N° d'ordre: 3273

THÈSE

Présentée devant devant l' **Université de Rennes 1** pour obtenir

le grade de : Docteur de l'Université de Rennes 1
Mention Informatique

par

David BERNER

Équipe d'accueil : ESPRESSO
École Doctorale : Matisse
Composante universitaire : IFSIC / IRISA

Titre de la thèse :

Utilisation de méthodes formelles dans la conception conjointe de systèmes embarqués

Using Formal Methods for the CoDesign of Embedded Systems

soutenue le 06 mars 2006 devant la commission d'examen

Patrice	Quinton	Président
Florence	Maraninchi	Rapporteur
Sandeep	Shukla	Rapporteur
Robert	de Simone	Examinateur
Paul	Le Guernic	Examinateur
Jean-Pierre	Talpin	Directeur de thèse

Why waste time learning, when ignorance is instantaneous?
Calvin & Hobbes

Contents

First Part – French Abstract **7**

Introduction **9**

Contributions principales 11

1 Flot de conception conjointe **13**

1.1 L'approche modulaire 14

1.2 Réutiliser c'est accélérer 16

2 Pourquoi utiliser des méthodes formelles dans la conception conjointe? **19**

2.1 Assurer que la spécification soit complète 19

2.2 Réduire le nombre d'erreurs dans la spécification 20

2.3 Réduire le nombre d'erreurs dans l'implémentation 20

2.4 Accélérer le développement, réduire les coûts 20

2.5 Améliorer la fiabilité du système 21

2.6 Prouver le respect de standards 21

3 L'intégration de méthodes formelles dans la conception conjointe **23**

3.1 Transformation automatisée vers un langage formel 23

3.2 La modélisation formelle itérative 24

3.3 Validation de protocoles insensibles aux latences 25

4 Mise en oeuvre **27**

4.1 Transformation de composants SystemC en SIGNAL 27

4.2 Extraction d'informations structurelles de modèles SystemC . . . 29

4.3 Modélisation formelle itérative 30

Conclusion **33**

Second Part – English Abstract **37**

Introduction **39**
Main Contributions 42

5 Embedded System CoDesign Flow **43**
5.1 The Basic Design Flow 43
5.1.1 Specification 43
5.1.2 Implementation 45
5.1.3 Synthesis 46
5.2 The Modular Approach 47
5.3 To Reuse Means to Accelerate 49

6 Why use Formal Methods for CoDesign? **53**
6.1 Assure Completeness of Specification 54
6.2 Reduce Specification Errors 54
6.3 Reduce Errors in Implementation 55
6.4 Speed up Development, Reduce Costs 55
6.5 Improve System Reliability 55
6.6 Prove the Adherence to Standards 56

7 Integration of Formal Methods into the CoDesign Flow **57**
7.1 Automated Transformation into a Formal Language 57
7.2 Extreme Formal Modeling 58
7.3 Validation of Latency Insensitive Protocols 59

Third Part – Formal Foundations **61**

8 Polychrony and SIGNAL **63**
8.1 An Algebraic Notation 64
8.1.1 Formal Syntax 66
8.1.2 Notational Conventions 67
8.2 A Polychronous Model of Computation 68
8.2.1 Scheduling Structure of Polychrony 69
8.2.2 Synchronous Structure of Polychrony 69
8.2.3 Denotational Semantics of the iSTS Algebra 70
8.3 The SIGNAL Language 72
8.3.1 Relating Polychronous Signals with Clocks 73
8.3.2 Code Generation via Hierarchization 74
8.3.3 Some More Concrete Syntax 75
8.4 Translating iSTS into SIGNAL 77

9 Behavioral Types for SystemC **79**
9.1 Example and Overview . . . 80
9.1.1 Static Single Assignment . . . 80
9.1.2 Propositional Behavior . . . 81
9.1.3 Static Abstraction . . . 83
9.1.4 Typed Modules . . . 83
9.1.5 Proof Obligations . . . 85
9.2 Formal Syntax of the SystemC Core . . . 85
9.3 Inference . . . 88
9.3.1 Completion of the State Logic . . . 91
9.3.2 Modular Extension to External Method Calls . . . 91
9.3.3 Static Interface of SystemC Modules . . . 92
9.4 A Behavioral Module System . . . 93
9.4.1 Type Inference for Declarations . . . 94
9.4.2 Type Inference for Modules . . . 94
9.4.3 Proof Obligation Synthesis . . . 95
9.5 Behavioral Types in Polychrony . . . 96

10 Applications **99**
10.1 Scalability . . . 99
10.2 Modularity . . . 100
10.3 Design Checking . . . 101
10.4 Design Exploration . . . 102
10.5 Systematic Formalization of Specification-Level Behavior . . . 103
10.6 Conformance Checking . . . 104

Fourth Part – Using Formal Methods for Embedded Systems **105**

11 Introduction **107**
11.1 Translating SystemC Behavior using SSA . . . 109
11.2 SytemCXML . . . 110
11.3 Contributions . . . 111
11.4 Related Work . . . 111
11.4.1 Ptolemy . . . 112
11.4.2 POLIS, Metropolis . . . 112
11.4.3 Existing Tools for Structural Reflection . . . 113
11.4.4 ESys.NET Framework . . . 114
11.4.5 BALBOA Framework . . . 114
11.4.6 Pinapa and LusSy . . . 114

11.4.7 Java, C# .NET Framework, C++ RTTI 115
11.4.8 Doxygen, XML, Apache's Xerces-C++ 115

12 Modular Verification of SystemC Components 117
12.1 Methodology and Tools 118
12.1.1 Static Single Assignment Form and GIMPLE 118
12.1.2 Formal Verification of Component Properties 119
12.2 Case Study of an FIR Filter 120
12.2.1 The SystemC Model 120
12.2.2 Obtaining a GIMPLE-SSA Representation 121
12.2.3 Extracting Clock and Scheduling Information 122
12.2.4 The Equivalent SIGNAL Program 124
12.2.5 Making a Boolean Model 127
12.2.6 Using the Model Checker 127
12.2.7 Abstraction of the SIGNAL Model 128
12.3 Summary 129

13 Automated Extraction of Structural Information from SystemC-based IP 131
13.1 Extracting Structural Information 132
13.2 Applications for Validation 137
13.2.1 Visualization 137
13.2.1.1 The DOT format. 138
13.2.1.2 Graph generation. 138
13.2.2 Design Management 140
13.2.3 Automated Test Generation 141
13.3 Current Work 144
13.4 Summary 144

14 Incrementally Building Formal Models 147
14.1 Introduction 147
14.2 Contributions 149
14.3 Incremental Model Building and Polychrony 150
14.3.1 Nondeterminism 151
14.3.2 Verification Logic 151
14.3.3 Compositionality 152
14.4 Methodology 152
14.4.1 Tools 153
14.4.2 Extreme Programming 154
14.5 Example 154
14.5.1 Traffic light model. 155

14.5.2 Model of a DLX Pipeline Control 156
14.6 Subsequent Work 160
14.6.1 Property Ordering 161
14.6.2 GUI Toolkit 162
14.7 Summary 162

15 Validating Latency Insensitive Protocols 165
15.1 Introduction 165
15.1.1 Related Work 166
15.1.2 Spin 168
15.2 Contributions 168
15.3 Methodology 168
15.3.1 Carloni's Latency Insensitive Protocol 168
15.3.2 Eliminating Relay Stations 170
15.3.3 Components with Different Clocks 172
15.3.4 Verification of LI Protocols 173
15.4 Subsequent Work 174
15.5 Summary 175

Conclusion 177
Future Work 179

References 182

List of Figures 195

First Part

French Abstract

Introduction

L'étude de l'évolution des méthodes industrielles de haute technologie fait ressortir deux tendances majeures. L'une se traduit, du fait d'une mondialisation croissante, par l'abandon de la conception d'un produit en un lieu unique. Aujourd'hui, de nombreuses équipes dispersées dans des laboratoires et des pays différents travaillent ensemble pour la réalisation d'un seul produit. L'exemple le plus représentatif de ce phénomène est la construction de l'airbus A380 qui a fait coopérer un grand nombre de scientifiques. L'autre se traduit par une miniaturisation des transistors en silicium qui permet d'assembler des centaines de millions de transistors sur une seule puce - 267 millions par exemple pour le processeur Power5 de IBM en 2004. La Semiconductor Industry Association (SIA) prédit qu'en 2020 le nombre de transistors sur une seule puce de haute performance dépassera les 22 milliards (Figure 1). La taille, et donc la complexité des systèmes, est en croissance exponentielle [Nai02], pourtant la concurrence et les habitudes de consommation exigent que les temps de mise à la vente décroissent. Des projets nécessitant 24 mois de travail en 1994 doivent en 2004 être accomplis en 12 mois seulement [LW98, SR98].

En observant ces évolutions on peut se demander comment les entreprises vont pouvoir répondre à ces défis. Comment des équipes de plus en plus distribuées vont trouver la capacité de développer des produits de plus en plus complexes en un temps plus court?

Une conséquence observée aujourd'hui est la dégradation de la qualité des produits. En effet, beaucoup de produits électroniques sont mis sur le marché alors qu'ils ne fonctionnent pas aussi bien qu'ils le devraient, notamment dans le secteur des innovations. Après l'achat d'un produit il est souvent possible de récupérer des mises à jour chez le constructeur qui corrigent des bogues ou rajoutent des fonctionnalités au produit. Ce phénomène est très courant pour de petits objets comme les téléphones portables et les graveurs DVD, mais plus surprenant, on le trouve également sur les nouvelles voitures. Il apparaît alors ici comme une évidence qu'on ne maîtrise plus la complexité des systèmes.

Pour lutter contre cette dérive et afin d'apporter une réponse plus satisfaisante au problème, il serait souhaitable d'augmenter la productivité des développeurs. Comme nous allons le voir, ceci peut être obtenu par l'utilisation (i) de niveaux

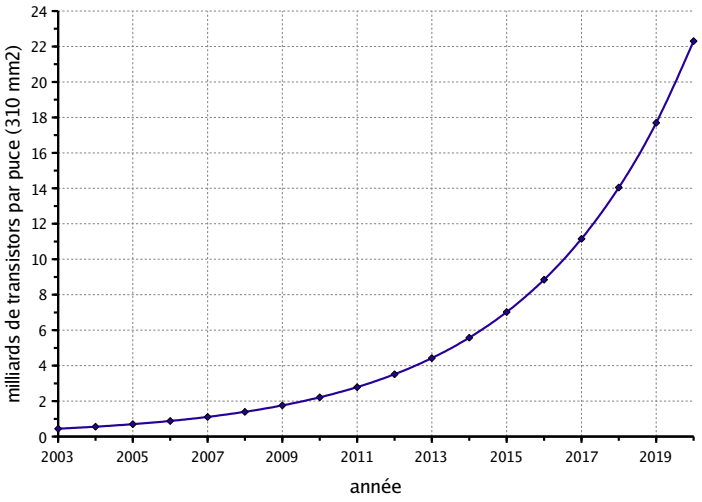


Figure 1: Prédictions de la SIA en 2005 pour le nombre de transistors dans des puces de haute performance [Ass06]

de description plus élevés, (ii) de modules existant prouvés, (iii) de méthodes formelles et (iv) d'outils innovants et performants qui intègrent intelligemment toutes ces fonctionnalités.

Aujourd'hui, le RTL (Register Transfer Level) est utilisé pour concevoir la plupart des circuits électroniques. Cette technique ne sera simplement plus utilisable d'ici quelques années car la complexité des systèmes ne sera plus abordable en RTL. Une évolution inévitable est de passer à un niveau de description plus élevé comme le TLD (Transaction Level Design) et le SLD (System Level Design) afin de décrire les mêmes fonctionnalités de manière plus concise.

La réutilisation des composants (prouvés dans des designs antérieurs, ou achetés) permettrait également de gagner en productivité. La difficulté est ici de trouver la bonne description pour échanger ces informations afin de faciliter la réutilisation d'un module par un maximum de projets différents. Elle doit être suffisamment générique pour pouvoir être intégrée dans les simulateurs de haut niveau, mais aussi suffisamment précise pour une utilisation au niveau physique, tout en restant indépendante d'un processus de production spécifique. Au fait que ces exigences semblent déjà impossible à satisfaire, il faut ajouter un fait important qui oppose les vendeurs de propriétés intellectuelles. Ils sont attentifs à protéger leur savoir-faire technologique et ne souhaitent donc pas révéler les détails de l'implémentation. Les développeurs d'autre part, ont besoin d'un descriptif détaillé afin de modéliser un environnement convenable.

Une erreur de conception dans un système est certes très coûteuse, mais elle l'est encore plus si elle est repérée très tard dans le processus de développement.

C'est pour éviter ce type de perte de productivité qu'il est nécessaire de veiller à ce que la qualité de construction soit maximale. Simuler une spécification ou une description ne suffit pas. En effet, cela ne révèle qu'un sous-ensemble limité d'erreurs qui sont fortement dépendantes du type de tests et des vecteurs d'entrée. Au contraire, les méthodes formelles [Win90] offrent des possibilités bien plus avancées pour réduire de manière drastique le nombre d'erreurs. Cependant, pour chaque étape de la conception d'une application on compte un nombre important de méthodes différentes et il est donc demandé de choisir la bonne. Aussi on comprend qu'elles soient réputées difficiles à utiliser et réputées inadaptées aux applications réelles par leur lenteur et leurs besoins en ressources. [Hal90, BH95a, BH95b].

Pour terminer, améliorer la qualité des outils est indispensable pour réaliser ce bond en avant en productivité. Ces nouveaux outils de conception doivent couvrir l'ensemble de la conception du système, de la spécification de besoins jusqu'à l'implémentation au niveau physique. L'idéal serait d'avoir un outil qui transforme automatiquement une description comportementale en une description de niveau RTL. Mais aujourd'hui, il n'est pas envisageable d'obtenir des résultats optimisés avec ce type d'outil. Cependant il est possible de faire un premier pas dans ce sens en effectuant un raffinement guidé. L'outil prend en charge toutes les transformations triviales, et laisse à l'utilisateur le soin de fournir des paramètres et de prendre des décisions de conception importantes. Il doit avoir la possibilité d'intégrer correctement des composants de propriété intellectuelle (IP) de bibliothèques prédéfinies pour remplacer des modules comportementaux par des modules détaillés. Le défi dans la création de systèmes embarqués consiste donc en la mise à disposition de techniques qui indépendamment l'une de l'autre améliorent et accélèrent tous les aspects de la conception, tout en réalisant des environnements capables d'intégrer ces techniques de façon cohérente afin de former un cadre sans rupture sémantique. Ces outils devront trouver un équilibre entre l'automatisation d'un maximum de tâches et le contrôle incontesté de l'implémentation par le développeur.

Contributions principales

Pour représenter le comportement d'un programme par un type ou une proposition, on utilise l'algèbre STS comme proposé dans [TG04] et inspiré des systèmes de transitions synchrones de Pnueli [PSS98]. On définit une syntaxe formelle du noyau SystemC et on l'utilise pour créer une méthodologie et un système de types comportementaux. On montre comment on peut inférer des types formels et comportementaux depuis des modules SystemC existants et comment ces types peuvent être exprimés dans Polychrony (chapitre 9).

L'étude de cas sur le filtre FIR dans chapitre 12 met cette théorie en pratique. Elle est validée et il est montré en détail comment une telle extraction de types peut marcher. Les outils nécessaires pour cette procédure sont identifiés et mis en oeuvre.

Cependant ce système de typage ne permet pas de conserver l'information structurelle d'un design. Dans la perspective de pouvoir traiter des systèmes avec de nombreux modules sans perdre l'information structurelle, un parser/-analyseur SystemC a été conçu et implémenté dans un prototype qui est capable d'extraire les informations structurelles. A partir de ces informations structurelles, une représentation intermédiaire est créée ainsi qu'une représentation XML (chapitre 13). Depuis cette structure interne, un squelette SIGNAL est généré représentant la structure originale du code SystemC. Une méthodologie est établie qui intègre la génération de types comportementaux SystemC modulaires avec l'extraction d'informations structurelles.

L'utilité des informations structurelles des systèmes embarqués en général pour d'autres applications est illustré par un exemple de visualisation et un environnement de génération automatique de tests. Le logiciel SystemCXML a été publié en tant que projet open source et hébergé sur Sourceforge [BMPS04].

Lors d'un travail avec Syed Suhaib, une méthodologie a été établie pour la création de modèles formels de façon incrémentale avec des techniques empruntées de la programmation extrême. Le but de cette méthodologie est de baisser le niveau de difficulté de la création de modèles formels depuis des spécifications en langage parlé et de créer des modèles formels correct par construction (chapitre 14). En contraste avec l'approche de types comportementaux, ces travaux montrent comment des méthodes formelles peuvent être utilisées plus tôt dans un flot de conception afin d'obtenir une qualité de modèle supérieure.

