou de booléens $binf..bsup \rightarrow \{\mathbb{N}, \mathbb{B}\}$.

Initialisation

L'ensemble des variables d'ordonnancement est initialisé à IND. Les variables d'implantation sont initialisées à une valeur par défaut [15]. Reste le cas de la variable *current* représentant la donnée courante

[15] Comme dans la plupart des langages les entiers sont initialisés par défaut avec la valeur 0.

en provenance de l'environnement. Le système modélisé ne peut pas imposer de valeur particulière à cette variable. Par conséquent, l'initialisation de cette variable se fait par une substitution :∈ illustrant bien le non-contrôle de l'application modélisée sur celle-ci : une nouvelle valeur de l'ensemble $DATA$ est choisie. On obtient une clause d'initialisation du type :

INITIALISATION
$current :\in DATA \parallel$
$count := 0 \parallel$
$A := IND \parallel$
$B := IND \parallel$
$C := IND$

Les couples d'événements

Pour toute variable d'ordonnancement V du modèle, il existe un couple d'événements eventOK_V et eventKO_V réalisant les traitements de la tâche associée à la variable d'ordonnancement V. Ce couple d'événements a la forme :

eventV_OK $\widehat{=}$
SELECT
$V = IND \wedge$
$\bigwedge P_i = OK \wedge$
$pred_{V,q,c}$
THEN
$V := OK \parallel$
$q := q'$
END

eventV_KO $\widehat{=}$
SELECT
$V = IND \wedge$
$\bigwedge P_i = OK \wedge$
$\neg pred_{V,q,c}$
THEN
$V := KO$
END

avec P_i les variables d'ordonnancement précédant V, v des variables d'implantation et c des constantes. La garde des deux événements d'un même couple est disjointe afin de modéliser le cas de dysfonctionnement et le cas de fonctionnement correct. Les deux événements modélisant l'exécution d'une tâche T ne peuvent se déclencher que si les tâches précédant la tâche T se sont correctement terminées ($\bigwedge P_i = OK$). Les traitements eux-mêmes sont représentés par le prédicat $pred_{V,v,c}$ qui est un prédicat construit sur les variables d'implantation et les constantes du modèle en rapport avec la variable d'ordonnancement V. Si ce prédicat est déterministe alors on est capable de proposer une implantation de la tâche. Dans un cas de non-déterminisme, il convient de faire un développement plus approfondi afin de lever l'indéterminisme à l'aide du raffinement. Dans ce cas, la traduction n'est pas possible car le comportement du modèle est encore trop abstrait.

Des développements de ce genre ont été présentés dans le chapitre 3. En effet, dans le cadre du projet EQUAST, les premiers développements menés ont été des tâches particulièrement complexes. C'est seulement par la suite que l'application a été considérée dans sa globalité. Dans le cadre d'une approche conduite par le raffinement (de l'abstraction à l'implantation), la prise en compte des communications entre les différents processus ou tâches n'implique pas d'avoir précédemment implanté précisément chaque tâche.

L'événement reset

L'événement reset est un cas particulier du modèle. Cet événement est le seul événement indéterministe autorisé, il est chargé de la réinitialisation du système. Lorsqu'il se déclenche, il "remet à zéro" l'ensemble des tâches considérées. Il est possible d'envisager une réinitialisation partielle l'application, qui laisserait des tâches actives ou en cours d'exécution durant la réception de plusieurs données successives [16]. Nous n'avons pas considéré cette possibilité pour l'instant.

[16] Ce cas de figure est présent dans l'étude de cas EQUAST, pour un paramètre comme *PAT_error* par exemple.

reset $\widehat{=}$
SELECT
$\forall x.(x \in \{A..Z\} \Rightarrow x = OK) \vee$
$\exists x.(x \in \{A..Z\} \wedge x = KO)$
THEN
$current :\in DATA \parallel$
$A := IND \parallel$
...
$Z := IND$
END

L'événement reset modélise également la réception d'une nouvelle donnée. Il affecte, de manière indéterminée, une nouvelle valeur à la variable *current* afin de simuler la réception d'une nouvelle donnée. Cet événement est en fait une réinitialisation complète du système une fois celui-ci dans un état de terminaison pour la donnée en cours. Comme précédemment, la politique de gestion des erreurs de l'application conditionne l'écriture de l'événement reset. Lorsque le choix d'une politique de réinitialisation a été effectué, l'indéterminisme de l'événement est supprimé.

7.1.2 Résultat de traduction

Le résultat de la fonction de traduction est un module SystemC réalisant les traitements (dans la mesure du possible, en fonction du niveau d'abstraction ou de complexité) de l'application modélisée mais, surtout, respectant les contraintes d'ordonnancement établies par la hiérarchie incrémentale construite dans les différents modèles B.

Nous nous contenterons donc de générer un module (interface et traitements) unique (voir figure 7.1). Il est inutile de générer tout un contexte au module produit, celui-ci ayant plusieurs utilités. En effet, si un tel module est le résultat d'une analyse formelle poussée du cahier des charges d'une application, c'est une brique de base pour la conception électronique d'un SoC. Le concepteur électronique peut utiliser ce module pour tester ses comportements sur différentes technologies. Dans ce cas, le concepteur va se servir du mécanisme de simulation de SystemC pour tester et valider, par exemple, les interactions du module produit avec d'autres (exploitation des résultats dans le cas d'applications dédiées de métrologie).

Dans une optique plus orientée "performance", le concepteur électronique va estimer le temps de traitement de l'architecture en fonction de la technologie utilisée (FPGA, DSP, etc). Dans ce cas, il peut simuler les temps de traitement de chaque tâche pour différentes technologies possibles et valider les choix faits en fonction des contraintes temporelles imposées.

Une autre utilisation est la production de circuit : le concepteur traduit le module SystemC en module VHDL à l'aide d'un traducteur puis synthétise la *netlist* permettant la gravure du circuit. Dans ce troisième cas de figure, il est implicitement considéré que les tests de respect des contraintes temporelles ont été réalisés.

Les multiples utilisations possibles d'un module SystemC limitent donc l'intérêt de construire un exécutable SystemC. La construction d'un tel exécutable demande par ailleurs l'écriture d'un nombre important de modules d'interfaçage simulant l'arrivée des données par une lecture dans un fichier de test par. L'intérêt de la production de tels modules est assez limité, c'est pourquoi nous nous contenterons de produire un unique module SystemC réalisant les traitements du système.

7.2 Algorithme de traduction

Notre traduction a lieu en deux étapes : production de la déclaration du module (son interface) puis production des traitements et du code associé à chaque processus. Plus formellement, la fonction de traduction $TRAD$ suit l'algorithme suivant :

```
TRAD(dag,X,Inv(X),events) =
```

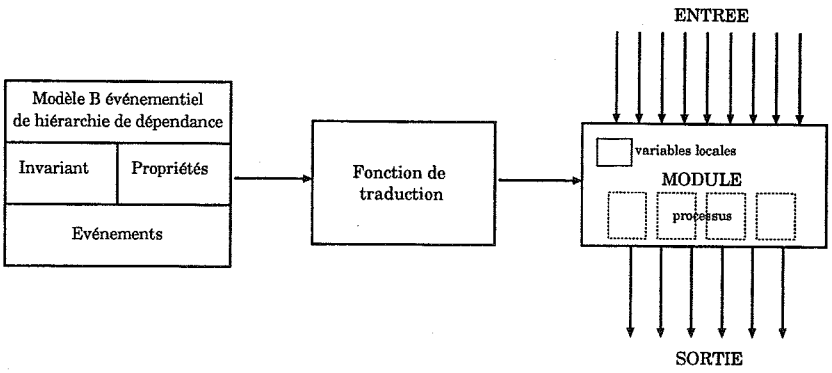


FIG. 7.1 – Forme générale de la traduction

```
make_interface(dag,V);
make_processes(dag,V,v),events);
```

avec `make_interface` la procédure de construction de l'interface et `make_processes` la procédure de construction des processus du module.

7.2.1 Construction de l'interface du module

L'interface du module doit contenir la déclaration de l'ensemble des variables locales du module, la déclaration des ports de communication (entrées et sorties) ainsi que la déclaration des différents processus du module et leurs listes de sensibilité. Nous partageons la construction de l'interface en trois grandes étapes qui suivent la syntaxe imposée par SystemC après la création d'un fichier `Archi.h` chargé de contenir le texte produit.

```
make_interface(dag) =
  creer_fichier(Archi.h);
  header(dag);
  declaration(dag);
  make_ctor(dag);
```

La procédure `header()` est chargée de produire l'en-tête du fichier et en particulier la déclaration des différents ports et les variables locales de type `signal`. La déclaration des variables locales est déléguée à la procédure `variables` qui doit faire une conversion/équivalence de type.

Pour chaque sommet V du graphe, deux signaux booléens sont générés (`endingV` et `errorV`) permettant le codage des trois valeurs OK, KO et IND. Comme nous disposons d'un codage à 4 valeurs, nous avons choisi de propager les erreurs et d'implanter la valeur OKO comme décrit dans le chapitre précédent.

```
header(dag) =
  ecrire(''#include<systemc.h>'' , Archi.h);
  ecrire(''SC_MODULE(Archi) {'' , Archi.h);
  ecrire(''sc_in<bool> clock;'' , Archi.h);
  ecrire(''sc_signal<bool> ready;'' , Archi.h);
  ecrire(''sc_signal<bool> reset;'' , Archi.h);
  variables(X,Inv(X));
  pour V dans dag faire
    ecrire(''sc_signal<bool> endingV;'' , Archi.h);
    ecrire(''sc_signal<bool> errorV;'' , Archi.h);
  finpour
```

La procédure `variables()` permet la déclaration, en tant que signaux pour des raisons de synthétisabilité, des variables d'implantation du modèle. La bibliothèque SystemC permet la déclaration de signaux de types relativement complexes `int`, `bool` et même des tableaux.

```
variables(X,Inv(X)) =
  pour v dans X faire
    ecrire(``sc_signal<Inv(X)> v;'' , Archi.h);
  finpour
```

La procédure `declaration` est extrêmement simple et génère, pour chaque variable de type *VALUE* sommet du graphe (càd les sommets du dag), la déclaration d'un processus spécifique. Elle déclare également le processus `RESET`, futur processus de supervision chargé d'indiquer le changement de phase aux autres processus du module.

```
declaration(dag) =
  pour V dans dag faire
    ecrire(``void procV();'' , Archi.h);
  finpour
  ecrire(``void RESET();'' , Archi.h);
```

Enfin, la procédure `make_ctor` produit la liste des sensibilités des différents processus à l'aide du graphe. Elle est également chargée de produire la liste de sensibilité du processus `RESET`, le seul processus non issu de la hiérarchie.

```
make_ctor(dag) =
  ecrire(``SC_CTOR(Archi){'', Archi.h);
  pour V dans dag faire
    ecrire(``SC_METHOD(procV);'', Archi.h);
    si V dans root(dag) alors
      ecrire(``sensitive_pos << ready;'' , Archi.h);
    sinon
      pour W dans dag[{V}] faire
        ecrire(``sensitive_pos << endingW;'' , Archi.h);
      finpour
    finsi
  finpour


  ecrire(``SC_METHOD(RESET);'' , Archi.h);
  ecrire(``sensitive_pos << clock;'' , Archi.h);
  ecrire(``} '' , Archi.h);
  ecrire(``}; '' , Archi.h);
```

La construction de l'interface du module `Archi` est terminée et est syntaxiquement correcte. Les variables internes du module (attribut de l'objet) sont déclarées et initialisées dans le constructeur du module. L'ensemble des processus et variables est désormais disponible et peut être utilisé dans le fichier de description des traitement `Archi.cpp`.