Les limites des communications synchrones de systèmes à grande échelle peuvent être repoussées en utilisant des protocoles insensibles à la latence. Le chapitre 15 montre comment de tels protocoles peuvent être vérifiés formellement et propose un protocole modifié qui améliore certains aspects sans faire de compromis au niveau performance.

Chapter 1

Flot de conception conjointe

Pour des systèmes d'une complexité limitée, on développe un modèle de haut niveau à partir d'une spécification textuelle du système qui comprend toutes les fonctionnalités des parties matérielles et logicielles. Figure 1.1 montre les étapes principales de la conception conjointe. Le modèle au niveau système permet d'analyser le comportement global du système. On peut simuler le modèle avec des vecteurs d'entrée et comparer les sorties avec ce qu'on attend d'après la spécification. Ce niveau manque des détails d'implémentation comme l'architecture des processeurs ou les types de bus utilisés pour la communication. Donc sa simulation ne permet pas d'obtenir des résultats pertinents au niveau performance et timing, par contre il permet de vérifier le comportement fonctionnel.

Après la vérification fonctionnelle du modèle au niveau système, on passe à la phase implémentation du système. On rajoute des détails d'architecture, de communication, et d'ordonnancement, et on répartit le système dans une partie logicielle et une partie matérielle. À la fin de la phase d'implémentation on obtient la partie matérielle au niveau RTL et la partie logicielle dans un code exécutable (e.g. C ou C++). La validation et vérification de la fonctionnalité du modèle devrait bien sur être faite à tous les niveaux du flot de conception. Puisque le niveau RTL est le dernier maillon d'une chaîne d'opérations plutôt manuelles, la majorité des tests et simulations est traditionnellement faite à ce niveau là [SVMS96]. L'étape suivante, la synthèse est aujourd'hui bien étudiée et automatisée. Elle consiste pour la partie matérielle à trouver pour tout code RTL des correspondances de fonctions logiques dans une librairie mise à disposition par le processus physique de production de puce. L'élément de base de la librairie est l'opérateur *NON-ET* et tous les autres éléments en sont dérivés. Ensuite toutes ces fonctions logiques sont placées sur le plan de la puce et les connections sont dessinées (*place and route*) afin d'obtenir un plan finalisé qu'on peut ensuite passer en production. Pour la partie logicielle de la puce il suffit de la compiler.

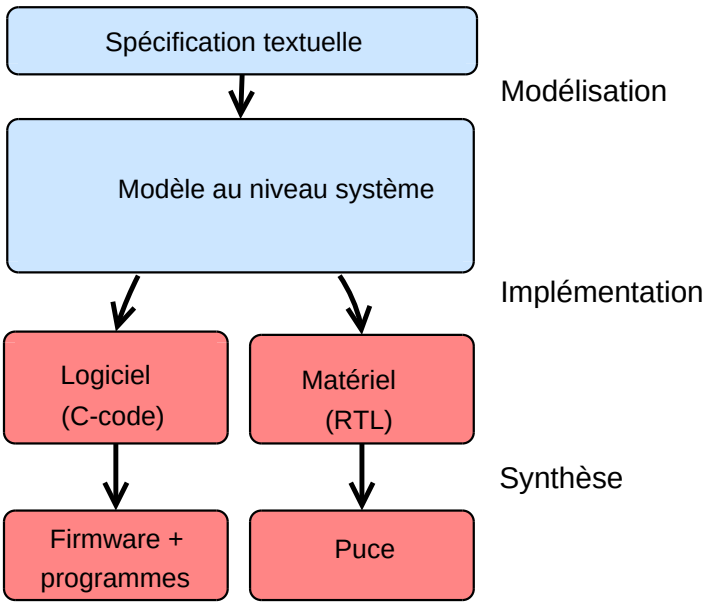


Figure 1.1: Flot de conception conjointe

Selon la complexité du système, on peut obtenir différentes couches de logiciel; un *Firmware* ou *Bios* pour les plus simples, pendant que les plus sophistiquées comprennent aussi un système d'exploitation et des applications.

1.1 L'approche modulaire

Vu la complexité de la plupart des projets, l'écriture du modèle au niveau système en un seul bloc est très difficile. La spécification est donc découpée en morceaux qui forment des unités fonctionnelles, et on définit des interfaces de communication entre ces entités pour pouvoir ensuite les traiter séparément. C'est comme ça que plusieurs personnes ou groupes peuvent travailler relativement indépendamment sur des modules distincts d'un même projet. En composant les modules, on obtient le système final, comme décrit dans la spécification initiale. Un modèle modulaire a aussi l'avantage de permettre la simulation séparée des modules, donc un débogage plus localisé.

Figure 1.2 montre un flot de conception conjointe un peu plus détaillé. On y voit une étape supplémentaire après le modèle au niveau système qu'on appelle *exploration d'architecture* comme dans [GZD+00]. Cette étape consiste respectivement en trois phases: allocation d'éléments d'exécution, partitionnement et ordonnancement. Pendant l'allocation d'éléments d'exécution on choisit l'architecture d'exécution, on décide quels processeurs et autres éléments

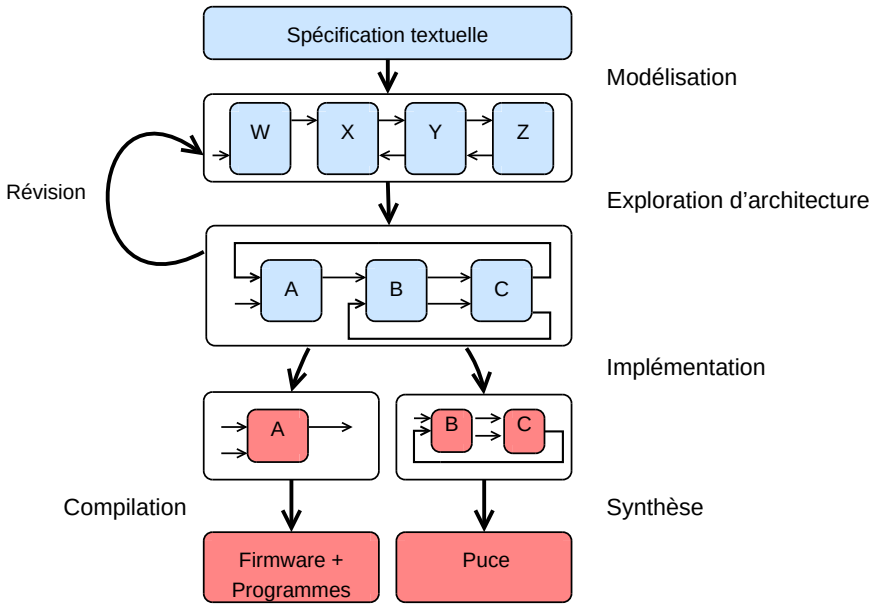


Figure 1.2: Flot de conception conjointe modulaire

d'exécution sont utilisés. Pendant le partitionnement, on décide quelles parties du modèle sont exécutées sur les éléments d'exécution et lesquelles vont être synthétisées en matériel. Ensuite pendant l'ordonnancement, on résout tout parallélisme superflu. Si plusieurs modules sont exécutés sur un seul élément d'exécution, on détermine l'ordre de leur exécution tout en respectant leurs points de synchronisation.

Après l'exploration d'architecture, on obtient donc un autre modèle modulaire, le modèle au niveau architecture. Ses modules ne correspondent pas forcément aux mêmes fonctionnalités que les modules du modèle au niveau système. Le modèle au niveau architecture peut alors subir des tests de fonctionnalités. Une simulation au niveau système est selon la complexité du modèle toujours possible mais elle nécessite bien plus de ressources et de temps qu'au niveau système. Pour une simulation plus rapide, on va simuler juste un ou peu de modules au niveau architecture. Le reste du système est simulé au niveau comportemental ce qui permet de focaliser sur des fonctionnalités spécifiques tout en gardant le système complet comme environnement. La validation ou la vérification d'équivalence sont d'autres méthodes pour assurer le fonctionnement correcte du modèle. Ces méthodes comparent le comportement des modèles au niveau architecture et système, pour savoir s'ils sont identiques.

Bien que le modèle au niveau architecture contienne déjà des détails d'implémentation, il manque encore de nombreux détails par rapport au niveau

RTL. Des manipulations comme le choix de protocoles de bus et l'implémentation des interfaces correspondantes sont comprises dans l'étape implémentation à l'issue de laquelle on obtient un modèle modulaire de la partie matérielle au niveau RTL et un modèle modulaire de la partie logicielle dans du code exécutable. Un modèle sur ce niveau de détail - appelé ici le niveau implémentation - exige beaucoup de temps et de ressources pour la simulation [Klo05]. Le modèle d'un système entier ne peut donc pas être simulé en détail sur ce niveau. Il existent cependant plusieurs techniques pour être capable de repousser la limite de taille de modèle qu'on est capable de traiter. Ces techniques comprennent l'émulation matériel [GNJ+96], l'accélération matériel [CMA02], la simulation mixed level [JVBEK91], la co-simulation matériel logiciel [GCNCM92, BST92], et d'autres approches spécialisés [YHP+97, Klo05].

Ce qu'on fait à la place, c'est de simuler chaque module séparément. Il est aussi possible de simuler un module au niveau implémentation avec son environnement qui est au niveau architecture.

Au total, la modularisation apporte une hiérarchisation des modèles. Des fonctionnalités qui vont ensemble peuvent être isolées et donc développées et testées séparément. Ceci est important pour distribuer le travail sur plusieurs personnes ou équipes. Elle aide également à surmonter les limitations de certains outils et méthodes par rapport à la complexité du modèle.

1.2 Réutiliser c'est accélérer

La réutilisation constitue un aspect important pour accélérer le processus de développement de systèmes embarqués. Elle est une conséquence logique d'un développement strictement modulaire. Si on parle aujourd'hui de *familles de produits* et *de générations de produits*, on fait référence à toute une gamme de produits qui contiennent un grand sous-ensemble de modules identiques. Ils diffèrent d'autres produits uniquement par quelques modules de modifiés ou rajoutés. Pendant que la réutilisation dans le cadre d'une évolution ou spécialisation d'un produit semble évident, elle l'est moins quand l'environnement d'utilisation s'éloigne de l'environnement d'intégration initial.Elle devient aussi plus difficile quand il n'y a pas accès au développeur d'origine de cette propriété intellectuelle (IP) qui connaît en détail les fonctionnements internes.

Figure 1.3 montre comment des IPs peuvent être intégrées dans un flot de conception conjointe. Puisqu'on travaille sur plusieurs niveaux d'abstraction d'un système embarqué, un IP doit être disponible dans plusieurs niveaux d'abstraction pour pouvoir l'intégrer dans tous les niveaux. On voit dans la figure comment le IP A contient à la fois une description au niveau architecture et au niveau implémentation. Ceci permet une simulation efficace au niveau architecture pour

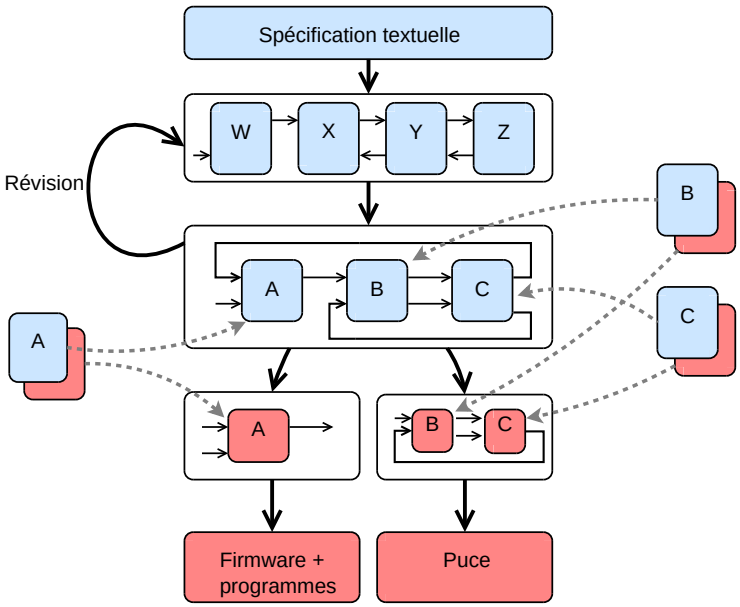


Figure 1.3: Flot de conception conjointe avec réutilisation

vérifier si l'environnement se comporte bien au niveau des entrées et sorties. De plus, au niveau implémentation, le IP permet de passer directement à la compilation pour le cas d'un IP logiciel ou de passer à la synthèse logique dans le cas d'un IP matériel.

La réutilisation fait aujourd'hui partie intégrante de la majorité des projets, soit sous forme d'IPs provenant du même projet ou de la même entreprise, soit avec des IPs achetés chez des vendeurs IP commerciaux, soit des IPs disponibles librement à la communauté (sur OpenCores.org [Cor] par exemple). Néanmoins une intégration IP réussie pose de nouveaux défis aux ingénieurs. Il est difficile d'intégrer correctement un composant sans avoir participé à son développement. Les protocoles des entrées et des sorties par exemple sont figés par le code du composant, mais la documentation n'est souvent pas très détaillée. En plus du code d'implémentation, il faut une documentation pertinente décrivant les interactions avec l'extérieur. Mais même une documentation bien détaillée peut contenir des erreurs ou laisser plusieurs possibilités d'interprétation. Pour des composants complexes la tâche reste complexe même avec une description détaillée, puisqu'il est difficile de vérifier si l'environnement adhère à toutes les exigences de la description.

Pour compliquer davantage, les vendeurs d'IP préfèrent conserver un maximum de détails d'implémentation afin de lutter contre le plagiat. Ils préfèrent proposer des IPs *boîte noire* ou *boîte grise* qui ne contiennent pas le code source

de l'implémentation mais uniquement les objets compilés ou synthétisés. Ceci est un dilemme évident dans la pratique de la réutilisation d'IP pour des systèmes embarqués.

Pour résumer, la complexité des projets et la cadence du marché des systèmes embarqués nécessitent la réutilisation de composants du système, et elle est déjà largement pratiqué dans l'industrie. Par contre le potentiel énorme de la réutilisation pour l'accélération des processus de développement et pour l'avancement dans la complexité des systèmes n'est pas facile à exploiter. L'effort et les coûts supplémentaires pour l'intégration et le test de ces composants entravent des gains de productivités plus importants.

Chapter 2

Pourquoi utiliser des méthodes formelles dans la conception conjointe?

Les méthodes formelles sont basées sur des techniques mathématiques pour la description de propriétés dans le cadre d'un développement de systèmes matériels et logiciels. Des méthodes formelles servent à spécifier, développer, et vérifier des systèmes d'une façon systématique. Il y a plusieurs intérêts à utiliser des méthodes formelles dans la conception conjointe, en particulier pour la prévention et à la détection d'erreurs. Dans la suite nous détaillerons les principales motivations à utiliser les méthodes formelles dans la conception conjointe.

2.1 Assurer que la spécification soit complète

Dans la conception conjointe, une première utilisation des méthodes formelles serait d'assurer que la spécification est conforme à ce qui est décrit dans le cahier des charges du système. Il est difficile d'assurer qu'une spécification non-formelle contienne tout ce qui a été prévu dans les demandes initiales.

Il est également difficile de veiller à ce que la spécification ne contienne pas de détails d'implémentation. Si elle contient des détails superflus, elle contraint l'implémentation à respecter ces détails, ce qui peut entraîner des inconvénients au niveau des performances et de la structure. Une spécification formelle n'est pas entièrement immunisée contre ce type de défauts, mais elle en contient conceptionnellement moins.

2.2 Réduire le nombre d'erreurs dans la spécification

Une deuxième raison pour l'utilisation de méthodes formelles serait de chercher à obtenir une spécification qui contient moins d'erreurs. En contraste au point précédent, il ne s'agit pas d'inclure ou d'exclure des comportements dans la spécification mais plutôt de s'assurer que ce qui est décrit est correct. Sachant que toute erreur dans la spécification est susceptible d'être propagée dans l'implémentation et que les coûts impliqués par la correction d'une erreur augmentent si elle est découverte plus tard dans le processus de développement, il est important de déboguer la spécification le plus méticuleusement possible. D'un point de vue commercial il est également conseillé de faire plus d'efforts dans la prévention d'erreurs afin d'éviter des coûts plus élevés après. Il est surtout important d'éliminer les problèmes de fonctionnalité sur ce niveau là puisqu'il est difficile de vérifier des comportements globaux dans un modèle plus détaillé. Des simulations avec des vecteurs de test limités d'une spécification non-formelle ne peuvent pas découvrir tous les problèmes, avec des méthodes formelles par contre on peut en découvrir davantage.

2.3 Réduire le nombre d'erreurs dans l'implémentation

Toute implémentation contient des problèmes, et une bonne partie du développement est dédiée au test et au débogage de l'implémentation finale. Alors que de vastes tests et simulations sont indispensables, les vecteurs d'entrées choisis ne peuvent que couvrir une petite partie de l'ensemble des états possibles du système. Les méthodes formelles sont un complément important pour éliminer plus de problèmes fonctionnels et logiques dans une implémentation. Beaucoup d'outils de recherche se concentrent sur ce domaine.

2.4 Accélérer le développement, réduire les coûts

Un des mythes évoqués dans [Hal90] dit que des méthodes formelles sont coûteuses à utiliser et que leur utilisation demande beaucoup de temps. Même s'il est vrai que l'utilisation de méthodes formelles prend un certain temps et coûte de l'argent, son rendement peut être énorme s'il aide à éviter seulement quelques erreurs critiques. En utilisant des méthodes formelles, la phase de spécification

du projet prend typiquement plus de temps, par contre, avec une spécification claire et correcte, l'implémentation, l'intégration et la phase de tests peuvent être effectuées beaucoup plus rapidement. Il est difficile de mesurer objectivement la productivité du développement et la qualité d'un système pour des méthodologies différentes. Mais beaucoup d'entreprises utilisant des méthodes formelles parlent de temps de développements et coûts réduits par rapport à des méthodologies plus conventionnelles.

2.5 Améliorer la fiabilité du système

Pour des produits comme un lecteur portable multimédia ou un système hi-fi de voiture, la fiabilité n'est pas un sujet primordial. L'utilisateur accepte quelques inconvénients, mais trop de fautes et des blocages fréquents vont être une expérience insatisfaisante pour le client. Pour des téléphones portables, la fiabilité devient déjà un sujet important et il est considéré comme normal qu'un nouveau téléphone nécessite des mises à jour du fabricant dans les premiers mois pour enlever des erreurs importants. Les temps de développement très limités par le marché vont accentuer cette tendance dans les années à venir; on va connaître des mises à jour pour le téléviseur, la climatisation et le chauffage domestique jusqu'au four et la machine à laver.

Même si une multitude d'erreurs peuvent être corrigées avec des modifications de *firmware* et que les mises à jour vont devenir de plus en plus transparentes pour l'utilisateur, chacune cause des coûts au fabricant et diminue la confiance du consommateur dans le produit et dans la marque. La vérification formelle pour les systèmes embarqués de base pourrait contrer cette tendance et permettre au fabricants de livrer des produits d'une qualité supérieure.

Il y a une catégorie de systèmes entièrement différente pour laquelle les méthodes formelles se sont déjà établies dans les flots de conception standard, ce sont les systèmes critiques. Des systèmes comme le contrôle de réacteurs nucléaires, des parties critiques d'un avion ou la commande de missiles nécessitent une fiabilité maximale qui ne peut être atteinte qu'avec l'utilisation de méthodes formelles tout au long du processus de développement. Le succès que les méthodes formelles ont connus dans ces domaines et le manque croissant de fiabilité dans les produits de tous les jours mettent en évidence l'inévitable universalisation des méthodes formelles.

2.6 Prouver le respect de standards

Dans le développement de systèmes à grande échelle en particulier, le respect de standards joue un rôle très important. Être capable de prouver le respect d'un

standard pour des protocoles de bus ou des API standardisés, facilite l'intégration à grande échelle. Si un standard est bien formulé et si on peut prouver que deux composants respectent celui-ci, on peut supposer que les deux composants vont bien pouvoir communiquer ensemble.

L'utilisation de méthodes formelles peut assurer qu'un composant suit un certain standard d'une façon beaucoup plus fiable que de simples tests ou simulations pourraient le faire. Des propriétés formelles peuvent vérifier les fonctionnalités importantes des interfaces et ainsi valider le respect du standard.

Chapter 3

L'intégration de méthodes formelles dans la conception conjointe

Selon les besoins, les résultats souhaités et les goûts, il y a différentes manières d'introduire des méthodes formelles dans un flot de conception conjointe traditionnel. Comme détaillé en Chapitre 2 l'utilisation de méthodes formelles a deux buts majeurs: la prévention d'erreurs et la détection d'erreurs. L'utilisation en amont dans la conception a un caractère plutôt préventif alors qu'une utilisation en aval de la conception cible plutôt la détection de problèmes. Les différents emplois de méthodes formelles sont aussi divers que les méthodes formelles et que les différentes étapes de conception elles mêmes. Pour promouvoir l'adoption du formel dans l'industrie, il est question aujourd'hui d'identifier des méthodologies de conception et des méthodes formelles adaptées pour atteindre un résultat optimal d'une façon efficace.

Cette partie présentera trois pistes que nous avons suivies, et qui illustrent comment intégrer les méthodes formelles dans différentes étapes du flot de conception conjointe. Ces approches sont assez différentes et potentiellement complémentaires et montrent comment la diversité des méthodes formelles et les différentes couches de la conception donnent un vaste champ d'application.

3.1 Transformation automatisée vers un langage formel

Un des plus grands reproches fait aux méthodes formelles concerne leur complexité d'utilisation et le fait qu'elles ne soient pas adaptées aux besoins des ingénieurs. Ce point faible pourrait être éliminé si on arrivait à les utiliser, sans

que l'utilisateur s'en aperçoive. Comme la réutilisation de composants est un sujet d'importance croissante, on a travaillé sur l'utilisation automatique de méthodes formelles pour détecter plus de fautes dans la composition de blocs IP. Dans ce but on génère des interfaces comportementales formelles à partir de blocs IP non-formelles.

L'interface d'un composant IP ne contient traditionnellement pas beaucoup d'informations. Souvent, seuls les signaux d'entrée et de sortie sont spécifiés ainsi que leur types respectifs. En plus de cela la documentation textuelle des composants peut contenir des informations sur la communication avec le composant pour en assurer le bon fonctionnement. Pour plus de détails on peut d'habitude se renseigner dans le code source des composants. Par contre pour des IPs achetés à un tiers, cette option n'est souvent pas disponible puisque les vendeurs préfèrent fournir des boites noires ou grises pour ne pas perdre le contrôle sur le code et pour empêcher leurs clients d'effectuer eux-même des adaptations.

Des interfaces comportementales peuvent contenir bien plus d'informations sur un composant que les noms et types des entrées / sorties. Elles peuvent contenir des informations comportementales du composant comme la synchronisation de signaux, des relations temporelles entre signaux et tout autre détail de comportement jusqu'à sa représentation complète. De la même façon qu'un compilateur vérifie si les types de données de deux signaux connectés sont compatibles, un outil formel peut signaler des erreurs dans l'interaction de deux composants quand les interfaces formelles sont composées.

Dans l'article [TGB+03], on montre comment passer automatiquement d'une description non-formelle à une représentation formelle. Une description JAVA temps réel est automatiquement traduite dans le formalisme SIGNAL [BLJ91] pour ensuite faire un mapping sur une architecture d'exécution spécifique. Les résultats montrent des gains de performance importants en temps d'exécution grâce à l'optimisation de l'ordonnancement.

Dans [TBS+04a], on présente la théorie pour un système d'inférence de types comportementaux formels. SIGNAL est utilisé comme support formel qui permet une description de systèmes avec différentes horloges. Ce formalisme est bien adaptée à cette tâche entre autres parce qu'il permet d'exprimer d'une façon naturelle les notions de dépendance et de synchronisation entre signaux. Comme environnement non-formel, cet article se base sur un flot de conception SystemC, qui est un des cadres de conception au niveau système les plus populaires.

3.2 La modélisation formelle itérative

Cette approche se base sur l'hypothèse que beaucoup d'erreurs dans l'implémentation sont dues au fait que la spécification n'est pas suffisamment claire, qu'elle contient

des erreurs logiques qui passent souvent inaperçues jusqu'à très tard dans le processus de développement. Des fautes dans une conception provoquent toujours des coûts, mais encore plus s'ils sont découverts dans une étape ultérieure. Leur coûts sont maximales après le déploiement chez le client.

On comprend donc bien l'intention derrière cette approche, par contre elle n'est pas sans problèmes. Le développement de spécifications formelles est souvent assez complexe et n'est toujours pas maîtrisé par la plupart des ingénieurs. Il demande également de passer plus de temps dans la période de spécification, ce qui peut signifier qu'on prend plus de temps pour créer une première implémentation prototype. La vérification de spécifications formelles peut prendre beaucoup de temps et de ressources comme la mémoire vive et le temps de processeur, et souvent des abstractions doivent être faites pour que la vérification se termine en un temps acceptable.

Une fois que la spécification formelle est complète et vérifiée, on commence à construire l'implémentation correspondante. Évidemment, il est difficile de garantir la transformation correcte, mais néanmoins une certaine catégorie d'erreurs a été éliminée.

On propose une approche différente pour créer des modèles formels. On utilise des méthodes agiles comme dans le *Extreme Programming* (XP) pour construire incrémentalement des modèles corrects par construction (CBC). On pense que suivant cette approche on est plus rapide à créer les modèles et qu'ils sont mieux structurés ce qui facilitera le passage à l'implémentation.

3.3 Validation de protocoles insensibles aux latences

Dans la conception de systèmes embarqués de taille importante et avec des horloges rapides, on atteint de plus en plus souvent une barrière où certains fils sont plus longs que la propagation d'un signal pendant un tic d'horloge. Suivant des estimations très optimistes, les délais pour des fils optimisés vont s'élever au moins entre 6 et 10 cycles pour une puce à 10 GHz fabriqué en 50 nm [BM02]. Pour ces puces, une conception strictement synchrone n'est plus possible, ils ont besoin de multiples domaines d'horloges ou de désynchronisations. Le domaine des protocoles insensibles à la latence [CMSV01] s'occupe justement de ces problématiques. Plusieurs protocoles ont étés présentés qui - d'une façon plus ou moins automatique - cherchent à éliminer les conséquences de fils longs sans être obligé d'effectuer trop de modifications dans les composants synchrones originaux. Un des problèmes des protocoles insensibles à la latence est par contre la preuve de l'exactitude de la transformation. Bien que la plupart des auteurs de ces protocoles affirme que le comportement avant et après les transformations

est préservé par construction, peu livrent des preuves formelles et celles qui sont fournies sont incomplètes ou difficile à suivre.

Dans [SMBS05], on a tenté de vérifier formellement la préservation du comportement avec un *model checker*. On fait ceci sur plusieurs versions de protocoles ainsi que sur une version que nous avons proposée, basée sur l'implémentation de Carloni, qui élimine le besoin pour des stations de relais - au dépens de fils supplémentaires. Dans [SBM+05], on utilise la programmation fonctionnelle pour la validation de différents protocoles insensibles aux latences. Finalement, [SMBS06] résume tous nos travaux sur ce sujet.

Chapter 4

Mise en oeuvre

Ce chapitre décrit les différentes étapes de l'intégration de méthodes formelles à différents moments du flot de conception conjointe.

4.1 Transformation de composants SystemC en SIGNAL

Pour illustrer la génération de modèles formels à partir de composants non-formels on a choisi SystemC [GLMS02] comme langage d'entrée, SystemC étant un des langages les plus répandus pour la modélisation au niveau système. Le formalisme de sortie est le langage SIGNAL. Ce langage fait partie de l'environnement Polychrony [IRI] qui dispose d'outils comme le vérificateur de modèle, SIGALI [MBLL00]. SIGNAL est un langage synchrone orienté flot de données qui permet de représenter des modèles polychrones, c'est-à-dire des modèles pouvant contenir plusieurs horloges.

La figure 4.1 montre le flot de conception pour cette approche. Pour traduire le comportement d'un module SystemC, on utilise le compilateur GCC et son format intermédiaire GIMPLE/SSA. La structure de contrôle d'un code GIMPLE/SSA est beaucoup plus simple que celle d'un code C++ et en conséquence une représentation GIMPLE/SSA est plus facile à traduire. Par contre, on perd l'information sur la structure initiale du programme SystemC. Une transformation purement comportementale, sans conservation de structure a cependant plusieurs inconvénients. Sans cette information structurelle, il est très difficile pour le compilateur de SIGNAL de faire des optimisations de haut niveau comme par exemple pour l'ordonnancement. De plus, sans structuration le code SIGNAL résultant devient illisible et complique donc toute modification manuelle du modèle. Finalement une fois une erreur se manifeste dans le modèle généré, il est difficile la tracer dans le modèle d'origine. Donc, sans informations struc-

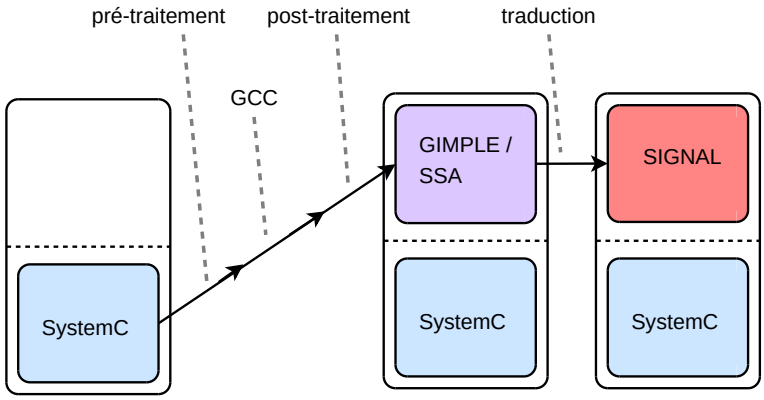


Figure 4.1: Traduction de modules SystemC en SIGNAL

turelles, une transformation correcte d'un modèle SystemC en SIGNAL n'est pas très utile. C'est pour cette raison qu'on extrait d'abord l'information structurelle comme décrit en section 4.2, et seulement ensuite on traduit le comportement des modules.

Il faut aussi noter que GCC ne connaît pas les macros SystemC. Donc, si elles étaient traitées directement avec GCC, elles seraient expansées, ce qui rendrait difficile la lisibilité du code généré. C'est pourquoi, afin de générer du code plus propre, il est nécessaire de faire du pre-processing pour masquer ces macros à GCC, puis de passer par GCC avant de finalement les faire réapparaître dans le code SSA pendant une étape de post-processing.

Dans [BTSG04], on démontre comment, se basant sur la théorie de [TBS+04a], des types comportementaux formels peuvent être générés automatiquement. À l'aide d'une étude de cas sur un filtre à réponse impulsionelle finie (RIF), cet article montre le flot complet de transformations de composants SystemC en une description SIGNAL. Il montre également comment GCC et son format intermédiaire GIMPLE/SSA peuvent être utilisés pour faciliter cette transition. Cette approche est présentée plus en détails dans [TBS+04b]. Toutefois, ces articles ne donnent aucune solution quant à la préservation de l'architecture du système qui est difficilement restaurable, voire perdue si on utilise une approche basée uniquement sur GIMPLE/SSA. Le procédé présenté dans ces travaux est donc utilisable pour le traitement de composants un par un, mais pas pour un système composé de plusieurs modules. La partie suivante présentera une solution pour préserver la structure du programme.

4.2 Extraction d'informations structurelles de modèles SystemC

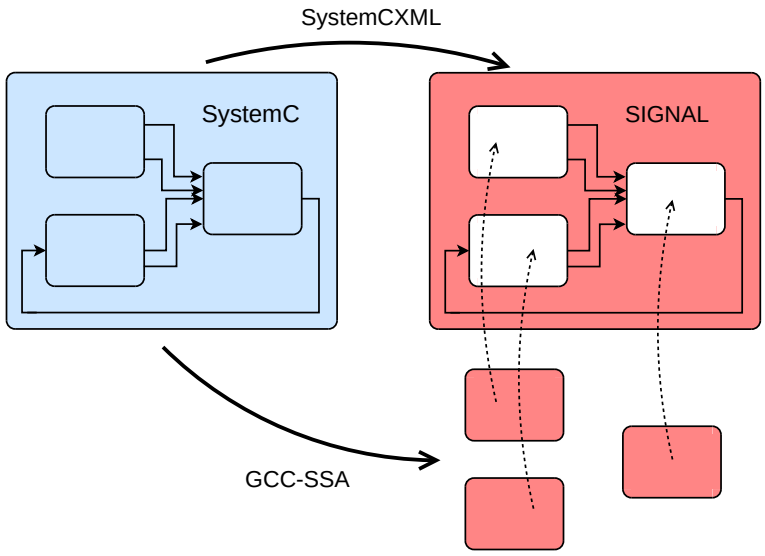


Figure 4.2: Méthodologie pour la transformation de SystemC vers SIGNAL

Afin d'éviter le problème de perte d'informations structurelles, on extrait d'abord ces informations du code original SystemC avant de traduire le comportement des modules comme cela est présenté dans la section 4.1. Ainsi, on a réalisé un *front-end* pour SystemC qui est présenté dans [BPM+05b]. Cet outil permet d'analyser des projets SystemC entiers, d'extraire leur informations structurelles et de générer ensuite un squelette formel en SIGNAL. Ce squelette représente uniquement l'architecture du programme d'origine avec l'information sur les modules et leurs connections, mais sans information sur le comportement de ces modules. Ensuite, il est possible de générer ce comportement à partir de ce squelette, de GCC et des transformations GIMPLE/SSA → SIGNAL. Ainsi, grâce aux informations de contrôle, le compilateur SIGNAL est capable de faire des optimisations de haut niveau comme par exemple un ordonnancement plus efficace.