7.2.2 Construction des processus

La phase de génération de la description des processus à proprement parlé commence par la création d'un fichier `Archi.cpp` chargé de contenir le texte du module SystemC. La procédure `processes` permet de générer les processus implantant les différentes tâches du système. Enfin la procédure `make_reset` génère le code du processus `RESET`.

```
make_processes(dag,X,Inv(X),events) =
  creer_fichier(Archi.cpp);
  processes(dag,X,Inv(X),events);
  make_reset(dag);
```

La procédure processes ci-dessous construit, pour chaque noeud V du graphe une méthode procV (la description d'un processus). Cette méthode a les caractéristiques suivantes :

- elle teste en premier lieu la valeur du signal de réinitialisation reset,
- elle teste ensuite la valeur des signaux de terminaison en provenance des processus parents,
- elle vérifie ensuite la valeur des signaux d'erreurs en provenance des processus parents.

C'est seulement une fois ces différentes vérifications effectuées que la tâche est réellement exécutée. Comme nous l'avons déjà précisé, la production de code n'est envisageable que si les traitements du processus sont déterministes et simples. Dans le cas contraire, un développement spécifique est nécessaire afin de supprimer l'indéterminisme et de produire un algorithme. Le résultat de ce développement sera encapsulé dans le code SystemC généré afin de respecter l'ordonnancement établi.

```
processes(dag,X,Inv(X),events) =
  ecrire(''#include Archi.h'' , Archi.cpp);
  pour V dans dag  faire
    ecrire(''void procV() {'' , Archi.cpp);
    ecrire(''if not(reset.read()) {'' , Archi.cpp); %% test reset

    ecrire(''if ( true'', Archi.cpp);   %% test execution parents
    pour W dans dag[{V}] faire
      ecrire(''&& endingW.read()'' , Archi.cpp);
    finpour
    ecrire('') { '', Archi.cpp);

    ecrire(''if ( false'', Archi.cpp);  %% test correction parents
    pour W dans dag[{V}] faire
      ecrire(''|| errorW.read()''; Archi.cpp);
    finpour
    ecrire('') { '', Archi.cpp);

    ecrire(''errorV.write(true);'' , Archi.cpp);
    ecrire(''endingV.write(true);'' , Archi.cpp);
    ecrire(''} else {'' , Archi.cpp);


    ecrire(''if ''+pred_V+''{'', Archi.cpp);  %% traitements
    ecrire(substitutions(eventVOK); Archi.cpp);
    ecrire(''endingV.write(true);'' , Archi.cpp);
    ecrire(''} else { '' , Archi.cpp);
    ecrire(''endingV.write(true);'' , Archi.cpp);
    ecrire(''errorV.write(true);'' , Archi.cpp);
    ecrire(''} '' , Archi.cpp);

    ecrire(''} '' , Archi.cpp);
    ecrire(''} '' , Archi.cpp);

    ecrire(''} else {'' , Archi.cpp);   %% reinitialisation
    ecrire(''endingV.write(false);'' , Archi.cpp);
    ecrire(''errorV.write(false);'' , Archi.cpp);
    ecrire(''} '' , Archi.cpp);
```

```
    ecrire(``} '' , Archi.cpp);  %% fin du processus
  finpour
```

Dans l'algorithme précédent, la fonction `substitutions()` permet la traduction des substitutions simples de la partie action de l'événement considéré.

Enfin la procédure `make_reset` joue un rôle tout à fait similaire à la procédure `processes` mais pour le processus de réinitialisation. La méthode SystemC définissant le processus de réinitialisation a le comportement suivant :

- elle vérifie que les feuilles du graphe ont toutes été exécutées,
- elle indique le début de la phase de la réinitialisation par une écriture sur le signal `reset`,
- elle vérifie que les processus du graphe ont tous fini la réinitialisation avant de signaler la fin de l'étape de réinitialisation.

```
make_reset(dag) =
  ecrire(`` void reset() {'' , Archi.cpp);
  ecrire(``if ( true '' , Archi.cpp);
    pour V dans leaves(dag) faire
      ecrire(``&& endingV.read()'' , Archi.cpp);
    finpour
  ecrire(``) {'' , Archi.cpp);
  ecrire(``reset.write(true);'', Archi.cpp);
  ecrire(``ready.write(false);'', Archi.cpp);
  ecrire(``} else {'' , Archi.cpp);


  ecrire(``if (true'' , Archi.cpp);
    pour V dans dag faire
      ecrire(`` && (! endingV.read())'' , Archi.cpp);
    finpour
  ecrire(``) {'' , Archi.cpp);
  ecrire(``reset.write(false);'', Archi.cpp);
  ecrire(``ready.write(true);'', Archi.cpp);
  ecrire(``} '' , Archi.cpp);
  ecrire(``} '' , Archi.cpp);
  ecrire(``} '' , Archi.cpp);
```

La procédure précédente permet la construction d'un processus "contrôleur" qui est commandé par une horloge et qui assure cependant un contrôle correct (mais pas optimal) de l'application quelle que soit la cadence de l'horloge. La génération de code SystemC implantant un modèle B événementiel d'application composée de processus concurrents est terminée.

7.2.3 Remarques et limitations

Le code produit ne comporte aucune notification manuelle d'événements. Il n'utilise, en particulier, aucune notification immédiate. Il évite ainsi tout risque de boucle infinie de la phase d'évaluation empêchant la remise à jour des variables du système. Ce code n'utilise que des processus de type `METHOD` et aucun `THREAD` qui ne sont pas, par nature, synthétisables [74].

Par ailleurs, la traduction définie a pour but essentiel la préservation de l'ordre d'exécution proposé et non la génération d'un code optimisé. De ce fait, le code SystemC des tâches complexes ne peut être généré et de telles tâches doivent faire l'objet de modélisations spécifiques. En effet, en accord avec la philosophie du raffinement appliquée au cours de nos travaux, nous considérons que la construction d'un ordre d'exécution des tâches peut être réalisé avant la description détaillée de la réalisation de celle-ci. Ainsi, le code SystemC produit pour des tâches complexes est un squelette de code assurant la synchronisation de l'exécution.

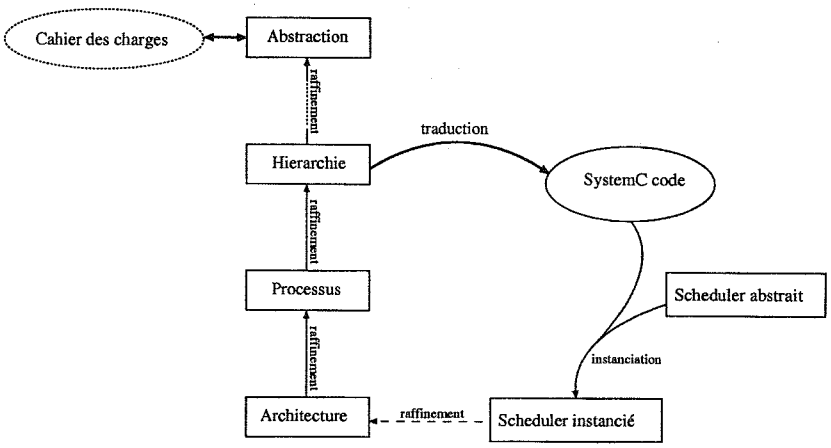


FIG. 7.2 – Démarche générale de validation de traduction

Cette démarche est également cohérente avec le flot de conception électronique : le code SystemC produit n'est pas définitif et est surtout un outil de test permettant l'évaluation des performances de différentes technologies.

7.3 Preuve de traduction

Dans cette section nous allons montrer que notre traduction produit un module SystemC implantant correctement le modèle B événementiel source. Nous avons défini la traduction comme un raccourci syntaxique aux différentes étapes de raffinement permettant de passer d'une spécification à une implantation. La preuve que nous devons établir s'apparente donc à une preuve de raffinement : préservation des comportements, etc.

Nous proposons de prouver la cohérence de la traduction à l'aide de la méthode B. En effet, les travaux que nous avons réalisés sur SystemC (chapitre 5) nous ont fourni des modèles B génériques de l'exécution d'un programme. Nous pouvons donc raisonner sur les programmes SystemC et instancier nos modèles génériques pour un programme spécifique. Dans le cadre de la validation de la traduction, le programme SystemC considéré sera un programme "abstrait" caractérisant la classe des programmes produits par la fonction de traduction.

Si nous arrivons à produire un modèle *ScheduledArchi* à la fois instance des modèles abstraits d'exécution de SystemC et raffinement de la hiérarchie formelle source (*Architecture*), nous aurons montré la validité de la fonction de traduction par bisimulation car nous aurons montré que le modèle *ScheduledArchi* a les comportements de la hiérarchie et s'exécute sous la contrainte de l'ordonnanceur SystemC, respectant ainsi la sémantique d'exécution de SystemC. La figure 7.2 résume l'approche générale proposée.

7.3.1 Construction d'un modèle d'exécution *ScheduledArchi*

Pour faire la preuve de la correction de la traduction nous devons construire un modèle B événementiel représentant un programme produit par la traduction. Comme nous désirons faire une preuve générale, le modèle doit représenter la classe des modules SystemC obtenus. La fonction de traduction produit un module SystemC `Archi` contenant autant de processus que de tâches distinctes. Nous avons déjà présenté une technique permettant la construction d'un modèle B événementiel représentant l'exécution d'un programme SystemC, figure 5.11 page 141, chapitre 5.