Ces informations structurelles sont converties dans une représentation XML. Dans [BPM+05b], on démontre comment cette représentation peut être utilisée dans d'autres cadres, comme la visualisation de différents aspects d'un modèle ou la génération automatisée de tests. Il existe d'autres outils qui permettent de visualiser des modèles SystemC, mais ils comportent certaines limitations par rapport à notre implémentation. Certains de ces outils ne sont pas libres comme

par exemple l'outil *Incisive* de Cadence [Cad05]. D'autres, comme [GLLA03], nécessitent des modifications de la librairie SystemC . Enfin, d'autres encore, comme les outils gSysC [EAH05] ou SystemPerl [Sny], exigent des modifications dans les modèles SystemC.

Dans [PMBS06], tous ces résultats sont placés dans un contexte plus large pour décrire une architecture orientée services pour la validation de modèles au niveau système. L'analyse de l'architecture SystemC y est utilisée pour une plateforme qui offre des services comme l'introspection et la réflection, le débogage interactif, la visualisation et la génération automatisée de tests fonctionnels.

4.3 Modélisation formelle itérative

La modélisation formelle itérative consiste à utiliser les méthodes agiles - utilisées avec beaucoup de succès dans la programmation extrême (PX) [Bec00] - pour la construction de modèles formels, afin d'accélérer leur construction et d'améliorer leur qualité en terme de nombre d'erreurs et de structuration.

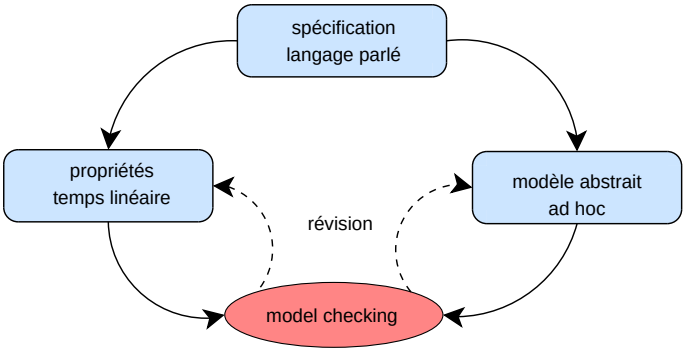


Figure 4.3: Construction classique de modèles formels

La figure 4.3 montre comment un modèle formel est habituellement construit. Partant d'une spécification exprimée dans un langage usuel comme l'anglais ou le français, on écrit un modèle abstrait *ad hoc*. Selon la taille du système décrit ceci peut être assez grand. Après on définit des propriétés en temps linéaire qui sont également déduites de la spécification initiale. À partir de ces propriétés formelles, le modèle formel doit être validé avec un *Model Checker* comme SPIN [Hol03a]. Cependant, cette approche connaît plusieurs problèmes. D'abord, il peut y avoir des erreurs dans le processus de construction d'un modèle formel et les efforts pour la construction et le débogage des modèles augmentent exponentiellement avec la taille. Puis, comme il n'y a pas de mécanisme pour vérifier que les propriétés formelles ont toutes été spécifiées, il est très probable d'en omettre quelques unes

ce qui en revanche va réduire la pertinence de la vérification du modèle. Aussi, si le modèle contient une erreur il est difficile de la trouver puisque les outils de vérification souvent ne donnent aucune indication par rapport à l'emplacement dans le code du problème. Finalement, un modèle ainsi construit a tendance à contenir plus de comportements que ce qui est nécessaire d'après la spécification. Le modèle peut donc contenir des propriétés non voulues qui vont être propagées dans l'implémentation.

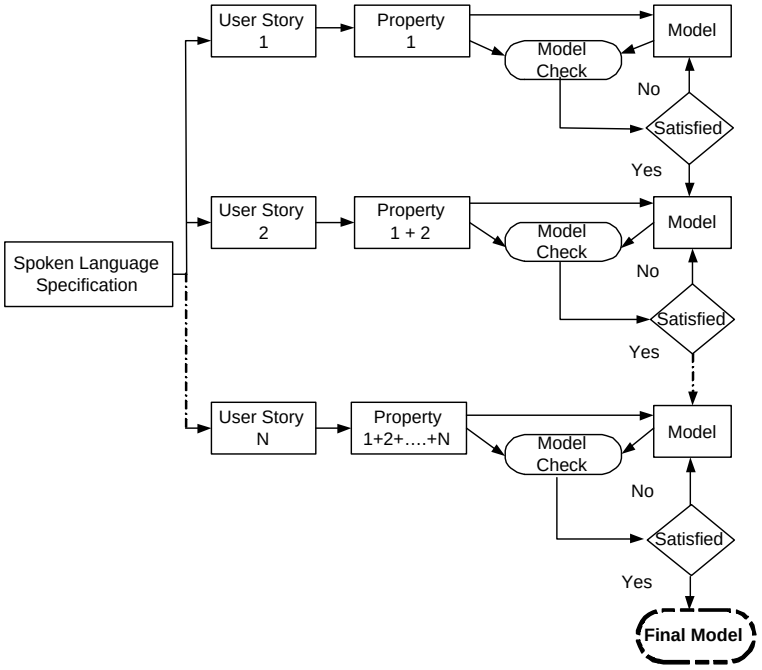


Figure 4.4: La modélisation formelle itérative

La figure 4.4 montre le processus de la modélisation formelle itérative. Comme dans la PX, on extrait des petits scénarios (*user stories*) de la spécification informelle. Ces scénarios décrivent une fonctionnalité spécifique du système qui peut alors être représentée par une propriété formelle. On choisit d'abord un scénario, puis on écrit le modèle formel correspondant et enfin on vérifie son exactitude en faisant du *model checking* avec la propriété formelle associée. Une fois ce petit modèle validé, on choisit un autre scénario, et élargit ce modèle de façon à ce qu'il contienne cette nouvelle fonctionnalité. Par la suite, on vérifie à la fois la validité de la propriété du nouveau scénario et de toutes les propriétés déjà intégrées dans le modèle. Et ainsi de suite jusqu'à ce que le modèle contienne tous les scénarios et que toutes les propriétés formelles soient vérifiées.

Un des avantages de cette méthodologie est que la construction de modèles formels est plus contrôlée. Si entre deux étapes incrémentales le nombre d'états

change de beaucoup, la cause de cette est identifié et peut donc éventuellement être évitée, soit par la réécriture des changements apportés, soit par l'intégration des changements apportés plus tard dans la création de modèle. La création de propriétés formelles se fait en même temps que la création du modèle. Donc, dès le début, on a un modèle formel exécutable et vérifié. Si la vérification d'une propriété échoue, on sait que l'erreur est due aux changements les plus récents et elle est donc plus facile à trouver. Un autre point fort est que cette représentation est très proche de la spécification d'origine. Comme la spécification est coupée en petits bouts et que l'ajout de ces pièces au modèle est surveillé en détails, le modèle formel de sortie est plus près de la spécification d'origine que si le modèle avait été créé directement. Ainsi, il est moins probable d'omettre des fonctionnalités puisque chaque scénario est intégré, et d'inclure des détails d'implémentation non voulus puisque, à chaque itération, seul le comportement correspondant strictement au scénario actuel est inclus.

Cette approche présente néanmoins toujours des difficultés. Comme avec l'approche ad hoc, il n'est pas évident de savoir si une propriété est correctement formulée. Pour exprimer correctement des propriétés formelles à partir d'une spécification verbale, il faut de l'expérience. Dans nos travaux, on utilise un visualiseur graphique de propriétés LTL (Linear Time Temporal Logic) [Odd01]. La représentation sous forme de machine à états finis permet de mieux évaluer si une propriété LTL correspond vraiment à ce que l'on voulait exprimer.

La méthodologie de la modélisation formelle itérative est présentée dans [BSST04]. On y trouve une illustration bien détaillée des différentes étapes comme la définition des propriétés formelles et l'extension itérative du modèle. On y démontre aussi l'approche avec plusieurs exemples plus ou moins complexes, le plus grand étant une pipeline DLX. Dans [SMBS04], on montre comment cette approche peut être utilisée pour des modèles matériels assez importants. On y traite notamment la construction de modèles formels pour le bus ISA et pour la phase d'arbitrage de bus du processeur Pentium Pro de Intel. Ces expériences valident notre approche pour des modèles plus grands et démontrent son utilité pour spécifier des applications matérielles.

L'article de journal [SMSB05] synthétise tout sur le sujet de la modélisation formelle itérative. En outre, on y étudie la dépendance de la qualité du modèle et l'ordre de modélisation des propriétés. On constate qu'en changeant l'ordre de modélisation des propriétés, le modèle résultant change aussi en taille et en nombre d'états accessibles. Différentes méthodes d'ordonnancement sont comparées afin d'optimiser la taille du modèle final ainsi que son espace d'états. L'article présente aussi une interface graphique pour expérimenter les effets des différents ordonnancements.

Conclusion

Les systèmes à grande échelle et la réutilisation croissante de composants depuis de sources diverses rendent la validation de systèmes extrêmement difficile. Les tests et simulations restent des méthodes de validation importantes, par contre ils ne suffisent plus pour un nombre d'applications de plus en plus important. Les modèles et méthodes formels sont une réponse possible à ce problème. Cependant, il est difficile de créer des modèles formels corrects. De plus, la plupart des ingénieurs de développement et vérification n'ont pas l'habitude d'utiliser des méthodes formelles et sont réticents envers leur adoption vu les efforts supplémentaires significatifs à fournir. Des contraintes *time-to-market* de plus en plus strictes et la réutilisation nécessitent donc l'intégration des méthodes formelles dans les flots de conception standards des systèmes embarqués.

Dans ce document, nous avons essayé de montrer dans quelle mesure la conception de systèmes embarqués peut profiter de l'utilisation de méthodes formelles. Pour cela, nous avons présenté plusieurs exemples concernant les difficultés et les divers modes d'utilisation dans la conception conjointe. Une des conditions pour une adoption globale des méthodes formelles dans les flots de conception standards pour systèmes embarqués est de cacher aux utilisateurs, du moins en partie, leur complexité inhérente. Ceci doit être fait sans trop limiter les bénéfices des outils formels: des définitions claires et des possibilités de vérification.

Il y a plusieurs pistes menant vers ce but. Une option est de créer automatiquement une description formelle depuis des modèles de systèmes non formels. Le chapitre 9 montre avec l'exemple du langage SystemC et l'environnement synchrone Polychrony comment des modèles formels peuvent être générés automatiquement *a posteriori* depuis des descriptions de systèmes existantes. Le chapitre 12 montre comment ces techniques peuvent être utilisées en pratique pour la vérification de systèmes modulaires. La méthodologie est détaillée, présentant les outils impliqués et le processus en pratique au travers d'une étude de cas.

Un inconvénient de cette méthodologie est la nature monolithique du modèle résultant. Puisque la structure du modèle original n'est pas retenue, il est difficile de garder une vision claire pour les grands systèmes, et de remonter les erreurs jusqu'à leur source. Pour éliminer ces restrictions, notre méthodologie a été étendue pour permettre d'abord d'extraire les informations structurelles

de la description système existante, puis de transformer cette structure dans le formalisme cible. Le comportement des modules peut ensuite être transformé séparément pour remplir les boites de la structure formelle. Le chapitre 13 décrit la partie structure de la méthodologie et détaille comment l'effort pour analyser du code SystemC peut être minimisé par l'utilisation intelligente d'outils open source comme Doxygen, l'analyseur XML Xerces et GCC. La transformation séparée de la structure et des comportements a pour résultat un modèle formel qui d'un coté contient les détails comportementaux nécessaires pour effectuer certaines vérifications formelles, et d'un autre coté préserve la structure de contrôle du modèle original. Ceci facilite la remontée des erreurs, la réutilisation de composants et les optimisations d'ordonnancement.

Une autre possibilité dans l'utilisation de méthodes formelles est de débuter la conception conjointe avec un modèle formel. Puisqu'une description formelle modélise des spécifications très clairement et sans équivoque, c'est intuitivement une bonne façon de commencer la modélisation d'un système. Toujours est-il que de commencer avec une description formelle présente aussi des failles et dangers, comme par exemple la difficulté de comprendre intuitivement des expressions formelles, et les problèmes liés au nombre d'états du modèle. Le chapitre 14 présente une méthodologie qui tente de simplifier le processus de construction de modèles formels. Elle emprunte des méthodes de la programmation extrême pour construire des modèles formels de façon incrémentale avec les tests formels correspondants. Comme les modèles sont vérifiés dès le début et tout au long du processus, les erreurs sont détectées plus tôt et des accroissements brusques du nombre d'états sont constatés à leur source et peuvent être évités ou réduits.

Avec une complexité grandissante des systèmes embarqués viennent des problèmes d'intégration sur puce. Certains fils qui relient des composants peuvent être plus long que ce que permettrait la vitesse de marche anticipée. Une façon de dépasser cette limite, serait d'insérer des tampons tout au long de ces fils de communication et de rajouter une logique qui se chargerait de déterminer les valeurs manquantes et les délais supplémentaires. Cette logique n'est pas facile à insérer et peut induire des changements majeurs dans des composants ou dans l'architecture du système entier. La théorie des protocoles insensibles à la latence met en place des méthodes qui permettent la génération automatique d'interfaces de protocoles quand cela est requis. Elle définit également des critères auxquels les composants doivent répondre pour être compatibles avec l'application de ces techniques. Le chapitre 15 présente une méthodologie qui aide à formellement vérifier que le résultat d'une telle conversion correspond fonctionnellement exactement au système sans délais de communication. Il est montré comment des protocoles existants insensibles à la latence peuvent être modélisés et vérifiés formellement. Cette méthode est appliquée avec succès au protocole proposé par Carloni en le vérifiant dans notre environnement de test. Dans la suite, un pro-

tocole modifié est proposé qui dispose de la nécessite d'insertion de répétiteurs sur les longs fils sans que la performance du système soit affecté. Ce protocole est vérifié de la même façon.

Perspectives

Le travail présenté dans ce document décrit une méthodologie avec laquelle il est possible d'extraire des types comportementaux formels depuis des descriptions SystemC. Les bénéfices possibles d'un tel projet sont évidents. Par contre, la puissance réelle d'une telle approche peut uniquement être démontrée avec un outil qui peut effectuer ces tâches automatiquement sur un sous-ensemble suffisamment complet du langage d'entrée. Dans l'état actuel, il manque quelques étapes avant d'atteindre ce but. Pendant que l'extraction automatique de la structure et la génération de squelettes de processus SIGNAL correspondants sont implémentées, il faut encore automatiser la transformation du code SSA vers SIGNAL. Des travaux ont déjà été menés sur ce sujet et les premiers résultats sont présentés dans [KTBB06]. Dès que le code SSA pourra être traduit automatiquement, une librairie SIGNAL correspondant à la librairie SystemC en C++ devra être mise en place. Beaucoup de ces fonctions sont déjà disponibles grâce au travail présenté dans [GGG06], mais il faut encore le compléter et le tester. Un élément important de cette librairie sera l'équivalent de l'ordonnanceur SystemC en SIGNAL, qui est indisponible pour assurer un comportement runtime équivalent. Finalement, la transformation SSA-SIGNAL devra être intégrée avec SystemCXML pour automatiquement compléter les squelettes SIGNAL générés.

Une fois que l'extraction de types marchera de façon automatique, il serait intéressant de voir comment elle se comporte en ce qui concerne le passage à l'échelle des projets, et avec des différents types de systèmes. La transformation SSA-SIGNAL une fois mise en place devrait également marcher avec les autres langages supportés par GCC comme Fortran, Java et ADA et donc il serait également très intéressant de voir comment l'extraction comportementale marcherait pour ces langages là. Une autre chose serait d'étudier comment les performances de la simulation et de la vérification sont affectées à différents niveaux d'abstraction.

Le travail sur la modélisation incrémentale formelle montre que des techniques de la programmation extrême peuvent être appliquées à la création de modèles formels. Les résultats sont encourageants et semblent indiquer que la création de modèles formels de façon incrémentale est plus rapide et plus fiable qu'en utilisant des techniques standards. Toutefois, il faudrait étudier les effets de cette méthode en terme de temps de développement, performance et qualité pour un nombre significatif de projets. Une telle étude pourrait être effectuée avec la

coopération d'entreprises de développement. Par contre elle demanderait toujours de ressources considérables, et ses résultats dépendraient beaucoup des types de projets et de l'expérience des développeurs. Pour la programmation extrême, un nombre d'études ont été entreprises qui vont dans cette direction [SASB04, LWC04], mais il n'existe toujours pas une opinion prédominante sur l'avantage empirique de la méthode [Lay04, LBB+02]. Donc pour la modélisation formelle incrémentale une étude empirique est loin d'être réalisée. Par contre, un prochain pas pourrait être son déploiement dans un environnement industriel pour avoir des retours des développeurs et des chefs de projets.

Enfin, la vérification des protocoles insensibles à la latence est un sujet important puisqu'il n'y a pas de méthodes de vérification faciles à suivre. Un grand avantage de la technique présentée est la transparence du processus de vérification. Cependant, un des problèmes restants est le passage à l'échelle. Pour des grand modèles, le nombre d'états à vérifier devient difficilement maniable. Des implémentations de protocoles complexes deviennent donc difficile ou impossible à vérifier. Un autre problème est le fait que les modèles formels pour les protocoles ne sont pas très intuitifs à écrire et donc des efforts substantiels doivent être investis pour modéliser correctement l'implémentation d'un protocole spécifique. Le travail effectué actuellement à Virginia Tech répond à ce sujet en ajoutant un environnement de vérification fonctionnel. Modéliser les protocoles par l'intermédiaire d'un langage fonctionnel est beaucoup plus simple que de construire un modèle formel en Promela. De plus, les programmes fonctionnels passent à l'échelle beaucoup plus facilement donc même des implémentations complexes ne poseraient pas de problèmes au niveau du temps de vérification ou d'utilisation de la mémoire. Les langages fonctionnels évidemment ne permettent pas la vérification formelle, donc une prochaine étape pourrait être de mélanger les deux approches. Ainsi, la vue au niveau système des protocoles pourrait être modélisée dans un environnement fonctionnel de façon efficace en même temps que les fonctionnalités noyau des protocoles pourraient toujours être modélisées en un langage formel. Ceci pourrait être fait de façon à ce que la combinaison des deux assure les fonctionnalités noyau des protocoles et les fonctions de haut niveau.

Second Part

English Abstract

Introduction

In today's industry, several trends can be observed. Increasing globalization is leading to the fact that it is uncommon that a product is both developed and produced in one place. Teams spanning over different laboratories and countries working together on one product, such as the Airbus A380, are becoming the rule rather than the exception. Another trend is the miniaturization of silicon transistors that makes us able to integrate hundreds of millions of them on a single chip - 267 millions for example for the Power5 processor of IBM in 2004. The Semiconductor Industry Association (SIA) is predicting that by 2020 the number of transistors on a single high performance chip will be exceeding 22 billions [Ass06] (Figure 4.5). The size and therefore the complexity of systems is growing exponentially [Nai02] but competition and consumer demand is asking for shorter time to market. Projects that took 24 months in 1994 had to be accomplished within only 12 months in 2004 [LW98, SR98], putting enormous pressure on development teams.

While observing these developments, one may ask how companies will be able to deal with these challenges. How can groups that are more and more dispersed develop increasingly complex products in an ever diminishing time frame?

One possibility is the degradation of product quality. Indeed, many electronic products we can buy today, especially in the leading edge lines, are not working like they are supposed to. Often products need manufacturer upgrades after purchase for correcting bugs or adding functionality that could not be integrated in time. This phenomenon is very common for mobile phones and DVD writers, but is also true for other products such as cars. It is getting obvious even to the end consumer that we are not able to handle the complexity of today's systems any more.

In order to work against this trend, designer productivity has to be increased drastically. This can be done by (i) modeling at a higher level of abstraction, (ii) the reuse of components, (iii) using formal methods in the design process, and (iv) using more advanced tools that properly integrate one or more of the above in existing design flows.

The great majority of electronic circuits is still developed at a low level of abstraction, the Register Transfer Level (RTL). A couple of years from now, this

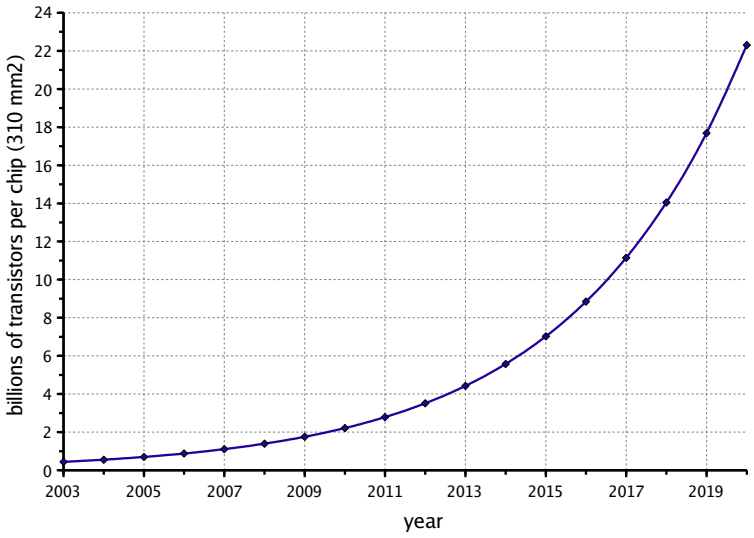


Figure 4.5: Number of transistors on a single chip predicted by the SIA in 2005

will not be a practical approach because the systems will be too complex to develop at the RTL. Therefore we eventually will have to move to higher levels of abstraction such as Transaction Level Design (TLD) or System Level Design (SLD) in order to describe the same amount of functionality with less code. The main problem in increasing the level of abstraction is that we lose tight control over implementation details and therefore key parameters such as performance, chip size and power consumption can be suboptimal. Thus, the challenge to move to higher levels of abstraction while still keeping sufficient control over the refinement process.

Another possibility to increase productivity, is to reuse the components that have proved their correctness in previous designs or to buy them directly from third party Intellectual Property (IP) vendors. The difficulty here is to find a proper format for the exchange of design information that facilitates the reuse for many different design projects. The IP description has to be available at a sufficiently high level of abstraction in order to enable an efficient high level simulation with its environment. At the same time, it has to be in sufficient detail in order to allow the transition to the physical chip description without too many problems. However, ideally it should remain independent of the physical design process. This would allow for (i) efficient simulation, (ii) fast synthesis, and (iii) reuse in different manufacturing technologies. To satisfy all of these requirements is a big challenge. Moreover, for successful integration, the existence of a detailed description of the internal functionality and the behavior of the interface is also

important in order to model the environment accordingly. However, IP vendors are not keen about disclosing technical details about their implementations as they do not want to give out their technological know-how. This is in stark contrast to the needs of the IP integration companies.

Any error that occurs during the process of system design increases costs. However, it is more expensive to correct an error late in the design process than if it is corrected earlier. Due to this fact, it is important to ensure a maximum quality of the design construction process. The simulation of a specification or a model reveals only a small fraction of design errors depending on the type and amount of test vectors fed into the system. Formal methods [Win90] offer more advanced possibilities to drastically reduce design errors. Basically they do exhaustive analysis of properties on a mathematical model of the system. There exist a plethora of different formal methods, and a particular formal method may be more or less adapted to the application in a specific step of the design flow. Also formal methods still have the reputation - and sometimes rightly so - to be too complicated to use, too slow, resource hungry, and not adapted to real application needs [Hal90, BH95a, BH95b].

Finally, yet another way to make this sorely needed leap in designer productivity is the conception and use of advanced tools. A design tool suite has to be able to follow the system creation starting from the verbal specification up to the transistor level implementation, and that too without a rupture in the semantics. Ideally, a behavioral system level design description should be automatically transformed into an RTL level description. This is already being done in some cases, however, it is not and will not be possible for optimized generic systems. More promising is a solution where the tool is taking care of all trivial and well understood transformations and the user is steering the refinement process by few critical design decisions and parameters. Such tools should also provide functionality to assist in the integration of IP's from a library of predefined blocks in order to replace modules with behavioral description by implementation level module descriptions. The challenge in the creation of future embedded systems can be solved by providing techniques that independently improve and speed up different aspects of the design flow and by then creating tools that are able to integrate these techniques coherently, in order to form a uniform design flow without semantic rupture. These tools have to find the balance between automating a maximum number of tasks and the developer's uncontested control over the implementation result.

Main Contributions

To represent the behavior of a program by a type or a proposition, we use the iSTS algebra proposed in [TG04] and inspired from Pnueli's synchronous transition systems [PSS98]. We define a formal syntax of the SystemC core statements and use it to describe a methodology and a behavioral module system to infer formal behavioral types from SystemC modules. We then show how to express these behavioral types in Polychrony (Chapter 9).

The case study on the FIR filter in Chapter sc:implgcc puts this theory to practice, validates it and illustrates how exactly such a type extraction can work. The tools needed for this process (such as the tree-ssa functionality of the GCC) are identified and put to work.

The modular type system defined does not support the handling of structural information. In the perspective to be able to treat systems with multiple modules all while preserving the structural information, a SystemC parser/analyzer is conceived and implemented as a prototype that is able to extract the structural information of SystemC system descriptions and keep it in an internal data structure as well as in XML form (Chapter 13). From this internal structure a SIGNAL skeleton code can be generated, representing the original structure of the SystemC code. We establish a methodology that integrates the modular SystemC behavioral type generation with the extraction of structural information that is able to provide formal type descriptions to general SystemC models. In order to put this methodology to work and to treat large scale models, the modular SystemC type extraction has to be completely automated, which is what we are currently working on [KTBB06].

The usefulness of this structural information for other applications is demonstrated with a visualization example and an automated test generation framework. SystemCXML has been published as an open source project and is hosted at SourceForge [BMPS04].

In joint work with Syed Suhaib, a methodology is established to incrementally build formal models with agile methods borrowed found in extreme programming. Its aim is to lower the difficulty of building formal models from spoken language requirement specifications and to create formal models that are correct by construction (Chapter 14). In contrast to the behavioral type approach, this work shows how formal methods can be used early in the design flow, aiming for a higher design quality.

The limits of synchronous communication of large systems can be extended by using latency insensitive protocols (LIP). Chapter 15 shows how to to formally verify the correctness of LIP and propose a modified LIP concept that eliminates the need for relay stations without compromising on performance.

Chapter 5

Embedded System CoDesign Flow

The objective of a codesign flow is to describe an embedded system at a functional level, independent of what parts will be implemented as hardware or software. Given this specification and certain objectives such as performance, size, costs, power consumption, weight, and memory demands, the codesign flow guides the design from the specification to an implementation with a hardware-software partitioning that satisfies these constraints.

In the large world of embedded systems, there are various different design methodologies. Differences in these designs may arise because of the complexity of the systems that are handled, the size of the company, the funding of the projects, the background of the developing engineers, and historical reasons of the company. In this chapter, we try to give some essential aspects for the joint development of hardware/software systems, that are representative of the different design methodologies.

5.1 The Basic Design Flow

A typical codesign flow - just as most chip design methodologies - can be decomposed into three basic phases: specification, implementation, and synthesis. In this chapter we first describe these three basic sections and their peculiarities with respect to codesign before making some finer grain distinctions later on.

5.1.1 Specification

Specification consists in capturing the behavioral functionality of a system into a computer readable language and arguably is the first phase in system design. While from a designers point of view it would be the starting point of a design

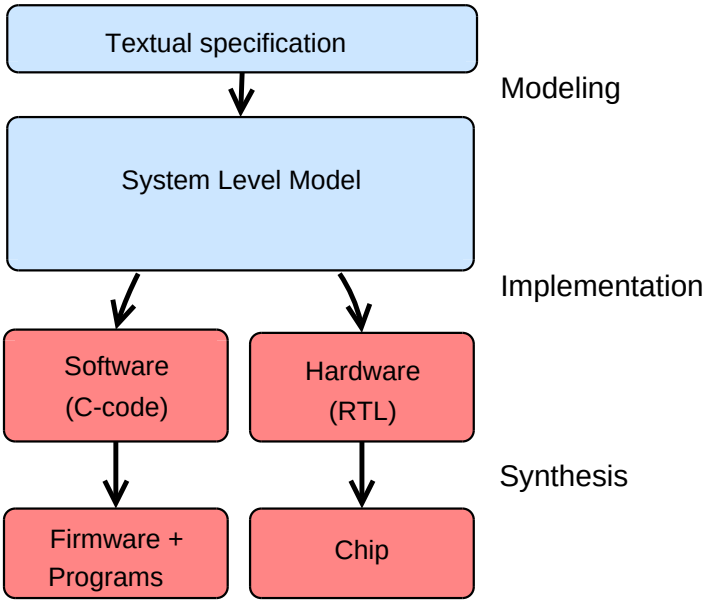


Figure 5.1: Simple codesign flow

flow, there is in fact a lot of work to be done before. Before being able to write down the specification of a system, it is advised to spend some time to pin down the exact requirements. The requirements are a set of constraints, key functional elements or tasks that the final product has to be able to perform. In starting, the requirements are usually expressed in a natural language form such as English, and may later be refined or formalized. When designing a system, it is important to have a clear notion of what the system will look like in the end, what it will contain and what not, and how it will be used. While for small systems a bright person may be able to do all this analysis without expressing it on paper, it still usually is a good idea to write it down. However it is imperative to do so for larger systems where a team is working on a project. Also, the requirements may form a contract basis, which the design team and the client agree upon. Obviously, the more clear and specific requirements are, the less likely will be the chances of expensive misunderstandings.

Typically, the system design flow of embedded systems begins by writing down requirements for the resulting system. Based on these requirements, a spoken language specification is written that tries to describe as exactly as possible all essential system behaviors and functionalities. This system specification then serves as a guideline for writing a system level model in some system description language such as SpecC [GZD+00], SystemC [GLMS02], or just plain C or C++. This system level model is supposed to describe the overall functional behavior of

the system, without covering any details about the implementation. It describes the hardware as well as the software aspects of the design since so far no distinction has been made between the parts that will eventually be implemented as hardware and the others that may be implemented as software. This first software model of the system can then be used to do system level simulation. This involves simulation of the entire system behavior by writing a testbench, which provides the model with input vectors that represent real world system input or that are likely to thoroughly test as many internal functions and critical behaviors as possible. The resulting values on the outputs of the system can then be compared to what is expected. Based on the system specification and according to the results, bugs in this system level description can be identified.

5.1.2 Implementation

While the system level model is suitable to verify all functional behavior of the system under design, its simulation cannot give any pertinent information about for example performance and timing. This is due to the fact that it is lacking implementation details such as the choice of processors used, the partitioning of the system onto the different processors and into hardware and software, and the internal and external communication protocols used. All the details are introduced in the later steps which usually start off by deciding which parts go into hardware and which one go into software. For reasonably small models or in old fashioned design style, the parts that go into hardware are subsequently rewritten in a hardware description language such as VHDL [VHD] or VERILOG [TM91]. There description are typically written on the Register Transfer Level (RTL), which can be automatically synthesized into hardware. In more complex designs there are several steps in between before reaching RTL, some of which we describe later in this chapter.

An RTL model is a cycle accurate system description, describing the data flow between registers. It contains all implementation details such as processors used, bus protocols, and algorithm implementations. RTL describes only the hardware parts of the system and it can be simulated using a logic simulator. The logic simulator can determine - given a set of input vectors - the logic states of all signals at any clock cycle. Although useful for detailed debugging, it is very resource intensive. For this reason, RTL simulation has only limited abilities to check for high level functionalities. For example, simulating the boot sequence of a 32 bit processor alone takes a lot of time, let alone verifying data processing for large systems Pure RTL level simulation is not possible for long functional sequences of large systems, however it can be used to verify in detail the functioning of small parts of the design. Of course, design correctness checking should be performed extensively at every design stage. However, since the RTL level is the lowest

element in a chain of traditionally manual operations, the bulk of verification and testing still occurs that this point of the design flow [SVMS96].

When going from a high level behavioral description to RTL, it is desirable to do this without *semantic rupture*. This means that the behavioral model is to be refined into an implementation model without rewriting it in another formalism, preserving the behavior and keeping an executable model at all times. Rewriting a model not only means a high probability of introducing errors, it also makes it impossible in most cases to refine the model successively and execute it on different stages of this process. This results in a big break for simulations and tests, which causes even more inconsistencies. Even though the positive benefits of a design flow without semantic rupture have been known for a long time, its implementation and adoption are still in early stages.