Une première étape consiste en une analyse simple de la structure des processus déclarés. Tous les processus générés par la traduction sont basés sur le même squelette et il est facile de reconstruire un arbre binaire de décision. La figure 7.3 présente la construction d'un tel arbre en partant du squelette d'un processus. Les tests des instructions conditionnelles sont les noeuds de l'arbre et les feuilles de l'arbre

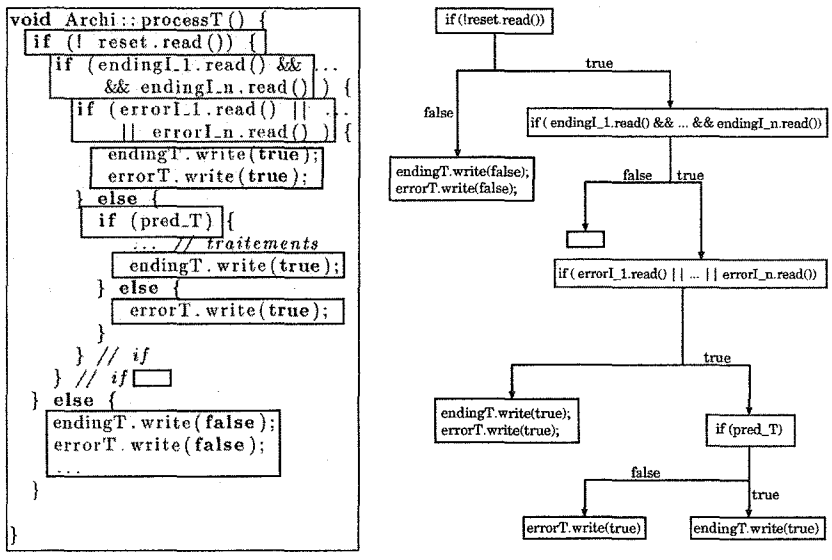


FIG. 7.3 – Analyse de flot d'un processus SystemC

sont les différents blocs du processus. Une exécution du processus se traduit en un chemin dans l'arbre qui va de la racine à une feuille selon le résultat des tests conditionnels. Dans notre cas, on construit un arbre ayant 4 feuilles et qui représente toutes les exécutions possible d'un processus. L'arbre de décision est le résultat de cette première étape et est issu d'une analyse de flot simple.

La seconde étape permet la reconstruction des événements du modèle B événementiel chargé de représenter l'exécution du programme SystemC. A partir de l'arbre de décision binaire obtenu à l'étape précédente il est facile de dériver des événements B. En effet, les règles de reconstruction de Jean-Raymond Abrial font exactement le travail inverse : elles produisent un algorithme en réunissant les différents événements d'un modèle. Pour un processus T donné différent du processus de réinitialisation $RESET$, nous générons alors 5 événements correspondant aux 5 feuilles et aux cinq exécutions possibles :

- evaluateReset qui représente une exécution du processus T réinitialisant ses signaux de sortie,
- evaluateNothing qui représente une exécution du processus T suite à un front montant d'une de ses entrées mais qui ne modifie et ne calcule rien,
- evaluateKO qui représente une exécution du processus T ayant pour conséquence l'émission de la valeur `true` sur le signal `errorT`,
- evaluateOK qui représente une exécution du processus T ayant pour conséquence l'émission de la valeur `true` sur le signal `endingT`,
- evaluateOKO qui représente une exécution du processus T ayant pour conséquence la propagation d'une erreur antérieure par l'émission de la valeur `true` sur les signaux `endingT` et `errorT`.

La figure 7.4 illustre la création des événements B associés aux différentes feuilles de l'arbre de décision. Les événements B présentés sur la figure ne sont pas complets, le code SystemC doit être réécrit sous la forme d'expressions B événementielles. Une telle réécriture n'est cependant pas un problème et se fait de manière relativement naturelle.

Les cinq événements suivants sont le résultat de ce procédé de reconstruction. Les événements en question constituent la partie dynamique du modèle *ScheduledArchi*.

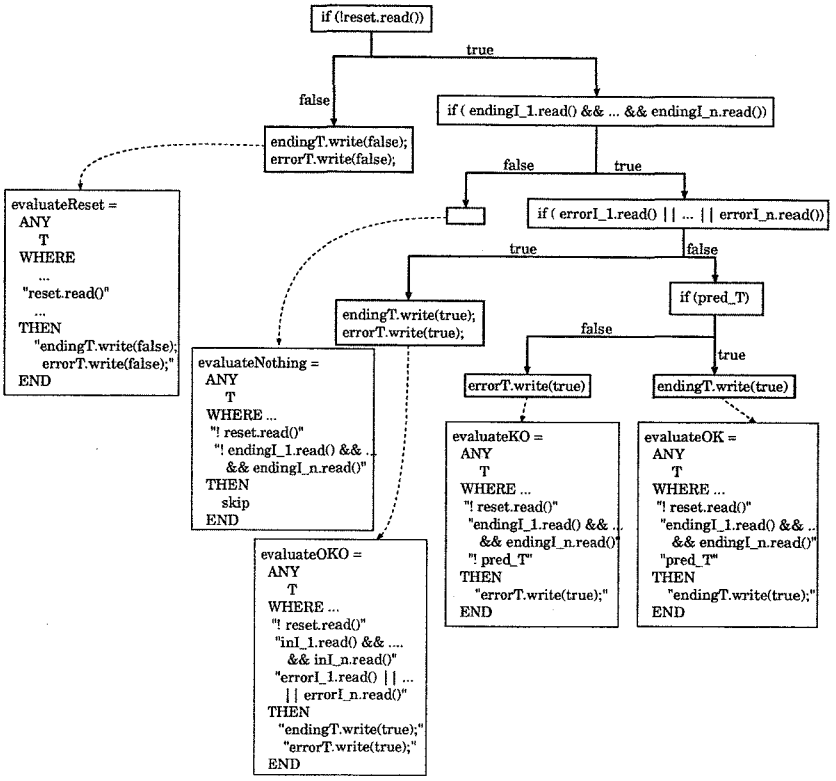


FIG. 7.4 – Reconstruction d'événement

EvaluateReset $\widehat{=}$
ANY
p
WHERE
$p \in runnable \wedge$
$p \neq RESET \wedge$
$value(Creset) = high$
THEN
$runnable := runnable - \{p\} \parallel$
$newValue := newValue \Leftarrow \{error(p) \mapsto low, ending(p) \mapsto low\}$
END

EvaluateNothing $\widehat{=}$
ANY
p
WHERE
$p \in runnable \wedge$
$p \neq RESET \wedge$
$value(Creset) = low \wedge$
$value[inputs[\{p\}] \cap Cend] \neq \{high\}$
THEN
$runnable := runnable - \{p\}$
END

EvaluateOK $\widehat{=}$
ANY
p
WHERE
$p \in runnable \wedge$
$p \neq RESET \wedge$
$value(Creset) = low \wedge$
$value[inputs[\{p\}] \cap Cend] = \{high\} \wedge$
$predicate(p) = TRUE$
THEN
$runnable := runnable - \{p\} \parallel$
$newValue :=$
$newValue \leftarrow \{ending(p) \mapsto high\}$
END

EvaluateKO $\widehat{=}$
ANY
p
WHERE
$p \in runnable \wedge$
$p \neq RESET \wedge$
$value(Creset) = low \wedge$
$value[inputs[\{p\}] \cap Cend] = \{high\} \wedge$
$predicate(p) = FALSE$
THEN
$runnable := runnable - \{p\} \parallel$
$newValue :=$
$newValue \leftarrow \{error(p) \mapsto high\}$
END

EvaluateOKO $\widehat{=}$
ANY
p
WHERE
$p \in runnable \wedge$
$p \neq RESET \wedge$
$value(Creset) = low \wedge$
$value[inputs[\{p\}] \cap Cend] = \{high\} \wedge$
$\exists i.(i \in inputs[\{p\}] \cap Cerr \wedge value(i) = high)$
THEN
$runnable := runnable - \{p\} \parallel$
$newValue := newValue \leftarrow \{error(p) \mapsto high, ending(p) \mapsto high\}$
END

Nous construisons donc une instanciation, *ScheduledArchi*, du modèle générique *CompleteScheduler* qui permettra la modélisation de la simulation. Les différents événements présentés raffinent l'événement abstrait evaluate qui modélisait l'exécution d'un processus. Deux autres événements représentant les exécutions du processus $RESET$ raffinent également l'événement evaluate. Les propriétés particulières de l'instanciation sont données ci-dessous et expriment les spécificités de l'architecture considérée.

CONSTANTS
$Cend, Cerr, ending, error, Creset$
$inputs, sensitivity, produce$
PROPERTIES
$Cend \subseteq CHANNELS \wedge$
$Cerr \subseteq CHANNELS \wedge$
$Cend \cap Cerr = \emptyset \wedge$
$Creset \in CHANNELS \wedge$
$Creset \notin (Cend \cup Cerr) \wedge$
$Cend \cup Cerr \cup \{Creset\} = CHANNELS \wedge$
$ending \in PROCESSUS - \{RESET\} \rightarrowtail\!\!\!\!\!\!\rightarrow Cend \wedge$
$error \in PROCESSUS - \{RESET\} \rightarrowtail\!\!\!\!\!\!\rightarrow Cerr \wedge$
$ending \cap error = \emptyset \wedge$
...
$sensitivity = (inputs; produce)$

Les ensembles abstraits ne sont pas instanciés par des ensembles plus concrets. Seules de nouvelles constantes viennent décrire les spécificités de l'application. Les constantes ajoutées et leurs propriétés sont

exactement celles des modèles abstraits *Channels* et *Architecture* présentés précédemment et décrivant l'implantation de l'architecture. L'instanciation explicite cependant les relations constantes comme *inputs* et surtout *sensitivity*. La propriété $sensitivity = (inputs; produce)$ indique que les processus du système sont sensibles à l'ensemble de leurs signaux d'entrée comme c'est le cas dans le code SystemC produit. Grâce aux propriétés de ce modèle nous allons pouvoir raisonner sur une abstraction du résultat de la traduction et montrer, à l'aide de la relation de raffinement, qu'il implante bien la hiérarchie.

EvaluateRESET $\widehat{=}$
ANY
p
WHERE
$p \in runnable \wedge$
$p = RESET \wedge$
$ran(ending) = \{high\}$
THEN
$runnable := runnable - \{p\} \parallel$
$newValue := newValue \leftarrow \{Creset \mapsto high\} \parallel$
$predicate :\in PROCESSUS \rightarrow Bool \parallel$
$phase := run$
END

EvaluateRNot $\widehat{=}$
ANY
p
WHERE
$p \in runnable \wedge$
$p = RESET \wedge$
$ran(ending \cup error) = \{low\}$
THEN
$runnable := runnable - \{p\} \parallel$
$newValue := newValue \leftarrow \{Creset \mapsto low\} \parallel$
$phase := run$
END

EvaluateRESETnot $\widehat{=}$
ANY
p
WHERE
$p \in runnable \wedge$
$p = RESET \wedge$
$ran(ending \cup error) = \{high, low\}$
THEN
$runnable := runnable - \{p\} \parallel$
$phase := run$
END

Le modèle instancié a les comportements d'un programme SystemC exécuté par l'ordonnanceur et est un modèle abstrait de l'exécution du résultat de la traduction.

7.3.2 Montrer le raffinement

Nous devons prouver que le modèle instancié *ScheduledArchi* précédent raffine le modèle *Architecture*. En effet, par sa construction (et avec la garantie des preuves d'instanciation) le modèle *ScheduledArchi*

a les propriétés et les comportements du modèle *CompleteScheduler*. De ce fait, il représente bien l'exécution d'une architecture particulière par l'ordonnanceur SystemC. Nous devons donc "relier" le modèle *ScheduledArchi* au modèle *Architecture* (voir figure 7.2) pour montrer que notre description formelle du programme SystemC, résultat de la traduction, a les mêmes comportements que le modèle *Architecture*. Nous aurons montré ainsi que la description formelle du résultat de la traduction a les mêmes comportements que la hiérarchie, source de la traduction. De cette manière, nous aurons validé la fonction de traduction en montrant que modèle source et programme résultat ont les mêmes comportements.