For the software part, going from the system level description to the implementation level is not as time consuming as for the hardware part. Since the behavior is already described in a software programming language, there needs to be added only some implementation details, mostly optimizations concerning the platform used, the communication and the instruction set of the processor. Simulation of the software part on the implementation level can be done just as for the system level description. More significant timing and performance results can be obtained by using a technique called Co-Simulation, where the RTL logic simulator provides an interface to include software code and simulate both parts together. As evoked before, the simulation performance may be very slow except for small system descriptions. For more advanced designs, there are simulation systems that can interface actual processor boards or FPGA (Field Programmable Gate Array) implementations of parts of the hardware. This can result in simulation speed ups of several orders of magnitude.

5.1.3 Synthesis

Figure 5.1 illustrates the different steps and layers of a simple codesign flow. The last step in the diagram are the transitions from the implementation level or RTL to the actual chip with its firmware and application programs. These transitions have been studied extensively and are largely automated for typical designs. The software side basically just has to be compiled into firmware/bios, operating system, and application programs. The hardware part is being transformed from RTL to a transistor level description (netlist) by a step called logic synthesis. For each RTL statement, the synthesis program looks for logic functions from a library of elements, provided by the physical design process of the target technology. The basic element of the library is the logic gate NAND, and all other more or less sophisticated logical functions such as multiplexers, exclusive OR (XOR), adders, and multipliers are inferred from it. If the synthesis target is an

FPGA , the chip can be programmed directly based on the synthesis result. If the target is an ASIC (Application Specific Integrated Circuit), all netlist elements are placed on a virtual chip design and then connected by wires. This process is called place and route, and delivers a detailed digital floor plan of the chip that contains layout information about all the layers of the physical chip. This layout is then sent to the chip foundry that generates lithographic mask layouts for the actual chip production.

5.2 The Modular Approach

Considering the complexity of the majority of projects, writing models on the system level is not a viable solution. Too many people would have to work on the same piece of code, with too many interactions to manage it efficiently. It is difficult to divide up tasks to several developers when developing one monolithic model. Also testing and debugging is getting all pushed towards the end where they tend to turn into a nightmare. It is therefore natural to split up the specification into several parts that represent functional units. Then, we define communication interfaces between these units in order to be able to build and test them independently, and thus provide for a more controlled and localized debugging process. Using hierarchical system specifications, several people or groups can work on different modules of the same design, without the need to interact too much - still communication among the different project members is vital, especially in early design stages, where specifications and requirements are not completely fixed yet. On composing the separate modules, we then obtain the final system model, as described in the original specification.

Figure 5.2 shows a codesign flow illustrating the aspect of modular development. The major difference is that the system model at any design step is modular and hierarchical (to simplify, the textual specification is depicted as one block, but typically has a modular structure as well). In comparison to Figure 5.1 there is an additional design step called *architecture exploration* like in [GZD+00]. It consists of transformations, guiding a design from system level to a level called architecture level. The three main steps of architecture exploration are allocation of processing elements, partitioning, and scheduling. During the allocation phase, a specific execution architecture is chosen. We decide which general purpose processors to use and the types and sizes of the RAM and ROM memory blocks. The partitioning phase consists in deciding which parts of the system are executed on which processors - these parts will eventually go into software - and which parts will be implemented as hardware blocks. Finally, we take the decision about how to resolve the superfluous parallelism. While the behavioral model is supposed to expose as much parallelism as possible, much of it is not needed any more after

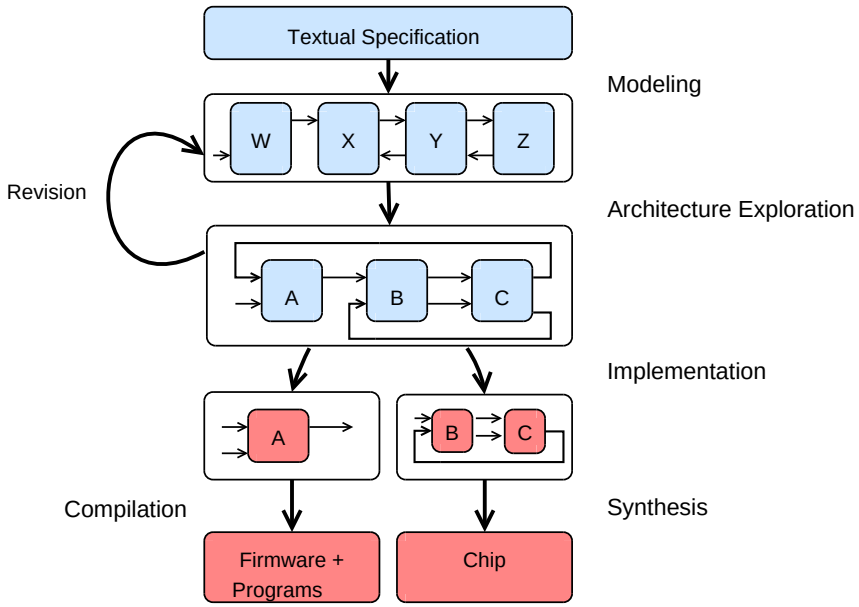


Figure 5.2: Modular codesign flow

partitioning since it is decided which parts are executed on which blocks. When two parallel blocks are mapped onto one single processor, this excess parallelism has to be resolved. This is done by scheduling the respective blocks in a way that their execution is obstructing the general execution flow as little as possible. Also, scheduling has to take into account synchronization points between blocks and prevent deadlocks in the system.

The *revision* arrow in the figure points out that the architecture exploration process is not one definitive step but rather an iterative process. A large part of the most important design decisions are taken during this phase. The design team experiments with different scenarios to find out with the help of simulation which one is the best in terms of the requirements. Obviously, the designer experience adds a lot to the speed and quality of the architecture exploration process, but it is very common that a first set of choices does not hit the target to the spot, and the designer has to go back and try a different solution.

After performing architecture exploration, the system model is at the *architecture level*. While some modules in this model may have an exact correspondence in the system level model, most of them will not. This is because the initial behavioral description does not take into account the partitioning choices but rather has a modular partitioning reflecting functional blocks. The architecture level description can now undergo additional simulation and testing, which provides us with more detailed information and probably exposes more bugs since

there are many details added. However its simulation also is much slower than on the system level.

Exhaustive functional simulation on the architectural level may not be practical any more for large designs, however, the modules can be simulated in detail separately. Also, it is good practice to perform mixed level simulation, where critical parts of the system are in architecture level detail, and the other parts are in system level detail. This enables us to do a full fledged behavioral simulation of the concerned parts while leaving them in their respective environment. Other methods to assure the correct functioning of the model comprise equivalence checking, where the external behavior of an architecture level model or parts of it are compared to the corresponding system level model.

The steps that lead from the architecture level to the RTL are combined here in the *implementation* step. While at the architectural level some predictions about performance and timing can be done, it is only after choosing the bus protocols and the implementation of the interfaces that definitive answers about these can be given. A modular implementation step can be undertaken module by module, which gives a certain continuity to the design, permits earlier testing, tolerates more parallelism in this refinement process, and therefore speeds up the overall development time. System wide simulation at RTL level is very limited or not possible at all. In order to push the limit of design sizes that still can be verified, there now exist several methods ranging from hardware emulation [GNJ+96], hardware acceleration [CMA02], mixed level simulation [JVBEK91], hardware software co-simulation [GCNCM92, BST92], and other specialized approaches [YHP+97, Klo05] enabling us to perform a minimum of behavioral testing at RTL.

Overall, modularization adds hierarchical models. Behavior that has a strong functional link can be isolated and then developed, refined, and tested separately. This is important in order to distribute work over several individuals or groups. Also, it helps to overcome the limits of certain tools or resources for testing and simulation of complex designs. Finally, the hierarchical structure makes it easier to understand the functionality of the design and therefore increases designer productivity.

5.3 To Reuse Means to Accelerate

An important potential for speeding up the development process of embedded systems lies in component reuse. It is a logical consequence of strictly modular development. Today when we talk about product families and product generations, we are referring to a whole range of products containing a bulk of identical modules. These systems differ from each other by few added or modified func-

tionalities in few modules. While reusing parts seems obvious through evolutions or specializations of products, it gets more complex with the increasing differences between the new integration environments and the original environment of the component. The reuse of blocks of functionality also becomes more difficult when there is no access to the original developer of a specific IP, who would know the details and quirks of the implementation and integration.

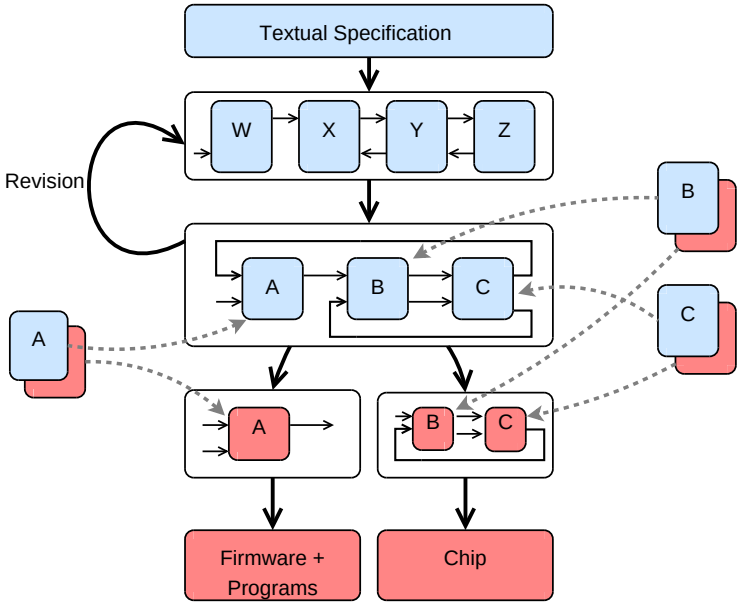


Figure 5.3: Codesign flow with IP integration

Figure 5.3 shows how the reuse of IP components can be integrated into the codesign flow. Since the actual development of embedded systems is happening on several different levels, an IP description should be available in different levels of abstraction as well. Then, on each level of abstraction, the corresponding version of the IP can be integrated, and proper functionality can be verified at all levels. The figure shows how IP *A* contains a description in the architectural level as well as another in the implementation or RTL level. This is important since it enables the designer to do an efficient simulation in the architectural level to verify the high level functionalities and also gives the possibility to go directly to logic synthesis for a hardware block or to compile the application for the case of a software module.

Reusing components nowadays is common practice in the majority of industry projects. The IPs may stem from the same project or company, from third party IP vendors or subcontractors, or from community projects for the open development of free IPs such as OpenCores.org [Cor]. Wherever they originate

from, any successful IP integration represents more or less of a challenge to design engineers. Without having been implicated in the development of a component it is difficult to integrate it properly. The interface protocols of the inputs and outputs are determined by the implementation details that are too often not properly documented. The implementation code can help to understand how to interact with the IP and what to expect from it. But figuring out these details is hard work, so a pertinent documentation of the interactions of the component with the exterior is indispensable. It is difficult, however, to provide a well structured documentation that covers all details, that is up to date, and that does not leave room for interpretation. So for complex components, even with a detailed description and well documented code, successful integration of IP is not obvious. Once the integration seems successful, it still remains to be shown that the environment adheres to all eventualities of the described interactions.

What makes matters worse, IP vendors tend to keep a maximum of implementation details for themselves in order to fight against plagiarism and not to give away the technology. They prefer to offer gray box or black box IPs, that partly or entirely hide the source code implementation but instead contain only compiled or synthesized objects. This is an obvious dilemma in the practice of reusing IPs for embedded systems.

To summarize, the complexity of projects and the growing embedded systems market need the reuse of system components, and today's industry largely relies on it. However, the enormous potential of component reuse for the speed up of development cycles and the mastery of growing complexity is more and more limited by the additional effort and costs caused by the problems of integration and testing of these components. Reducing the complexity of IP integration and assuring the correctness of component compositions will boost the productivity of the embedded system design sector. This is especially true when at the same time IP vendors have the option to hide implementation details without loosing design correctness.

Chapter 6

Why use Formal Methods for CoDesign?

Formal methods are mathematically based techniques for the description of system properties in the development of software and hardware systems [Win90, BH95b]. A method is called formal if it has a sound mathematical basis, usually in the form of a formal specification language. With formal methods one can specify, develop, and verify systems in a systematic manner. They allow to check for properties such as completeness, deadlock, or correctness for all possible system states without even having to execute the system or providing specific input vectors.

There are many different formal methods ranging from state space enumeration and abstract interpretation to deductive methods using theorem proving. For the codesign of embedded systems, formal methods can be used at different levels of abstraction and at all stages of the design flow from specification and design capture to refinement, and synthesis. However, there is still reluctance to use and adopt these techniques. This is for a large part due to general misconceptions, also called the classical myths of formal methods [BH95a, Hal90], about usability, complexity, and the effort to benefit proportion. Another reason for the slow adoption of formal methods is the lack of methodologies that smoothly integrate them into existing design flows, while smartly choosing the right formal method for the right aspect of the design flow.

Certainly, formal methods in codesign do require additional expertise and thought. However, they bring benefits that cannot be delivered with other methods and that are indispensable for an ever growing number of systems. Initially high security applications such as military, avionic, and nuclear were the main target applications of formal methods, since these kind of projects require a maximum of reliability and are ready to spend the money and effort. With the growing complexity of digital systems, however, their use is bound to propagate into all

fields.

In order to understand the different formal techniques used in codesign and the *raison d'etre* of the large bandwidth of tools and frameworks that are emerging, it is important to point out that there are number of different motivations and reasons to adopt the usage of formal methods in a codesign methodology. Most motivations ultimately come down to the prevention or detection of errors. However, it may be worthwhile to take a closer look at some of the different motivations to use formal methods in embedded systems codesign.

6.1 Assure Completeness of Specification

The initial specification of a system is crucial to the rest of the design process, since any problem that it contains will be propagated on to lower levels of the design process. With common tools, it is hard to assure that a specification is complete. A functionality that is lacking in the specification is likely to be missing in the implementation too, or, if added later it will cause delays in the development schedule.

Almost as problematic as missing system functionality is superfluous functionality or implementation details. If the specification contains details about the implementation, it restricts the free refinement of the system and results in lower key values (i.e. performance or power consumption) of the implementation. If it contains unneeded system functionality, in the best case the system will turn out to be more expensive and take longer to develop, and in the worst case, the added functionality can conflict with other functions.

6.2 Reduce Specification Errors

Another reason to use formal methods is to try to obtain a specification with as few errors as possible. In contrast to the previous point, this is not about including or excluding functionality, it is about the number of bugs in the specification. No matter how rigorously a system is developed, errors in the specification are propagated to the lower design levels. The costs for eliminating a bug is increasing exponentially the further you are in the design process. Therefore, it is important to have a high quality specification, and from an economical point of view it makes sense to spend a lot of effort on it in oder to prevent later costs.

High level functional problems are very difficult to detect at lower implementation levels. So if there are problems of this kind in the specification, they are much likely not discovered before prototype production or even later. Getting the specification right is therefore a step that deserves some precious attention, and formal methods are a powerful tool to do this.

6.3 Reduce Errors in Implementation

That today's systems contain errors is a widely accepted fact. Many errors do not have a big importance when they do not affect core functionalities, but still they might limit the usability of the product or they negatively affect the users experience. Critical errors however can make a system unusable. Therefore limiting the number of errors in a system implementation is a big concern, and a large part of system development is dedicated to debugging the final implementation. While extensive simulation and testing are important, all input vectors represent only a very limited subset of the possible input scenarios and therefore cover only a small part of the possible system states. Formal methods cover the entire state space of a model, and consequently are prone to discover a much larger range of errors. However, they are not the ultimate answer to all the errors since they typically work on an abstraction of the system that does not contain all the states of the actual implementation.

6.4 Speed up Development, Reduce Costs

One of the myths evoked in [Hal90] is that formal methods are expensive to use, and that it takes a lot of time. Moreover, in the field of embedded systems, time often corresponds to a lot of money. While the use of formal methods takes time and costs money, its return on investment can be huge if it only can help prevent few critical bugs. Using formal methods, typically more time is spent during the specification phase of the project. However, using a clear and correct implementation, integration, and testing can be done much faster. While it is difficult to measure design productivity and quality for different design methodologies, many companies using formal methods report faster development time and lower costs compared to more conventional methodologies.

6.5 Improve System Reliability

In the case of a portable multimedia player or a car hi-fi, reliability may not be the foremost concern for customers. The user accepts some inconvenience in the use, but too many quirks and too frequent crashes can quickly leave a user unsatisfied. For cellular phones, reliability is gaining importance as an increasing number of users experience malfunctioning products and are forced to install firmware updates themselves or have them done by the vendor. The ever shorter product cycles imposed by the market will further increase this trend; eventually we will see updates appearing for devices ranging from the TV set, over the refrigerator to the coffee machine.