Nous ne pouvons pas faire une preuve de raffinement simple entre les deux modèles *Architecture* et *ScheduledArchi*. Nous allons donc introduire des raffinements intermédiaires permettant de passer de la description statique de l'architecture à la description de sa simulation. Les raffinements intermédiaires auront pour but l'introduction de différentes notions relatives à l'exécution (ensemble des processus *runnable*, liste de sensibilité, etc...).

Introduction de la notion de processus exécutable : modèle *ArchiProc*

Une première étape de ces raffinements intermédiaires est la prise en compte de la liste des processus dits "exécutables" gérée par l'ordonnanceur. La variable *runnable* nous permet de modéliser cette liste. Le raffinement va donc consister à ajouter cette variable et faire la preuve que les processus appartenant à cette liste sont des processus non encore évalués.

VARIABLES
...
INVARIANT
...
$runnable \subseteq PROCESSUS \wedge$
$\forall p.(p \in runnable \wedge p \neq RESET \wedge getValue(Creset) = low \Rightarrow getValue[outputs[\{p\}]] = \{low\}) \wedge$
...

La propriété quantifiée exprime le fait que les processus considérés comme exécutables sont des processus qui n'émettent rien encore sur leurs signaux de sortie. Cette propriété nous permet de supprimer le prédicat $getValue[outputs[\{p\}]] = \{low\}$ des gardes des événements progressOK et progressKO. ces deux événements représentent désormais l'exécution d'un processus dont l'ensemble des parents a été correctement exécuté. De la même façon, les autres événements du modèle sont raffinés afin de ne manipuler que des processus exécutables.

progressOK $\widehat{=}$
ANY
p
WHERE
$getValue(Creset) = low \wedge$
$p \neq RESET \wedge$
$p \in runnable \wedge$
$predicat(p) = TRUE \wedge$
$\forall i.(i \in inputs[\{p\}] \cap Cend \Rightarrow getValue(i) = high)$
THEN
$getValue := getValue \Leftarrow \{ending(p) \mapsto high\} \parallel$
$runnable := runnable - \{p\}$
END

progressKO $\widehat{=}$
ANY
p
WHERE
$getValue(Creset) = low \wedge$
$p \neq RESET \wedge$
$p \in runnable \wedge$
$predicat(p) = FALSE \wedge$
$\forall i.(i \in inputs[\{p\}] \cap Cend \Rightarrow getValue(i) = high)$
THEN
$getValue := getValue \Leftarrow \{error(p) \mapsto high\} \parallel$
$runnable := runnable - \{p\}$
END

Avec l'ajout de la variable *runnable* représentant la liste des processus exécutables, apparaissent de nouveaux événements chargés de la mise à jour de cette variable. Ces nouveaux événements décrivent le fonctionnement du scheduler SystemC et sont des versions abstraites des événements du modèle *CompleteScheduler* présentés dans le chapitre 5 sur la modélisation de l'ordonnanceur SystemC.

DeltaCycle $\widehat{=}$
ANY
S
SELECT
$S \subseteq PROCESSUS \wedge$
$\forall p.(p \in S \wedge getValue(Creset) = low \Rightarrow$
$getValue[outputs[\{p\}]] = \{low\}) \wedge$
$runnable = \emptyset$
THEN
$runnable := S$
END

L'événement DeltaCycle représente la mise à jour de la liste des processus exécutables une fois celle-ci vide. Les notions d'événements SystemC et de sensibilité n'ayant pas encore été introduites, l'événement reconstruit un ensemble de processus susceptible de correspondre.

L'ajout de la variable *runnable* a pour conséquence l'apparition d'un nouvel événement d'évaluation des processus progressNothing qui représente l'exécution d'un processus de *runnable* qui ne peut encore réaliser ses traitements. Un tel événement n'est pas présent dans les abstractions car de telles situations n'apparaissent pas. L'implantation, prenant en compte un ensemble de processus exécutable *runnable*, implique l'existence de ce genre de situation et c'est le raffinement qui nous permet de montrer que ces cas de figure ne compromettent pas le fonctionnement prévu du système. En effet, l'événement en question ne modifie que la variable d'exécution *runnable* et ne change pas l'état du processus p exécuté.

progressNothing $\widehat{=}$
ANY
p
WHERE
$getValue(Creset) = low \wedge$
$p \neq RESET \wedge$
$p \in runnable \wedge$
$\exists i.(i \in inputs[\{p\}] - \{Creset\} \wedge getValue(i) = low)$
THEN
$runnable := runnable - \{p\}$
END

Le modèle présenté est incomplet mais permet cependant de comprendre l'introduction de la variable *runnable*. L'ajout de cette variable permet la prise en compte, dans le modèle, de la gestion de la simulation SystemC. Afin de compléter cette prise en compte, il convient de considérer la notion d'événements SystemC et d'introduire la notion de δ cycle telle qu'elle est définie par l'ordonnanceur SystemC.

Introduction des événements SystemC : modèle *ArchiSC*

L'introduction des événements SystemC se fait de manière assez simple en définissant un ensemble abstrait SC_EVENTS ainsi qu'un certain nombre de constantes précisant les relations existantes entre signaux et événements SystemC. La liste de sensibilité des processus est définie par l'intermédiaire de la constante *sensitivity*.

SETS
SC_EVENTS
CONSTANTS
$C\_EVENTS, EOS, produce, sensitivity$
PROPERTIES
$C\_EVENTS \subseteq SC\_EVENTS \wedge$
$produce \in CHANNELS \rightarrowtail\!\!\!\!\!\!\rightarrow C\_EVENTS \wedge$
$sensitivity \in PROCESSUS \leftrightarrow SC\_EVENTS \wedge$
$sensitivity = (inputs; produce) \wedge$
$EOS \in NATURAL$

Le point le plus délicat des raffinements intermédiaires est le raffinement de la fonction $getValue$ représentant la valeur des signaux de communication en deux fonctions distinctes modélisant la mise à jour différée des valeurs introduite par le mécanisme de δ cycle de SystemC. On introduit les deux variables *value* et $newValue$ symbolisant la valeur courante et la valeur calculée des signaux de communication. La dernière propriété de l'invariant ci-dessous permet de lier la variable concrète *value* à la variable abstraite $getValue$ en ne considérant que les signaux d'entrée des processus de *runnable*. On introduit également les différentes variables nécessaires à la modélisation de l'ordonnancement comme δ mais aussi la variable $SCphase$ permettant de modéliser les différentes étapes de l'exécution de l'architecture.

VARIABLES
$predi, predicat, \delta, timed, time,$
$value, newValue, runnable, SCphase$
INVARIANT
$value \in CHANNELS \rightarrow VALUE \wedge$
$newValue \in CHANNELS \rightarrow VALUE \wedge$
$\delta \subseteq SC\_EVENTS \wedge$
$timed \in SC\_EVENTS \nrightarrow NATURAL \wedge$
$\delta \cup dom(timed) = \emptyset \wedge$
$C\_EVENTS \cup dom(timed) = \emptyset \wedge$
$time \in NATURAL \wedge$
$newValue = getValue \wedge$
$\forall c.(c \in CHANNELS \wedge c \in inputs[runnable] \wedge$
$c \notin outputs[runnable - \{RESET\}] \Rightarrow value(c) = getValue(c))$
...

Les cinq événements updateValue, NoEvent, EndingSimulation, timed et DeltaCycle sont exactement les événements définis dans le modèle *CompleteScheduler* du comportement d'une architecture SystemC exécutée. Ils préservent l'invariant du modèle courant mais expriment le fonctionnement de l'ordonnanceur SystemC.

L'ajout de la notion de δ cycle au modèle implique la réécriture des événements modélisant l'exécution d'un processus. Grâce aux différentes propriétés invariantes du modèle, il est cependant assez facile de montrer que la fonction abstraite de valuation $getValue$ est raffinée en deux fonctions concrètes $value$ et $newValue$. Les versions raffinées des événements sont présentées ci-dessous :

progressOK $\widehat{=}$
ANY
p
WHERE
$value(Creset) = low \wedge$
$p \neq RESET \wedge$
$p \in runnable \wedge$
$predicat(p) = TRUE \wedge$
$\forall i.(i \in inputs[\{p\}] - \{Creset\} \Rightarrow value(i) = high)$
THEN
$newValue := newValue \Leftarrow \{ending(p) \mapsto high\} \parallel$
$runnable := runnable - \{p\}$
END

progressKO $\widehat{=}$
ANY
p
WHERE
$value(Creset) = low \wedge$
$p \neq RESET \wedge$
$p \in runnable \wedge$
$predicat(p) = FALSE \wedge$
$\forall i.(i \in xinputs[\{p\}] - \{Creset\} \Rightarrow value(i) = high)$
THEN
$newValue := newValue \Leftarrow \{error(p) \mapsto high\} \parallel$
$runnable := runnable - \{p\}$
END

Les propriétés de collage liant la variable abstraite $getValue$ aux variables concrètes permettent de faire la preuve que ces versions définitives des événements sont des raffinements corrects.

Avec cette dernière étape, nous avons terminé le processus de raffinement d'une hiérarchie. Le modèle obtenu est l'implantation de la hiérarchie sous forme d'un ensemble de processus communiquant par l'intermédiaire de signaux. Les derniers raffinements nous ont permis de prendre en compte l'aspect dynamique de l'exécution de ces processus. Pour pouvoir montrer que la traduction définie produit une implantation correcte de la hiérarchie, il convient maintenant de comparer les deux modèles.

renommage de variables et d'événements

Les modèles *ArchiSC* et *ScheduledArchi* sont identiques aux noms des événements près. En effet, un simple renommage des événements progressOK, progressKO, ... en EvaluateOK, EvaluateKO, ... permet d'obtenir des modèles identiques. Pour les variables des deux modèles, nous avons été vigilant et avons employé les mêmes noms de variables pour désigner les mêmes éléments des deux systèmes. Nous avons donc construit un modèle identique en suivant deux chemins de raffinements différents. Certaines propriétés ont été ajoutées dans un modèle ou dans l'autre afin de permettre de faire les preuves de raffinement, mais la dynamique des deux modèles est la même : dans les deux modèles, les variables sont modifiées par des événements identiques (du point de vue des gardes et des substitutions).

7.4 Validité de la traduction

Afin de valider la fonction de traduction définie, nous avons décidé de comparer des modèles. Nous avons d'abord construit un modèle *générique* abstrait représentant l'exécution d'un programme SystemC, résultat de la traduction. Ce modèle a été obtenu par l'instanciation du modèle représentant la gestion de l'ordonnancement des processus par l'ordonnanceur SystemC.

Par ailleurs, nous avons construit, par raffinements successifs, un modèle représentant l'implantation d'une hiérarchie *générique* sous forme de processus concurrents communiquant via des signaux. Ce modèle *générique* a ensuite été raffiné afin de se rapprocher d'une simulation SystemC. A la fin du processus de raffinement, il apparaît que les deux modèles construits (par raffinement de la hiérarchie et par instanciation du modèle d'ordonnanceur) sont identiques (aux noms des événements près). Par conséquent, nous pouvons en conclure que la fonction de traduction permettant de passer d'un modèle B événementiel décrivant une hiérarchie à son implantation SystemC est correcte et conserve les comportements du modèle : nous avons ainsi validé notre fonction de traduction.

Le schéma de validation suivi montre par ailleurs que la traduction de modèle B vers le code SystemC n'est bien qu'un "simple" raccourci du processus de raffinement. En effet, la validation de la fonction de traduction se fait par raffinements successifs d'un modèle *générique* source. La preuve de validité fournie est donc générale et valable pour toutes les traductions de hiérarchies décrites sous la forme de modèles B spécifiques.

La fonction de traduction permet de préserver les propriétés des modèles B en particulier du point de vue des dépendances entre tâches et de l'ordonnancement. Le résultat de traduction est un module SystemC réalisant les traitements définis mais, surtout, respectant les propriétés invariantes des modèles. Il est également possible de vérifier les propriétés invariantes des différents modèles (hiérarchies) sur le code produit à l'aide d'un model checker [99]. Une vérification simple est de s'assurer de la cohérence des valeurs émises sur les signaux en particulier le respect de la relation de dépendance (les signaux de sorties ne sont positionnés qu'une fois les signaux d'entrées évalués). Une telle vérification permet aussi de s'assurer de la cohérence du code proposé et de la conservation des propriétés invariantes lors de la traduction.

Le module SystemC généré peut servir de point de départ à des considérations plus électroniques et permettre une exploration architecturale. C'est un renseignement et un outil important pour les électroniciens qui vont pouvoir tester différentes technologies d'implantation selon les contraintes de l'application.

Conclusions

Le travail réalisé durant cette thèse a permis d'étudier l'applicabilité d'une méthode formelle dans le cadre d'une étude de cas de taille réelle et d'évaluer l'apport de cette méthode dans la conception de SoC.

Projet EQUAST

Nos travaux au sein du projet EQUAST nous ont permis de confronter la méthode B événementielle à une étude de cas originale qui n'aboutit pas que sur la production de codes informatiques mais aussi sur la génération de circuits électroniques. Les différentes modélisations conduites au sein de ce projet ont permis de définir une méthode de conception fondée sur la preuve et assurant une continuité entre cahier des charges et architecture par l'utilisation du raffinement.

Nous avons pu expérimenter des solutions en collaboration avec les ingénieurs électroniciens et, plutôt que de réinventer entièrement une nouvelle méthode de conception, nous avons proposé l'utilisation de méthodes formelles rigoureuses dans le flot de conception déjà existant. En utilisant la méthode B événementielle, nous construisons des modèles clairs et précis de systèmes qui permettent de raisonner et de prouver des propriétés invariantes. Cette méthode est relativement différente des méthodes de vérification généralement utilisées en électronique (tests ou model checking).

Nous avons, dans un premier temps, travaillé sur des points de détail spécifiques de l'étude de cas. Ces réalisations spécifiques étaient particulièrement complexes et elles nous ont permis de comprendre les problèmes rencontrés. Nous avons alors expérimenté, en collaboration avec les électroniciens, plusieurs solutions permettant de répondre aux problèmes de conception (respect et compréhension du cahier des charges, choix architecturaux judicieux).

Dans cette première phase de notre travail, la méthode B et nos modélisations ont permis de communiquer en employant un vocabulaire commun (les mathématiques) permettant ainsi de comprendre les différentes solutions et problèmes apportés par chacun. L'aspect pédagogique du couple abstraction/raffinement a joué dans cette phase du projet un rôle extrêmement important.

Ces premiers travaux réalisés, nous avons compris que le flot de conception utilisé ne permettait pas de garantir la préservation des fonctionnalités et des propriétés des systèmes conçus. Toujours au sein du projet d'EQUAST, nous avons donc modélisé l'intégralité du cahier des charges et, avec l'aide du raffinement, structuré celui-ci. Dans ce cadre, l'utilisation (intensive) du raffinement a permis d'exhiber des relations de dépendances entre les différentes tâches constitutives de l'application. En collaboration avec les ingénieurs électroniques, nous avons déduit de ces relations une architecture et un ordonnancement permettant une implantation correcte du cahier des charges en un SoC.

La modélisation incrémentale du système a eu un rôle important sur la conduite du projet en suscitant les premières discussions "de fond" sur la norme et sur le sens de certaines vérifications. Nous considérons ces discussions comme des résultats importants de notre travail : nos modèles ont imposé une explication ou une explicitation du cahier des charges par les experts et, de ce fait, la conformité des réalisations, par rapport aux attentes initiales, est réelle.

L'illustration de ce travail est la justification apportée aux paramètres de QdS empiriquement définis. Les hiérarchies incrémentales issues de nos modèles ont permis de justifier le sens des paramètres de synthèse et de montrer leur pertinence au regard de l'ensemble des paramètres du système.

Par ailleurs, deux prototypes (AMETHYST II et SDVB-T-M) sont issus du projet EQUAST. Ces deux prototypes opérationnels, implantent la totalité de la norme d'analyse de qualité DVB et sont en

passe de devenir des produits du catalogue des partenaires industriels du projet. Un des prototypes est déjà utilisé en tête de réseau, au Japon, et évalue la qualité de transmission du réseau réel.

Généralisation

Au vu des résultats obtenus dans le projet EQUAST, nous avons décidé de généraliser la démarche suivie. Dans un premier temps nous avons étudié la classe des systèmes pouvant être décrits selon les principes de modélisation que nous avons employés. Pour réaliser cette étude nous avons décrit des modèles génériques de systèmes. Ces modèles ont servi de support à nos raisonnements. Le raffinement a également été utilisé afin de montrer que ces modèles d'application pouvaient être dérivés en une implantation à base de processus communiquant par l'intermédiaire de signaux.

La réalisation de ces modèles a eu pour conséquence la définition d'un mécanisme de traduction d'une classe de modèles B événementiels en code SystemC. Cette traduction, sorte de raccourci aux raffinements successifs, a pour résultat un code SystemC permettant une simulation rapide et efficace de la collaboration des processus. Le code produit peut alors permettre une étude plus électronique, en particulier la vérification du respect des contraintes temps-réel en fonction des technologies (type de circuits) choisies. Un premier noyau de traducteur est à finaliser.

Afin de pouvoir produire un code SystemC cohérent et conforme à nos idées, nous avons étudié SystemC et en particulier son mécanisme de simulation géré par un ordonnanceur spécifique. Une fois encore, nous avons utilisé la modélisation et le raffinement pour comprendre et expliquer le fonctionnement de l'ordonnanceur SystemC. L'écriture de modèles généraux expliquant le fonctionnement de la simulation SystemC permet également une instanciation en des modèles spécifiques permettant de prouver des propriétés de programmes SystemC particuliers.

Afin de valider la traduction définie, nous avons prouvé que les contraintes d'ordonnancement définies par les modèles B événementiels génériques des applications sont conservées par la simulation SystemC du programme résultat de la traduction. On construit le modèle B simulant l'exécution d'un programme issu de la traduction par instanciation du modèle générique de la simulation SystemC. Par une série de raffinements on peut montrer que le modèle instancié est également un raffinement des modèles généraux représentant la classe de systèmes concernés. Cette preuve de la conservation des comportements de l'abstraction nous a permis de conclure sur la validité de la traduction.

Nous sommes donc en mesure de proposer une méthodologie de conception basée sur la preuve et qui respecte la méthodologie orientée *modèle* en vigueur dans les SoCs. Nos travaux, basés sur le raffinement et la méthode B événementielle, assurent un respect du cahier des charges dans l'implantation par la préservation des comportements des modèles. Les modèles abstraits de l'application permettent au concepteur de déduire un ordonnancement et une architecture adaptée, toujours dans le respect du cahier des charges. L'architecture ainsi déduite se base sur la structure même de l'application et sur les rapports existants entre les différentes tâches la constituant. Avec la définition d'une fonction de traduction, cette déduction est même en partie automatisable.

Perspectives

Certains points intéressants n'ont pas pu être abordés dans cette thèse, faute de temps, ou parce qu'ils étaient trop éloignés de nos préoccupations immédiates. Nous allons présenter ici ces différents points et discuter rapidement de leur intérêt.

Dans un premier temps, afin de conclure les travaux entamés sur la traduction de modèles B en code SystemC, la réalisation d'un outil automatisant cette traduction est nécessaire. Bien que la programmation d'un tel outil ait débuté, le prototype actuel n'est pas encore en mesure de traduire de manière cohérente un modèle B événementiel. Cependant l'analyse lexicale et syntaxique des modèles B est déjà entièrement programmée et les principales structures de données de la traduction sont construites. La traduction telle qu'elle a été définie est implantable, à court terme, de manière rapide et efficace.

Un prolongement intéressant de ces travaux est la réalisation d'un environnement de développement plus complet. Si le raffinement et la preuve aident à la conception d'un SoC, le model checking est un

outil de vérification important. La simulation du code SystemC produit respecte les propriétés invariantes des modèles abstraits mais un nombre important de propriétés de vivacité, par exemple, peuvent être vérifiées. L'utilisation conjointe du raffinement et du model checking dans un outil de développement permettrait de garantir un haut niveau de conformité et de correction du système construit en utilisant les avantages de chaque approche selon la granularité de la description.

D'un point de vue plus formel, nos travaux permettent d'envisager la définition d'une notion de comparaison plus rigoureuse entre programmes SystemC. En effet, le "raffinement" est une notion centrale de la conception de SoC et est renforcée, dans SystemC notamment, par les différences de niveaux de description (TLM et RTL). Il n'existe cependant aucun lien formel entre deux descriptions SystemC successives ("raffinées") du même système, et le test est la seule vérification de la conservation des comportements entre deux descriptions.

Grâce aux modèles formels de simulation SystemC que nous avons réalisés, nous sommes en mesure de construire, par instanciation, un modèle représentant la simulation d'un programme spécifique. A l'aide de ce mécanisme, nous pouvons produire des modèles B des descriptions SystemC successives d'un même SoC. Les différents modèles obtenus peuvent donc être comparés, en particulier au regard du raffinement. L'idée est alors de définir une notion de *raffinement électronique* permettant de garantir la conservation de certaines propriétés spécifiques à l'électronique en s'appuyant sur le formalisme de la méthode B. Le passage d'une description TLM à une description RTL synthétisable serait alors garanti par des preuves formelles et respecterait un schéma de conception conservant les propriétés générales du système et en particulier ses fonctionnalités. Cependant,les propriétés souhaitées ainsi que la notion de raffinement envisagée doivent être spécifiques à la problématique électronique.