Even if many bugs can be corrected with firmware updates and these updates are getting more and more transparent to the user, each of them is causing costs to the manufacturer and is damaging consumer confidence in the brand. Widespread use of formal verification for the design of embedded systems may counter this trend and make it possible for manufacturers to deliver higher quality products for a reasonable price.

There is another category of systems however where this transition has happened some time ago. Avionic systems, nuclear energy plants, and drive by wire systems are examples of products where required reliability and safety levels cannot be achieved without the use of formal methods. These systems were in the past the playground for a big part of the formal methods research community, since even a complex and costly use of formal methods is acceptable here even if it only would give a slight advantage in reliability and safety.

6.6 Prove the Adherence to Standards

The adherence to standards is very important, especially in the development of large scale systems and for IP reuse. Being able to formally prove the adherence to standards such as bus protocols or standard APIs is making large scale integration much less error prone and favors the reuse of third party IP. If a bus standard for example is properly formulated and if two components adhere to it, it can be assumed that they are able to communicate together.

Using formal methods can help to assure that a certain component adheres to a standard. Again, they are much more efficient for this than simple simulation and tests. With the help of formal properties it can be mathematically checked for important functionalities of interfaces and therefore validate the requirements of a standard.

Chapter 7

Integration of Formal Methods into the CoDesign Flow

Depending on the needs, the desired results, and taste, there are different possibilities to integrate formal methods into an existing codesign flow. There are basically two goals of formal methods: the prevention and the detection of errors. Applying them early in the design process typically has a rather preventive character, while using them towards the end of the design flow will rather aim for the detection and elimination of problems. There are very different ways of how to use formal methods during the design of embedded systems just as there are very diverse formal methods and different design stages. In order to promote their adoption in the industry, one has to identify design methodologies that are able to efficiently integrate certain well chosen formal methods and ease their application.

In this chapter we present some leads that we followed, which exemplify how to integrate formal methods in different stages of a codesign flow. These three approaches are quite different and possibly complementary and they show how the diversity of formal methods combined with the different design stages result in a wide field with plenty of opportunities for optimization.

7.1 Automated Transformation into a Formal Language

Some of the main reproaches made against formal methods concern their complex handling and the fact that they are not adapted to the needs and skills of engineers[Hal90, BH95a]. These weak points could be eliminated if formal methods could be transparently used in the background, where the user would not even notice it. As the reuse of components is a topic of increasing importance,

we worked on the automatic application of formal methods to detect more problems in the composition of IP blocks. In order to do this, we generate formal behavioral interfaces from non formal IP blocks.

The interface of an IP component in a typical design flow does not expose much information. Often only the input and output signals and their respective types are visible. Usually the textual documentation for the IP is trying to make up for this by detailing the necessary communication protocols in order to assure correct functioning. If more details are needed, the source code of the model maybe available for closer examination, however it is not easy to obtain the needed insight from there. Moreover, IP vendors do not like to give away the source code, they rather provide gray or black boxes of the components in order to prevent the clients to make adaptations themselves, thereby hindering the flow of information needed for integration.

Behavioral interfaces can contain much more information than just the names and types of the input and output signals. They can contain any behavioral detail of the component such as signal synchronizations and temporal relations between signals up to representing the complete behavior of the component. In a similar way that a compiler verifies if the data types of two connected variables are compatible, a formal tool can point out errors in the interaction of two components when they contain formal behavioral interfaces.

In the article [TGB+03], we show how we can transition automatically from a non formal description to a formal representation. A real-time JAVA description is automatically translated into the SIGNAL [BLJ91] formalism and then mapped onto a specific execution architecture. The results show significant performance gains mainly due to the scheduling optimizations.

In [TBS+04a], we present the theory for an inference system of behavioral types from non formal components. To capture the formal behavior we use the language SIGNAL, which is able to describe systems with multiple independent clocks. This formalism is specifically suited to this task also because it enables the dependence and synchronization relations quite naturally. On the non formal end we use here the SystemC design flow, one of the design languages most popular for system level design.

7.2 Extreme Formal Modeling

This approach is based on the hypothesis that the root of many implementation errors is actually within the specification. Often specifications are not sufficiently clear and contain logical errors that go undetected until very late into the development process. The costs for errors multiply when detected later in the development process, so the idea is to build a clear formal specification with as

few errors as possible in order to avoid later debugging costs, and which can serve as a golden reference model for later implementation steps.

The intention behind this approach is obvious, however its implementation is not without problems. The development of formal specifications is quite complex and not part of the skillset of the average engineer. Building such a formal model also means to spend more time during specification stage which can mean to reach a first functional prototype later. The verification of formal specifications can take a lot of time and resources in terms of memory and processor time. Moreover, abstractions have to be made for verifications to complete in an acceptable time.

However, once the formal specification is complete and verified, it can be used as a starting point for a corresponding implementation. Depending on the chosen languages and environments it may be difficult to obtain a transition without semantic rupture. But even if there is a semantic rupture, the formal specification can help to avoid many errors.

For the construction of formal specification models, we propose a different approach. We use agile methods that have been applied with success in the field of Extreme Programming (XP) in order to incrementally construct formal models that are verified from the beginning. The models grow with the number of formal properties integrated in a correct by construction (CBC) fashion. We think that using this approach we are able to build correct formal models faster and that they are better structured, which facilitates the transition to the implementation.

7.3 Validation of Latency Insensitive Protocols

When modeling complex embedded systems with fast system clocks, there is an upper bound for the wire length with respect to the clock. For large high performance systems, certain wires in the final chip layout can be longer than the signal propagation during one clock cycle. Based on optimistic estimations, a 10 GHz chip in 50 nm technology will contain wires with delays of 10 clock cycles [BM02]. For such a chip, a strictly synchronous design is not possible any more. We need to introduce multiple clock domains or desynchronizations. The field of latency insensitive protocols [CMSV01] is trying to deal with this problem domain. Several protocols have been presented that - in a more or less automated way - try to eliminate the consequences of long wires without being forced to make drastic changes in the original synchronous components. One common problem of latency insensitive protocols is the proof of correctness of the transformation. Even though the majority of the approaches claims to be correct by construction, few of them deliver formal proofs for the behavior preservation. Those that do it, are mostly incomplete or difficult to follow.

In [SMBS05], we try to verify formally the preservation of behavior between

the original synchronous model and the latency insensitive one. This is done on several different protocols. One of them is our modification of the Carloni implementation that eliminates the need for relay stations at the expense of using additional wires. This work has been extended mainly at Virginia Tech, USA with a functional programming framework for latency insensitive protocol validation [SBM+05].

Third Part

Formal Foundations

Chapter 8

Polychrony and SIGNAL

Polychrony is a synchronous framework, and as such its main characteristic is that it can be used to describe systems that contain components, which function at different, or even independent clocks. Therefore, the description of a system does not require an *a priori* definition of a global master clock as do endochronous descriptions such as in Lustre [HCRP91]. Instead of requiring the user to define a global clock, the clock calculus of Polychrony calculates clock trees, that result from the different clock dependencies expressed in the description. For the generation of executable code, this description has to be refined to a state where all clocks can be integrated into a single clock tree. This also mean that the system is now endochronous.

The polychronous model of computation offers a high level of expressiveness. It supports the description of non deterministic behavior, as you can find it for example in the interactions of real-time embedded applications with their environment. This is an important property for the high level specification of large scale systems, as it permits the description on a very abstract level, containing very much independently functioning components that can then be refined into a more integrated description where all communication and synchronization information is present. To deal with the apparent heterogeneity of synchrony and asynchrony in GALS architectures, designers usually consider stratified models, such as CSP (communicating sequential processes) or Kahn networks (communicating data-flow functions). By contrast, polychrony (or multi-clocked synchrony) establishes a continuum from synchrony to asynchrony: modeling, design, transformation, verification issues are captured within the same model and hence independently of spatial and temporal considerations implied by a synchronous (local) or imposed by an asynchronous (global) viewpoint. It is this flexibility that makes it so attractive for large scale embedded systems, where all components still function synchronously, but the complete system cannot be described in an entirely synchronous manner any more.

In this chapter we give a general introduction into the synchronous language SIGNAL and its associated framework Polychrony. We describe Polychrony with the help of the iSTS algebra as done in [TG04, LTL03]. iSTS is an easily accessible notation able to capture all of Polychrony. After the definition of the iSTS notation and the polychronous model of computation in Sections 8.1 and 8.2, we give a short introduction into the SIGNAL language in Section 8.3, which is the standard input formalism of Polychrony, and finally give a translation scheme between iSTS and SIGNAL in Section 8.4.

8.1 An Algebraic Notation

We start with a detailed and informal outline of an algebraic formalism which we call the iSTS (implicit synchronous transition systems). The key notions put to work in this notation are essentially borrowed to Pnueli's STS [PSS98] and Dijkstra's guarded command language [Dij76]. In the iSTS, a process consists of simultaneous propositions that manipulate signals. A signal is an infinite flow of values that is sampled by a discrete series of instants or reactions. An event corresponds to the value carried by a signal during a particular reaction or instant. The main features of the iSTS notation are put together in the example of Figure 8.1, that describes the behavior of a counter modulo 2, noted P ($\stackrel{\text{def}}{=}$ means "is defined by"), through a set of simultaneous *propositions*, labeled from (1) to (3).

$$
\begin{array}{lll}
 & \neg s_0 & (1) \\
P \stackrel{\text{def}}{=} \quad | \quad \hat{x}_i \Rightarrow s_{i+1} = \neg s_i & & (2) \\
 | \quad s_i \Rightarrow \hat{x}_{i+1} & & (3)
\end{array}
$$

Figure 8.1: Specification of a counter modulo 2

- The proposition (1) is an invariant. It says that the initial value of the signal s, denoted by s_0, is false. This is specified by the proposition $\neg s_0$ (equivalent to $s_0 = 0$).

- The proposition (2) is a guarded command. It says that if the signal x is present during a given reaction then the value of s is toggled.

  - The leftmost part of the sentence, the proposition $\hat{x}_i$, is a condition or a guard. It denotes the *clock* of x at the instant i. It is true iff the signal x is present during the current reaction.

  - The rightmost part of the sentence, the proposition $s_{i+1} = \neg s_i$, is a transition. The term s_{i+1} refers to the next value of the signal s. The proposition $s_{i+1} = \neg s_i$ says that the next value of s is the negation of the current value of s.

- The proposition (3) is another guarded command. It says that if s_i is true then x_{i+1} is present.

Notice that, in proposition (3), the guard expects the signal s_i to hold the value true but that its action does not actually specify the value of the signal x_{i+1}, it simply requires it to be present. Proposition (3) is hence an *abstraction*: a proposition that partially describes the properties of the system under consideration without implying a particular implementation.

To implement a *function* or a *system*, this proposition needs to be compositionally *refined* by another, saying which value y should hold when present. To this end, one may for instance compose the counter P of Figure 8.1 with the proposition Q:

$$Q \stackrel{\text{def}}{=} (x_{i+1} = x_i + 1 \,|\, x_0 = 0)$$

Notice that the iteration specified by the proposition Q is not guarded. This means that it is an invariant which describes the successive values of the signal x in time but not the particular time samples at which the signal x should occur. The composition of Q with P has the effect of synchronizing the signal x in Q to the *clock* s in P. This composition represents a refinement: the system obeys the specification denoted by the initial proposition P and is constrained to further satisfy the additional requirements of Q.

$$P \text{ is an abstraction of } P \,|\, Q \qquad P \,|\, Q \text{ is a refinement of } P$$

The notions introduced so far hold necessary and sufficient ingredients to specify the behavior of multi-clocked synchronous systems. Pnueli's original STS notation features two additional notions which, in retrospect, are essentially geared towards verification by model-checking.

- One feature is choice $P \vee Q$. For instance, the STS $a = 1 \vee b = 1$ allows to non-deterministically have a or b present with the value true at all time. Non-determinism can equally be modeled using guarded commands and a partially defined signal s whose scope is lexically restricted to the desired proposition; In the iSTS, we write:

  $$R \stackrel{\text{def}}{=} (s \Rightarrow a \,|\, \neg s \Rightarrow b)/s$$

  to mean the non-deterministic proposition of choosing a or b upon the value of an internal signal s, whose calculation is left unspecified. Here, the

notation P/s means that the scope of s is local to the process P. Again, notice that the proposition R may best be understood as the *abstraction* of an executable specification, e.g., one that specifies the actual value of s in time (for instance, the toggle s of the counting process P).

$$R \text{ is an abstraction of } \left(s \Rightarrow a \,|\, \neg s \Rightarrow b \,|\, \left(\neg s^0 \,|\, s' = \neg s\right)\right)/s$$

- Another feature of the STS is explicit absence $\perp$. The proposition $x = \perp$ explicitly specifies that x does not hold a value in the context in which it is considered. For instance, the STS:

$$(a = 1 \wedge x \neq \perp) \vee (x = \perp \wedge b = 1)$$

means that either a is true and x is present or that b is true and x is absent. In the iSTS, this notion is implicit. It can for instance be specified by a refinement of R with the invariant "x is present iff b is true":

$$R \,|\, b = \hat{x}$$

In the aim of moving back and forth from abstraction to refinement, one last essential feature of the iSTS is the notion of scheduling specification. A scheduling specification is designed to imply an order of execution to otherwise purely logical propositions. Whereas a transition, e.g. $s' = \neg s$, implicitly means that the next value of s is computed using the present value of s, a proposition, e.g. $y = x$, just means that x and y are equal. It does not specify any order of execution. An order of execution can be imposed to this proposition by its refinement with a scheduling constraint, noted $y \rightarrow x$. Then,

$$x = y \text{ is an abstraction of } x = y \,|\, y \rightarrow x$$

where $y \rightarrow x$ informally means that "x cannot happen before y". This additional requirement refines the time scale, from one in which x and y happen simultaneously, to a more precise one in which one observes that x cannot happen before y. We will adopt the following syntax, borrowed to the Signal language, and write $x := y$ for an assignment of y to x:

$$(x = y \,|\, y \rightarrow x) \text{ is an abstraction of } x := y$$

8.1.1 Formal Syntax

The formal syntax of the iSTS is defined by the inductive grammar P in Figure 8.2. A process P manipulates boolean values noted $v \in \{0, 1\}$ and signals noted x, y, z. A location l refers to the initial value x^0, the present value x and

the next value x' of a signal. A reference r stands for either a value v or a signal x.

Clock expressions e, f are propositions on boolean values. When true, a clock e defines a particular moment in time. The clocks 0 and 1 denote events that never/always happen. The clock $x = r$ denotes the proposition: "x is present and holds the value r". Particular instances are:

- the clock $\hat{x} \stackrel{\text{def}}{=} (x = x)$ which means that "x is present",
- the clock $x \stackrel{\text{def}}{=} (x = 1)$ which means that "x is true",
- the clock $\neg x \stackrel{\text{def}}{=} (x = 0)$ which means that "x is false".

Clocks are propositions combined using the logical combinators of conjunction $e \wedge f$, to mean that both e and f hold, disjunction $e \vee f$, to mean that either e or f holds, and symmetric difference $e \setminus f$, to mean that e holds and not f.

A process P consists of the simultaneous composition of elementary propositions. 1 is the process that does nothing. The proposition $l = r$ means that "l holds the value r". In the introductory example, we wrote $x' = \neg x$ to mean the proposition $x' = \neg x \stackrel{\text{def}}{=} x = 1 \Rightarrow x' = 0 \,|\, x = 0 \Rightarrow x' = 1$.

The proposition $x \rightarrow l$ means that "l cannot happen before x". The process $e \Rightarrow P$ is a guarded command. It means: "if e is present then P holds". Processes are combined using synchronous composition $P \,|\, Q$ to denote the simultaneity of the propositions P and Q. Restricting a signal name x to the lexical scope of a process P is written P/x.

$$
\begin{array}{rr}
\text{(reference)} & r ::= x \mid v \\
\text{(location)} & l ::= x^0 \mid x \mid x' \\
\text{(clock)} & e, f ::= 0 \mid x = r \mid e \wedge f \mid e \vee f \mid e \setminus f \mid 1 \\
\text{(process)} & P, Q ::= 1 \mid l = r \mid x \rightarrow l \mid e \Rightarrow P \mid (P \,|\, Q) \mid P/x
\end{array}
$$

Figure 8.2: Formal syntax of iSTS algebra

8.1.2 Notational Conventions

In the formal presentation of the iSTS, we restrict ourself to a subset of the elementary propositions in the grammar of Figure 8.2, which we call atoms a:

$$\text{(atoms)} \quad a, b ::= x^0 = v \mid l = y \mid x \rightarrow l \; s.t. \; l ::= x \mid x'$$

Other propositions as well as additional syntactic shortcuts, used in the examples, can be defined by using this restricted subset as follows.

$$l = v \stackrel{\text{def}}{=} (l = x \,|\, x^0 = v \,|\, x' = x)/x \text{ iff } x \neq l \neq x' \qquad \hat{x} \stackrel{\text{def}}{=} (x = x)$$
$$l := x \stackrel{\text{def}}{=} (l = x \,|\, x \rightarrow l) \qquad l \stackrel{\text{def}}{=} (l = 1)$$
$$\hat{x} = \hat{y} \stackrel{\text{def}}{=} (\hat{x} \Rightarrow \hat{y} \,|\, \hat{y} \Rightarrow \hat{x}) \qquad \neg l \stackrel{\text{def}}{=} (l = 0)$$

8.2 A Polychronous Model of Computation

After having seen the notation and formal syntax of the iSTS we present now a short presentation of the polychronous model of computation, the structure of polychrony and the denotational semantics of the iSTS.