Bibliographie

[1] D. Abraham, D. Cansell, P. Ditsch, D. Méry, et C. Proch. Synthesis of the QoS for digital TV services. Dans *Proceedings of IBC'05*, Amsterdam, Septembre 2005. A paraître.

[2] J.-R. Abrial. Data semantics. Dans J. W. Klimbie et K. L. Koffeman, éditeurs, *IFIP TC2 Working Conference on Data Base Management*, pages 1–59. Elsevier Science Publishers (North-Holland), Avril 1974.

[3] J.-R. Abrial. *The B Book - Assigning Programs to Meanings.* Cambridge University Press, 1996. ISBN 0-521-49619-5.

[4] J.-R. Abrial. Etude système : méthode et exemple. Clearsy, Atelier B : `http ://www.atelierb.societe.com/ressources/articles/Portes.pdf`, Novembre 1998.

[5] J.-R. Abrial. Event driven circuit construction. MATISSE project, `http ://www.atelierb.societe.com/ressources/articles/cir.pdf`, Août 2000.

[6] J.-R. Abrial. B# : Toward a synthesis between Z and B. Dans D. Bert, J. P. Bowen, S. King, et M. Waldén, éditeurs, *ZB'2003 - Formal Specification and Development in Z and B*, volume 2651 de *Lecture Notes in Computer Science*, pages 168 – 177, Turku, Finland, Juin 2003. Springer-Verlag.

[7] J.-R. Abrial. Event based sequential program development : Application to constructing a pointer program. Dans K. Araki, S. Gnesi, et D. Mandrioli, éditeurs, *FME*, volume 2805 de *Lecture Notes in Computer Science*, pages 51–74. Springer, 2003.

[8] J.-R. Abrial et D. Cansell. Click'n'prove : Interactive proofs within set theory. Dans B. Basin, D. et Wolff, éditeur, *16th International Conference on Theorem Proving in Higher Order Logics - TPHOLs'2003, Rome, Italy*, volume 2758 de *Lecture notes in Computer Science*, pages 1–24. Springer, Septembre 2003.

[9] J.-R. Abrial, D. Cansell, et D. Méry. A mechanically proved and incremental development of IEEE 1394 tree identify protocol. *Formal Aspects of Computing*, 14(3) :215–227, Avril 2003.

[10] P. Alexander et D. Barton. A tutorial introduction to Rosetta. Dans *Hardware Description Languages Conference (HDLCon'01)*, San Jose, Californie, Mars 2001. Présentation disponible : `http ://www.sldl.org/`.

[11] P. Alexander, R. Kamath, et D Barton. System specification in Rosetta. Dans *7th IEEE International Symposium on Engineering of Computer-Based Systems (ECBS 2000)*, pages 299–307. IEEE Computer Society, 2000.

[12] F. Andersen, U. Binau, K. Nyblad, K. D. Petersen, et J. S. Pettersson. The HOL-UNITY verification system. Dans P. D. Mosses, M. Nielsen, et M. I. Schwartzbach, éditeurs, *TAPSOFT'95 : Theory and Practice of Software Development : 6th International Conference : Proceedings*, volume 915 de *Lecture Notes in Computer Science*, pages 795–796. Springer-Verlag, 1995.

[13] C. André. Representation and analysis of reactive behaviors : A synchronous approach. Dans *Proceedings of Computational Engineering in Systems Applications (CESA)*, pages 19–29, Lille, France, Juillet 1996. IEEE-SMC.

[14] C. André. Computing SyncCharts reactions. *Electronic Notes in Theoretical Computer Science*, 88 :3–19, Octobre 2004.

[15] Arvind. BlueSpec : A language for hardware design, simulation, synthesis and verification, invited talk. Dans *First ACM & IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE'03)*, page 249. IEEE Computer Society, 2003.

[16] R.-J. Back et J. von Wright. *Refinement Calculus.* Springer-Verlag, 1998.

[17] R. J. R. Back. On correct refinement of programs. *Journal of Computer and System Sciences*, 23(1) :49–68, 1979.

[18] R. J. R. Back et K. Sere. From action systems to modular systems. Dans M. Naftalin, T. Denvir, et M. Bertran, éditeurs, *FME'94 : Industrial Benefit of Formal Methods*, volume 873 de *Lectures Notes in Computer Science*, pages 1–25. Springer-Verlag, Berlin, Heidelberg, 1994.

[19] R. J. R. Back et J. von Wright. Trace refinement of action systems. Dans B. Jonsson et J. Parrow, éditeurs, *CONCUR'94 : Concurrency Theory /Proceedings of the 5th International Conference*, pages 367–384. Springer-Verlag, Berlin, Heidelberg, 1994.

[20] I. D. Bates, E. G. Chester, et D. J. Kinniment. A statechart based HW/SW codesign system. Dans *CODES '99 : Proceedings of the seventh international workshop on Hardware/software codesign*, pages 162–166, New York, NY, USA, 1999. ACM Press.

[21] R. Beers, R. Ghughal, et M. Aagaard. Applications of hierarchical verification in model checking. Dans T. Margaria et T. F. Melham, éditeurs, *CHARME '01 : Proceedings of the 11th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, volume 2144 de *Lecture Notes in Computer Science*, pages 40–57, London, UK, 2001. Springer-Verlag.

[22] P. Behm, P. Benoit, A. Faivre, et J.-M. Meynadier. METEOR : A successful application of B in a large project. Dans J. M. Wing, J. Woodcock, et J. Davies, éditeurs, *Proceedings of FM'99 : World Congress on Formal Methods*, number 1709 in Lecture Notes in Computer Science (Springer-Verlag), pages 369–387. Springer Verlag, Septembre 1999.

[23] P. Bellows et B. Hutchings. JHDL - an HDL for reconfigurable systems. Dans *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 175–184, Los Alamitos, Californie, 1998. IEEE Computer Society Pres.

[24] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, et R. De Simone. The synchronous languages 12 years later. *Proceedings of IEEE, Special issue on embedded systems*, 91(1) :64–82, Janvier 2003.

[25] J. Bergeron, E. Cerny, E. Hunter, et A. Nightingale. *Verification Methodology Manual for SystemVerilog*, volume 18. Springer-Verlag, 2005.

[26] G. Berry. Esterel on hardware. *Philosophical Transactions of the Royal Society of London*, 339 :87–104, 1992.

[27] G. Berry. The Constructive Semantics of Esterel, 1998. `citeseer.ist.psu.edu/article/berry96constructive.html`.

[28] G. Berry. The foundations of Esterel. Dans G. Plotkin, C. Stirling, et M. Tofte, éditeurs, *Proof, Language, and Interaction : Essays in Honour of Robin Milner*, pages 425–454. MIT Press, 2000.

[29] G. Berry et G. Gonthier. The Esterel synchronous programming language : Design, semantics, implementation. *Science of Computer Programming*, 19(2) :87–152, Novembre 1992.

[30] BlueSpec. BlueSpec product overview. `http ://www.bluespec.com/index.htm`.

[31] R. S. Boyer, M. Kaufmann, et J S. Moore. The Boyer-Moore Theorem Prover and Its Interactive Enhancement. *Computers and Mathematics with Applications*, 5(2) :27–62, 1995.

[32] J. A. Brzozowski. Derivatives of regular expressions. *Journal of the ACM*, 11(4) :481–494, 1964.

[33] A. Bunker, S. A. McKee, et G. Gopalakrishnan. An overview of formal hardware specification languages. Dans *Grace Hopper Celebration of Women in Computing*, Octobre 2002.

[34] L. Cai et D. Gajski. Transaction level modeling : an overview. Dans *CODES+ISSS '03 : Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 19–24, New York, NY, USA, 2003. ACM Press.

[35] D. Cansell, J.-F. Culat, D. Méry, et C. Proch. Derivation of SystemC code from abstract system models. Dans *Forum on specification & Design Languages - FDL'04, Lille, France*, Septembre 2004.

[36] D. Cansell, S. Hallerstede, et Y. Zimmermann. Construction sûre de systèmes électroniques. *Génie Logiciel*, (69) :38–44, Juin 2004.

[37] D. Cansell et D. Méry. Integration of the proof process in the system development through refinement steps. Dans E. Villar, éditeur, *5th Forum on Specification & Design Language - Workshop SFP in FDL'02 , Marseille, France*, Septembre 2002.

[38] D. Cansell, D. Méry, et C. Proch. Modelling SystemC scheduler by refinement. Dans *Conference Proceedings of IEEE ISOLA'05 workshop on Leveraging Applications of Formal Methods, Verification, and Validation*, Septembre 2005.

[39] D. Cansell, D. Méry, et C. Proch. Proved design of hardware architecture. Dans *Poster session, ZB'2005 – Formal Specification and Development in Z and B*, Avril 2005.

[40] D. Cansell, D. Méry, et C. Proch. Un système d'analyse de la qualité : de la norme au produit en passant par le raffinement. *Génie Logiciel*, (73) :44–50, Juin 2005.

[41] D. Cansell, C. Tanougast, Y. Berviller, D. Méry, C. Proch, H. Rabah, et S. Weber. Proof-based design of a microelectronic architecture for mpeg-2 bit-rate measurement. Dans *Forum on specification and Design Languages - FDL'03, Frankfurt, Germany*, Septembre 2003.

[42] Dominique Cansell. *Assistance au développement incrémental et à sa preuve.* Thèse de Doctorat, Université Henri Poincaré (Nancy I), Avril 2003. Habilitation à diriger des recherches.

[43] P. Caspi et M. Pouzet. Synchronous Kahn Networks. Dans *ACM SIGPLAN International Conference on Functional Programming*, Philadelphia, Pensylvania, Mai 1996.

[44] K. M. Chandy et J. Misra. *Parallel Program Design A Foundation.* Addison-Wesley Publishing Company, 1988. ISBN 0-201-05866-9.

[45] M. Chiodo, P. Giusto, Hsieh; H., A. Jurecska, L. Lavagno, et A. Sangiovanni-Vincentelli. A formal methodology for hardware/software codesign of embedded systems, Août 1994. IEEE Micro.

[46] E. M. Clarke, O. Grumberg, et D. A. Peled. *Model Checking.* The MIT Press, 2000.

[47] ClearSy. *Atelier B, Manuel Utilisateur*, 2001. `http ://www.clearsy.com/html/supportsB.htm`.

[48] ClearSy. *Web site B4free set of tools for development of B models*, 2004. `http ://www.b4free.com/index.php`.

[49] E. W. Dijkstra. *A Discipline of Programming.* Prenctice-Hall, 1976.

[50] D. Déharbe, S. Shankar, et E. M. Clarke Jr. Model checking VHDL with CV. Dans *Formal Methods in Circuit Automation Design (FMCAD'98)*, volume 1522 de *Lecture Notes in Computer Science*, pages 508–513. Springer Verlag, 1998.

[51] R. Dömer. *System-level Modeling and Design with the SpecC language.* Thèse de Doctorat, University of Dortmund, 2000.

[52] R. Dömer, J. Zhu, et D. Gajski. The SpecC language reference manual. Rapport technique, Department of Information and Computer Science, University of California, Irvine, 1998.

[53] S. Edwards. Compiling Esterel into sequential code. Dans *Proceedings of the Seventh International Workshop on Hardware/Software Codesign (CODES-99)*, Rome, Italy, Mai 1999. ACM Press.

[54] European Broadcasting Union. implementation guidelines for the use of MPEG-2 systems, video and audio in satellite, cable or terrestrial broadcasting applications. Rapport technique TR 101 154, ETSI, Mai 2000. `http ://www.etsi.org`.

[55] European Broadcasting Union. Digital video broadcasting (DVB)- measurement guidelines for DVB systems. Rapport technique TR 101 290 v1.2.1., ETSI, Mai 2001. `http ://www.etsi.org`.

[56] P. L. Flake et S. J. Davidmann. SuperLog, a unified design language for system-on-chip. Dans *Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 583–586, 2000.

[57] A. Gawanmeh, A. Habibi, et S. Tahar. Enabling SystemC verification using abstract state machines. Dans *Languages for Formal Specification and Verification, Forum on Specification & Design Languages (FDL'04)*, Septembre 2004.

[58] A. Gawanmeh, A. Habibi, et S. Tahar. An executable operational semantics for SystemC using Abstract State Machines. Rapport technique, Concordia University, Department of Electrical and Computer Engineering, Mars 2005.

[59] U. Glässer, E. Börger, et W. Müller. The semantics of behavioral VHDL'93 descriptions. Dans *Proceedings of EURO-VHDL'94*, 1994.

[60] U. Glässer, E. Börger, et W. Müller. Formal definition of an abstract VHDL'93 simulator by EA-machines. Dans C. Delgado Kloos et P. T. Breuer, éditeurs, *Formal Semantics for VHDL*. Kluwer Academic Publishers, 1995.

[61] G. Gonthier. *Sémantique et modèles d'exécution des langages réactifs synchrones : application à Esterel*. Thèse de Doctorat, Université d'Orsay, Paris, France, Mars 1988.

[62] M. Gordon, J. Iyoda, S. Owens, et K. Slind. Automatic formal sunthesis of hardware from higher order logic. Dans *Proceedings of the 4th International Workshop on Verification of Critical Systems (AVOCS 04)*, 2005.

[63] Hardware Verification Group. *Hands-on Manual to FormalCheck, Version 2.3*. Concordia University, Montreal, Quebec, Canada, Mai 2000.

[64] A. Habibi et S. Tahar. A survey : System-on-a-chip design and verification. Rapport technique, Concordia University, Department of Electrical and Computer Engineering, Janvier 2003.

[65] A. Habibi et S. Tahar. SystemC fixpoint semantics. Rapport technique, Concordia University, Department of Electrical and Computer Engineering, Janvier 2005.

[66] N. Halbwachs, P. Bournai, et C. Ratel. Programming and verifying real-time systems by means of the synchronous data-flow programming language lustre. Dans *IEEE Transactions on Software Engineering, Special Issue on the Specification and Analysis of Real-Time Systems*, Septembre 1992.

[67] N. Halbwachs, P. Caspi, P. Raymond, et D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9) :1305–1320, Septembre 1991.

[68] N. Halbwachs, F. Lagnier, et P. Raymond. Synchronous observers and the verification of reactive systems. Dans M. Nivat, C. Rattray, T. Rus, et G. Scollo, éditeurs, *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93*, Workshops in Computing, pages 83–96. Springer, Juin 1993.

[69] S. Hallerstede et Y. Zimmermann. Circuit design by refinement in EventB. Dans *Forum on Specification and Design Languages - FDL'04, Lille, France*. Pierre Boulet, Septembre 2004.

[70] D. Harel. Statecharts : A visual formalism for complex systems. Dans *Sci. Comput. Program.*, volume 8, pages 231–274, Juin 1987.

[71] T. S. Hoang, Z. Jin, K. Robinson, A. McIver, et C. Morgan. Development via refinement in Probabilistic B - foundation and case study. Dans H. Treharne, S. King, M. C. Henson, et S. Schneider, éditeurs, *ZB05*, volume 3455 de *Lecture Notes in Computer Science*, pages 355–373. Springer, 2005.

[72] C. A. R. Hoare. Communicating sequential processes. *Comm. of the ACM*, 21(8), 1978.

[73] S. Holmström et K. Sere. Reconfigurable hardware - a case study in codesign. Dans *FPL : From FPGAs to Computing Paradigm*, volume 1482 de *Lectures Notes in Computer Science*, pages 451–455. Springer-Verlag, 1998.

[74] Synopsys Inc. Describing synthesizable RTL in SystemC. Rapport technique, Synopsys, Novembre 2002.

[75] K. M. Kahn et V. A. Saraswat. Complete visualizations of concurrent programs and their executions. Dans *Proceedings of the IEEE Workshop 1990 on Visual Languages*, pages 7–15, Skokie, IL, 1990.

[76] R. Kaivola et N. Narasimhan. Formal verification of the pentium® 4 floating-point multiplier. Dans *DATE '02 : Proceedings of the conference on Design, automation and test in Europe*, page 20, Washington, DC, USA, 2002. IEEE Computer Society.

[77] D.C. Ku et G. DeMicheli. HardwareC - a language for hardware design. Rapport technique, Computer Systems Lab, Stanford University, Août 1998.

[78] P. Le Guernic, T. Gautier, M. Le Borgne, et C. Lemaire. Programming real-time applications with signal. *Proceedings of the IEEE*, 79(9) :1321–1336, Septembre 1991.

[79] P. Le Guernic, J.-P. Talpin, et J.-C. Le Lann. Polychrony for system design. *Journal of Circuits, Systems, and Computers - Special Issue : Application Specific Hardware Design*, 12(3) :261–303, Décembre 2003.

[80] N. Lopez, M Simonot, et V. Viguie Donzeau-Gouge. Deriving software specifications from event based models. Dans *ZB'2000 - International Conference of B and Z Users*, volume 1878 de *Lecture Notes in Computer Science*, pages 209–229, Helsington, York, UK YO10 5DD, 2000. Springer-Verlag.

[81] D. Méry, D. Cansell, C. Proch, D. Abraham, et P. Ditsch. The challenge of QoS for digital television services. *EBU Technical Review*, Avril 2005.

[82] J S. Moore, T. Lynch, et M. Kaufmann. A Mechanically Checked Proof of the AMD5$_K$86 Floating-Point Division Program. *IEEE Trans. Comp.*, 47(9) :913–926, Septembre 1998. See also URL http ://devil.ece.utexas.edu/lynch/divide/divide.html.

[83] C. Morgan. *Programming from specifications.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[84] MOSQUITO. Management Of the Service QUality in Television Operations. RTD in Advanced Communications Technologies and Services (ACTS). Project number : AC334.

[85] M. Moy, F. Maraninchi, et L. Maillet-Contoz. LusSy : A toolbox for the analysis of systems-on-a-chip at the transactional level. Dans *International Conference on Application of Concurrency to System Design*, Juin 2005.

[86] M. Moy, F. Maraninchi, et L. Maillet-Contoz. Pinapa : An extraction tool for SystemC descriptions of Systems-on-a-Chip. Dans *ACM International Conference on Embedded Software (EMSOFT'05)*, Jersey city, New Jersey, USA, Septembre 2005.

[87] W. Mueller, R. Dömer, et A. Gerstlauer. The formal execution semantics of SpecC. Dans *ISSS '02 : Proceedings of the 15th international symposium on System Synthesis*, pages 150–155, New York, NY, USA, 2002. ACM Press.

[88] Open SystemC Initiative. *SystemC 2.0.1 Language Reference Manual*, 2004. `http ://www.systemc.org`.

[89] S. Pasricha. Transaction level modeling of SoC in SystemC 2.0. Rapport technique, STMicroelectronics Ltd, 2002.

[90] J. Plosila et K Sere. Action systems in pipelined processor design. Dans *3rd International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC '97), 7-10 April 1997, Eindhoven, The Netherlands*, pages 156–166. IEEE Computer Society, 1997.

[91] D. Potop-Butucaru et R. de Simone. Optimizations for Faster Execution of Esterel Programs. Dans *Proceedings of First ACM & IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE'03)*, page 227. IEEE Computer Society, 2003.

[92] C. Proch, D. Cansell, et D. Mery. Projet RNRT EQUAST ; SP2 spécification incrémentale du système. Rapport de recherche, LORIA, INRIA, Octobre 2004.

[93] QUOVADIS. QUality Of Video and Audio in DIgital tv Service. RTD in Advanced Communications Technologies and Services (ACTS). Project number : AC056.

[94] S. Ramesh. Efficient translation of Statecharts to hardware circuits. Dans *12th International Conference on VLSI Design - 'VLSI for the Information Appliance'*, pages 384–389. IEEE Computer Society, 1999.

[95] R. Reetz et T. Kropf. A flow graph semantics of VHDL : A basis for hardware verification with VHDL. Dans C. D. Kloos et P. T. Breuer, éditeurs, *Formal Semantics for VHDL*, pages 205–238. Kluwer, 1995.

[96] J. Ruf. RAVEN : Real-Time Analyzing and Verification Environment. *Journal of Universal Computer Science*, 7(1) :89–104, Janvier 2001.

[97] J. Ruf, D. Hoffmann, J. Gerlach, T. Kropf, W. Rosenstiel, et W. Mueller. The simulation semantics of SystemC. Dans *DATE '01 : Proceedings of the conference on Design, automation and test in Europe*, pages 64–70, Piscataway, NJ, USA, 2001. IEEE Press.

[98] J. Ruf, D. Hoffmann, T. Kropf, et W. Rosenstiel. Simulation-guided property checking based on a multi-valued AR-automata. Dans *DATE '01 : Proceedings of the conference on Design, automation and test in Europe*, pages 742–748, Piscataway, NJ, USA, 2001. IEEE Press.

[99] J. Ruf, D. Hoffmann, et W. Rosenstiel. Simulation based validation of FLTL formulas in executable system descriptions. Dans *In Forum on Design Languages (FDL 2000)*, 2000.

[100] A. Salem. Formal semantics of synchronous SystemC. Dans *DATE '03 : Proceedings of the conference on Design, Automation and Test in Europe*, pages 376–381, Washington, DC, USA, 2003. IEEE Computer Society.

[101] A. Sangiovanni-Vincentelli, M. Sgroi, et L. Lavagno. Formal models for communication-based design. Dans C. Palamidessi, éditeur, *Proceedings of CONCUR 2000 - Concurrency Theory, 11th International Conference*, volume 1877 de *Lecture Notes in Computer Science*, pages 29–47. Springer-Verlag, Août 2000.

[102] C. Seger et R. Bryant. Formal verification by symbolic evaluation of partially-ordered trajectories. *Formal Methods in System Design*, 6(2) :147–189, Avril 1994.

[103] K. Sere. Stepwise refinement of reactive processor farms. Dans B. Jonsson, J. Parrow, et B. Pehrson, éditeurs, *Protocol Specification, Testing and Verification*, pages 109–124, 1991.

[104] K. Sere. Procedures and atomicity refinement. *Information Processing Letters*, 60(2) :67–74, 1996.

[105] IEEE Std1076-1993. *Standard VHDL Langage Reference Manual.*

[106] IEEE Std1364. *Standard Verilog Langage Reference Manual.*

[107] IEEE Std315-1975. *Graphic Symbols for Electrical and Electronic Diagrams*, 1975.

[108] Synopsys Inc. CoCentric System Studio.
`http ://www.synopsys.com/products/cocentric_studio/cocentric_studio.html`.

[109] J. P. Van Tassel. A formalisation of the VHDL simulation cycle. Dans L. J. M. Claesen et M. J. C. Gordon, éditeurs, *Higher Order Logic Theorem Proving and its Applications : Proceedings of the IFIP TC10/WG10.2 International Workshop, Leuven, September 1992*, IFIP Transactions A-20, pages 359–374. North-Holland, 1993.

[110] Z. Zhou, Z. Lin, et W. Cao. Research on VHDL RTL synthesis system. Dans *The First IEEE International Workshop on Electronic Design, Test and Applications (DELTA '02)*, page 99. IEEE Press, 2002.

[111] Y. Zimmermann et T. Toma. Component reuse in B using ACL2. Dans *ZB 2005 : Formal Specification and Development in Z and B, 4th International Conference of B and Z Users, Guildford, UK, April 13-15, 2005*, pages 279–298, 2005.

Glossaire

Le présent glossaire référence les principaux acronymes électroniques utilisés dans ce document ainsi que certains termes techniques employés dans le monde de la télévision numérique.

ASIC : *Application-Specific Integrated Circuits*, circuits extrêmement performants car spécialisés dans la réalisation d'une tâche unique.

Continuity counter : Champ de l'en-tête d'un paquet PID permettant de déterminer l'ordre des paquets d'un PID donné.

DVB-T : *Digital Video Broadcasting Television.*

FPGA : *Field-Programmable Gate Array*, circuits configurables voire reconfigurables permettant des performances moyennes.

HDL : *Hardware Description Language*, famille de langages de description de circuits électroniques.

netlist : Liste et réseau de portes logiques permettant la gravure d'un circuit.

Paquet TS : paquets *Transport Stream*, les données DVB-T sont encapsulées dans des paquets permettant le transport.

Paramètres QdS ou de Synthèse : Paramètres proposés dans la norme TR 101 290, compositions de paramètres basiques, permettant un diagnostique simplifié des problèmes.

PAT : *Program Association Table*, paquet au PID réservé permettant le transport des tables de référencement des PIDs.

PCR : *Program Clock Reference*, référence temporelle contenue dans l'en-tête des paquets TS.

PES : *Packetized Elementary Stream*, paquets (en-tête+données) encapsulés dans les paquets TS. Structure de base permettant le transport du flux de données.

PID : *Packet IDentifier*, numéro d'appartenance d'un paquet TS.

PMT : *Program Map Table*, paquet permettant le transport des tables de référencement des PIDs.

QdS : Qualité de Service.

RTL : *Register Transfer Level*, niveau de description de circuits utilisant des registres et permettant, à l'aide d'un outil dédié, la production d'une netlist.

SoC : Abbréviation de *System on Chip*, système enfoui.

System on Chip : Système enfoui.

T-STD : Modèle de décodeur virtuel préconisé par la norme afin de vérifier la décodabilité du signal reçu par les récepteurs.

TLM : *Transaction Level Modelling*, niveau de description de systèmes enfouis essentiellement centré sur la communication entre processus.

TS_sync_loss : Paramètre indiquant l'état de synchronisation d'un récepteur.

VHDL : *Very high speed integrated circuits Hardware Description Language*, langage de description de circuits.

Annexe A

Algorithme de synchronisation (pseudo-code C)

L'algorithme suivant n'est qu'une traduction en pseudo-code C du modèle B contenant l'unique événement prog issu de la recomposition des différents événements du modèle.

```
/* constantes du système (environnement et autres) */
BYTE SyncByte;
tab[BYTE] flux;

/* initialisation */
TS_sync_loss = IND;
countOK = 0;
countKO = 0;
pointer = 0;
b = true;

/* algo */

while (true) {

 if b {
   if TS_Sync_loss != OK {
    if Flux(pointer) != SyncByte {
      countKO = countKO+1; // evenement readToDeSynchroKO
      countOK = 0;
      b = false;
    } elseif countOK +1 >= bsync {
      TS_Sync_loss = OK;  // evenement ToSync
      countOK = countOK+1;
      countKO = 0;
      b = false;
    } else {
      countOK = countOK+1;  // evenement readToSynchroOK
      b = false;
    }
   } else {
    if Flux(pointer) == SyncByte {
      countOK = countOK+1; // evenement readToDeSynchroOK
      countKO = 0;
```

```
      b = false;
    } elseif countKO +1 >= bdesync {
      TS_Sync_loss = KO;  // evenement ToDesync
      countKO = countKO+1;
      countOK = 0;
      b = false;
    } else {
      countKO = countKO+1; // evenement readToDeSynchroKO
      b = false;
    }
   }
 } else {
  if TS_Sync_loss == OK {
    pointer = pointer+TSsize; // evenement ProgressSynchro
    b = true;
  } elsif Flux(pointer) == SyncByte {
    pointer = pointer+TSsize; // evenement progressKODesynchro
    b = true;
  } else {
    pointer = pointer+1; // evenement progressOKDesynchro
    b = true;
  }
 }
}
```

Annexe B

Exemples de code SystemC

B.1 Un module SystemC type

Le code SystemC suivant présente l'exemple traditionnelle de la description d'une porte logique AND. Dans ce cas, la porte logique est décrite sous la forme d'un module SystemC à deux entrées (ports) et une sortie. Le code suivant est le contenu du fichier `and.h` qui décrit le module réalisant la porte logique. Ce fichier se contente de décrire les attributs et les profils des méthodes du module.

```
#include <systemc.h>

SC_MODULE(and) {

  /* Declaration des ports d'entree du module */
  sc_in<bool> in1;
  sc_in<bool> in2;

/* Declaration du port de sortie du module contenant
   le resultat du ET logique */
  sc_out<bool> result;


  void treatments();

  SC_CTOR(and) {

    SC_METHOD(treatments);
    /* Liste de sensibilite d'execution de la methode treatments */
    sensitive << in1;
    sensitive << in2;
  }
};
```

Le code suivant, contenu dans le fichier `and.cpp` est la description des méthodes du module and.

```
#include "and.h"

void and::treatments() {

  result.write(in1.read() & in2.read());
}
```

B.2 Un programme SystemC complet

Le listing ci-dessous présente un exemple de point d'entrée d'un programme SystemC. Le programme ci-dessous interconnecte, par l'intermédiaire de signaux, différents modules afin de réaliser une architecture complète. Le code suivant est issu d'un fichier `main.cpp`, point d'entrée du programme.

```
// inclusion des bibliotheques requises
#include <systemc.h>
#include <stdio.h>
#include <stdlib.h>
#include "lecture.h"
#include "RESET.h"
#include "Synchroni.h"
#include "PATMod.h"
#include "PIDMod.h"
#include "continuity.h"

int sc_main(int argc, char *argv[]) {

  int i;
  // horloge du èsystme
  sc_clock clock("clock", 1, 0.5, 0.0);

  // édclaration des signaux (variables locales)
  sc_signal<bool> s_lect;
  sc_signal<bool> s_flagSynchro;
  sc_signal<bool> s_errorSynchro;
  sc_signal<bool> s_flagCont;
  sc_signal<bool> s_errorCont;
  sc_signal<bool> s_flagPAT;
  sc_signal<bool> s_errorPAT;
  sc_signal<bool> s_flagPID;
  sc_signal<bool> s_errorPID;
  sc_signal<bool> s_reset;
  sc_signal<bool> s_start;
  sc_signal<int> s_PAT_PID;
  sc_signal<int> s_pid;
  sc_signal<int> s_conti;
  sc_signal<int> s_synchro;

  // initialisation des signaux
  s_lect.write(false);
  s_flagSynchro.write(false);
  s_errorSynchro.write(false);
  s_flagCont.write(false);
  s_errorCont.write(false);
  s_flagPAT.write(false);
  s_errorPAT.write(false);
  s_flagPID.write(false);
  s_errorPID.write(false);
  s_reset.write(false);
  s_start.write(false);
  s_pid.write(0);
  s_PAT_PID.write(0);
```

```
s_synchro.write(0);
s_conti.write(0);

LECTURE lect("lect");
lect.ready(s_start);
lect.flag(s_lect);
lect.synchro(s_synchro);
lect.pid(s_pid);
lect.conti(s_conti);
lect.reset(s_reset);


// instanciation des objets modules
PATMod patmod("PATMod");
patmod.reset(s_reset);
patmod.inSync(s_flagSynchro);
patmod.errorSync(s_errorSynchro);
patmod.PID(s_pid);
patmod.data(s_conti);
patmod.sendPID(s_PAT_PID);
patmod.flag(s_flagPAT);
patmod.error(s_errorPAT);
patmod.localPID = 0;

SYNCHRO syn("Synchro");
syn.reset(s_reset);
syn.ready(s_lect);
syn.data(s_synchro);
syn.flag(s_flagSynchro);
syn.error(s_errorSynchro);

PIDMod pidmod("PIDMod");
pidmod.reset(s_reset);
pidmod.inSync(s_flagSynchro);
pidmod.errorSync(s_errorSynchro);
pidmod.PID(s_pid);
pidmod.localPID(s_PAT_PID);
pidmod.flag(s_flagPID);
pidmod.error(s_errorPID);

continuity co("continuity");
co.reset(s_reset);
co.inPID(s_flagPID);
co.errorPID(s_errorPID);
co.data(s_conti);
co.flag(s_flagCont);
co.error(s_errorCont);
co.local = 0;


RESET reset("RESET");
reset.clock(clock);
reset.flag(s_flagPID);
reset.error(s_errorPID);
```

```
    reset.flagPAT(s_flagPAT);
    reset.errorPAT(s_errorPAT);
    reset.reset(s_reset);
    reset.ready(s_start);

    // intialisation du èsystme
    sc_initialize();

    // Lancement de la simulation par le scheduler
    sc_start();

    printf("**************************\n");
    printf("*      FIN TRAITEMENT      *\n");
    printf("**************************\n\n");

    return 0;
}
```

Monsieur **PROCH Cyril**

DOCTORAT DE L'UNIVERSITE HENRI POINCARE, NANCY 1

En INFORMATIQUE

VU, APPROUVÉ ET PERMIS D'IMPRIMER N°1177

Nancy, le 3 avril 2006

Le Président de l'Université

UNIVERSITÉ HENRI POINCARÉ, NANCY 1 LE PRESIDENT

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