The polychronous model of computation, proposed in [LTL03], consists of a unique *domain* of traces, that does not differentiate synchrony from asynchrony, and semi-lattice structures, that render synchrony and asynchrony using specific timing equivalence relations.

We consider a partially-ordered set $(\mathcal{T}, \leq, 0)$ of tags. A tag $t \in \mathcal{T}$ denotes a symbolic instant or a period in time. We note $C \in \mathcal{C}$ a *chain* of $\mathcal{T}$. Events, signals, behaviors and processes are defined starting from tags as follows:

Definition 8.1 (polychrony)
- *An* event $e \in \mathcal{E} = \mathcal{T} \times \mathcal{V}$ *is the pair of a value and a tag.*
- *A* signal $s \in \mathcal{S} = C \rightarrow \mathcal{V}$ *is a function from a* chain *to a set of values.*
- *A* behavior $b \in \mathcal{B}$ *is a function from names* $x \in \mathcal{X}$ *to signals* $s \in \mathcal{S}$*.*
- *A* process $p \in \mathcal{P}$ *is a set of behaviors that have the same domain.*

Figure 8.3 depicts a behavior b in the polychronous domain $\mathcal{P}$.

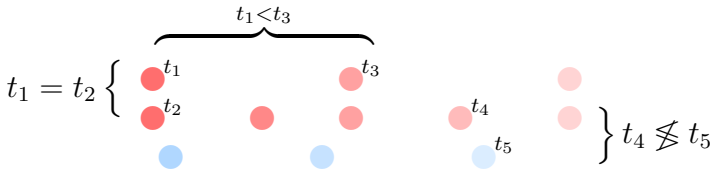


Figure 8.3: A behavior in the polychronous model of computation

Tags t_1 and t_2 are equal, meaning that the events they time are synchronous. Tag t_1 precedes t_3, written $t_1 < t_3$, to mean the scheduling relation that causally relates them in time. The signal at the bottom has no tag comparable to either of the upper two, therefore e.g. $t_4 \not\leqslant t_5$. It denotes a signal belonging to a entirely different clock domain.

Notations. In the remainder, we write tags(s) and tags(b) $= \cup_{x \in \text{vars}(b)}$tags ($b(x)$) for the tags of a signal s and of a behavior b, $b|_X$ for the projection of a behavior b on $X \subset \mathcal{X}$ and $b/X = b|_{\text{vars}(b) \setminus X}$ for its complementary, vars(b) and vars(p) for the domains of b and p.

Synchronous composition. $p \| q$ is defined by the union of all behaviors b (from p) and c (from q) which are synchronous: all signals along the interface $I = \text{vars}(p) \cap \text{vars}(q)$ between p and q carry the same values at the same time tags.

$$p \| q = \{ b \cup c \mid (b, c) \in p \times q, I = \text{vars}(p) \cap \text{vars}(q), b|_I = c|_I \}$$

8.2.1 Scheduling Structure of Polychrony

To render the scheduling relations between events occurring at the same time tag t, we refine the domain of polychrony with scheduling relations. A scheduling $t_x \rightarrow t'_y$ means that the event along the signal named y at t' may not happen before x at t.

Figure 8.4 depicts three scheduling relations superimposed to the signals x and y of Figure 8.3. The scheduling relation $t_x \rightarrow t_y$ denotes the observation that the event occurring along x at t precedes the event along y.

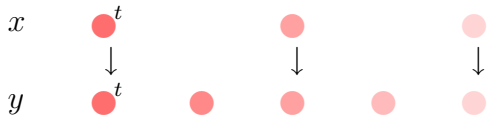


Figure 8.4: Scheduling relations between simultaneous events

The pair t_x of a time tag t and of a signal name x renders the date d of an event along the signal x at the symbolic time t. The tag t itself represents the period during which multiple events take place to form a reaction: the tag t corresponds is the equivalence class of a synchronization relation between dates d, as in the synchronous structures [NTLG99]. The domain of dates $\mathcal{D} = \mathcal{T} \times \mathcal{X}$ of a given behavior b is subject to a pre-order relation $\rightarrow^b$ that denotes scheduling and contains causality $<$. When no ambiguity is possible on the identity of b in a scheduling constraint $x \rightarrow^b y$, we write it $x \rightarrow y$.

$$\forall b \in \mathcal{B}, \forall x \in \text{vars}(b), \forall t, t' \in \text{tags}(b(x)), \quad t < t' \Rightarrow t_x \rightarrow^b t'_x$$
$$t_x \rightarrow^b t'_x \Rightarrow \neg(t' < t)$$

8.2.2 Synchronous Structure of Polychrony

In the previous section, we gave a structural definition of a domain of traces for capturing the possible behavior of processes in the iSTS algebra. Building

upon this domain, we define the semi-lattice structure which relationally denotes synchronous behaviors in this domain.

The intuition behind this relation is depicted Figure 8.5. It is to consider a signal as an elastic with ordered marks on it (tags). If the elastic is stretched, marks remain in the same relative and partial order but have more space (time) between each other.

The same holds for a set of elastics: a behavior. If elastics are equally stretched, the order between marks is unchanged. In the Figure 8.5, the time scale of x and y change but the partial timing and scheduling relations are preserved. Stretching is a partial-order relation which defines clock equivalence (definition 8.2).

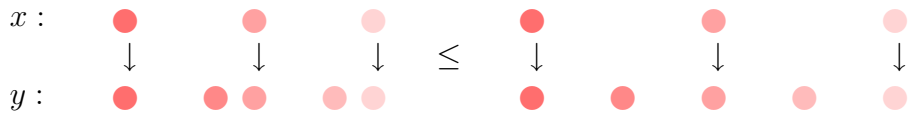


Figure 8.5: Relating synchronous behaviors by stretching.

Definition 8.2 (clock equivalence)

A behavior c *is a* stretching *of* b*, written* $b \leq c$*, iff* $\text{vars}(b) = \text{vars}(c)$ *and there exists a bijection* f *on* $\mathcal{T}$ *which satisfies*

$$\begin{array}{l} \forall t, t' \in \text{tags}(b), t \leq f(t) \wedge (t < t' \Leftrightarrow f(t) < f(t')) \\ \forall x, y \in \text{vars}(b), \forall t \in \text{tags}(b(x)), \forall t' \in \text{tags}(b(y)), t_x \rightarrow^b t'_y \Leftrightarrow f(t)_x \rightarrow^c f(t')_y \\ \forall x \in \text{vars}(b), \text{tags}(c(x)) = f(\text{tags}(b(x))) \wedge \forall t \in \text{tags}(b(x)), b(x)(t) = c(x)(f(t)) \end{array}$$

b *and* c *are* clock-equivalent*, written* $b \sim c$*, iff there exists a behavior* d *s.t.* $d \leq b$ *and* $d \leq c$*.*

8.2.3 Denotational Semantics of the iSTS Algebra

The detailed presentation and extension of the polychronous model of computation allows to give a denotational model to the iSTS notation introduced in section 8.1. This model consists of relating a proposition P to the set of behaviors p it denotes.

Meaning of clocks. Let us start with the denotation of a clock expression e (Figure 8.6). The meaning $[\![e]\!]_b$ of a clock e is defined relatively to a given behavior b and consists of the set of tags satisfied by the proposition e in the behavior b.

In Figure 8.6, the meaning of the clock $x = v$ (resp. $x = y$) in b is the set of tags $t \in \text{tags}(b(x))$ (resp. $t \in \text{tags}(b(x)) \cap \text{tags}(b(y))$) such that $b(x)(t) = v$ (resp.

$b(x)(t = b(y)(t))$. In particular, $[\![\hat{x}]\!]_b = \text{tags}(b(x))$. The meaning of a conjunction $e \wedge f$ (resp. disjunction $e \vee f$ and difference $e \setminus f$) is the intersection (resp. union and difference) of the meaning of e and f. Clock 0 has no tags.

$$
\begin{array}{ll}
[\![x = v]\!]_b = \{t \in \text{tags}(b(x)) \mid b(x)(t) = v\} & [\![0]\!]_b = \emptyset \\
[\![x = y]\!]_b = \{t \in \text{tags}(b(x)) \cap \text{tags}(b(y)) \mid b(x)(t) = b(y)(t)\} & \\
[\![e \wedge f]\!]_b = [\![e]\!]_b \cap [\![f]\!]_b & \\
[\![e \vee f]\!]_b = [\![e]\!]_b \cup [\![f]\!]_b & \\
[\![e \setminus f]\!]_b = b[\![e]\!]_b \setminus [\![f]\!]_b & \\
[\![1]\!]_b = \text{tags}(b) &
\end{array}
$$

Figure 8.6: Denotational semantics of clock expressions

Meaning of propositions. The denotation of a clock expression by a set of tags yields the denotational semantics of propositions P, written $[\![P]\!]$, Figure 8.7. The meaning $[\![P]\!]^e$ of a proposition P is defined with respect to a clock expression e. Where this information is absent, we assume $[\![P]\!] = [\![P]\!]^1$ to mean that P is an invariant (and is hence independent of a particular clock).

The meaning of an initialization $x^0 = v$, written $[\![x^0 = v]\!]^e$, consists of all behaviors defined on x, written $b \in \mathcal{B}|_x$ such that the initial value of the signal $b(x)$ equals v. Notice that it is independent from the clock expression e provided by the context. In Figure 8.7, we write:

- $\mathcal{B}|_X$ for the set of all behaviors of domain X
- $\min(C)$ for the minimum of the chain of tags C
- $\text{succ}_t(C)$ for the immediate successor of t in the chain C
- $\text{vars}(P)$ and $\text{vars}(e)$ for the set of free signal names of P and e.

The meaning of a proposition $x = y$ at the clock e consists of all behaviors b defined on $\text{vars}(e) \cup \{x, y\}$ such that all tags $t \in [\![e]\!]_b$ at the clock e belong to $b(x)$ and $b(y)$ and are associated with the same value. A scheduling specification $y \rightarrow x$ at the clock e denotes the set of behaviors b defined on $\text{vars}(e) \cup \{x, y\}$ which, for all tags $t \in [\![e]\!]_b$, requires x to precede y: if t is in $b(x)$ then it is necessarily in $b(y)$ and satisfies $t_y \rightarrow^b t_x$. The propositions $x' = y$ and $y \rightarrow x'$ is interpreted similarly by considering the tag t' that is the successor of t in the chain C of x.

The behavior of a guarded command $f \Rightarrow P$ at the clock e is equal to the behavior of P at the clock $e \wedge f$. The meaning of a restriction P/x consists of the behaviors c of which a behavior b/x from P are a stretching of. The behavior of $P \,|\, Q$ consists the synchronous composition of the behaviors of P and Q.

$$
\begin{aligned}
[\![x^0 = v]\!]^e &= \{b \in \mathcal{B}|_x \mid b(x)(\min(\mathrm{tags}(b(x)))) = v\} \\
[\![x = y]\!]^e &= \{b \in \mathcal{B}|_{\mathrm{vars}(e)\cup\{x,y\}} \mid \forall t \in [\![e]\!]_b, \\
&\qquad t \in \mathrm{tags}(b(x)) \wedge t \in \mathrm{tags}(b(y)) \wedge b(x)(t) = b(y)(t)\} \\
[\![y \to x]\!]^e &= \{b \in \mathcal{B}|_{\mathrm{vars}(e)\cup\{x,y\}} \mid \forall t \in [\![e]\!]_b, \\
&\qquad t \in \mathrm{tags}(b(x)) \Rightarrow t \in \mathrm{tags}(b(y)) \wedge t_y \to^b t_x\} \\
[\![x' = y]\!]^e &= \{b \in \mathcal{B}|_{\mathrm{vars}(e)\cup\{x,y\}} \mid \forall t \in [\![e]\!]_b, \\
&\qquad t \in C = \mathrm{tags}(b(x)) \wedge t \in \mathrm{tags}(b(y)) \wedge b(x)(\mathrm{succ}_t(C)) = b(y)(t)\} \\
[\![y \to x']\!]^e &= \{b \in \mathcal{B}|_{\mathrm{vars}(e)\cup\{x,y\}} \mid \forall t \in [\![e]\!]_b, \\
&\qquad t \in C = \mathrm{tags}(b(x)) \Rightarrow t \in \mathrm{tags}(b(y)) \wedge t_y \to^b (\mathrm{succ}_t(C))_x\}
\end{aligned}
$$

Figure 8.7: Denotational semantics of propositions

$$
\begin{aligned}
[\![f \Rightarrow P]\!]^e &= [\![P]\!]^{e \wedge f} \\
[\![P \,|\, Q]\!]^e &= [\![P]\!]^e \,|\, [\![Q]\!]^e \\
[\![P/x]\!]^e &= \{c \leq b/x \mid b \in [\![P]\!]^e\}
\end{aligned}
$$

Figure 8.8: Denotational semantics of propositions

8.3 The SIGNAL Language

We have now seen the theoretical constructs and bases of Polychrony, but for simplicity and generality we described it in the iSTS formalism. The Polychrony workbench, however, is using a different formalism as input notation, namely the SIGNAL [BLJ91] language. The polychronous model of computation is implemented by the multi-clocked synchronous data-flow notation SIGNAL. It also serves as the specification formalism used for the case study in Section 12.2. We therefore give at this point a short introduction to SIGNAL, its basic constructs and some peculiarities. In Section 8.4 we then show that the SIGNAL formalism and the iSTS formalism can be used interchangeably and detail how they can be translated in each direction.

In SIGNAL, a process P consists of the composition of simultaneous equations $x := f(y, z)$ or $x := y\, f\, z$ over input signals y, z and output signals x. A signal $x \in \mathcal{X}$ is a possibly infinite flow of values $v \in \mathcal{V}$ sampled at a clock noted $\hat{}x$.

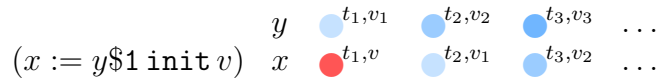
$$P, Q ::= x := y\, f\, z \mid P/x \mid P \,|\, Q \qquad \text{(SIGNAL process)}$$

In the polychronous model of computation, Section 8.2, the denotation of a clock $\hat{}x$ is the domain of the signal associated to x: a chain of tags. We note $[\![P]\!]$ for the denotation of a process P. The synchronous composition of processes $P \,|\, Q$ consists of the simultaneous solution of the equations in P and in Q. The process

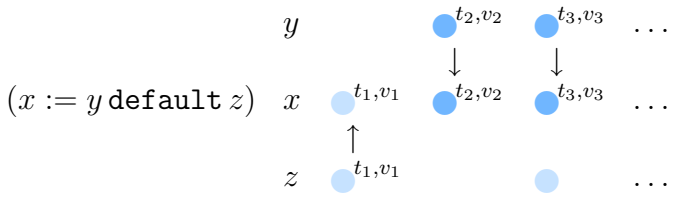
P/x restricts the signal x to the lexical scope of P.

$$[\![P \,|\, Q]\!] = [\![P]\!] \,|\, [\![Q]\!] \text{ and } [\![P/x]\!] = [\![P]\!]/x$$

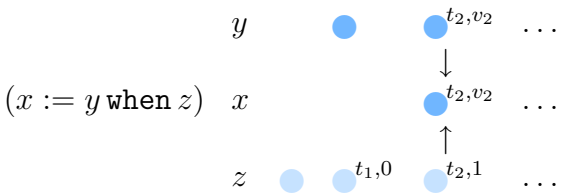
An equation $x := y\ f\ z$ denotes a relation between the input signals y and z and an output signal x by a combinator f. An equation is usually a ternary and infixed relation noted $x := y\ f\ z$ but it can in general be an $m+n$-ary relation noted $(x_1, \ldots x_m) := f(y_1, \ldots y_n)$. SIGNAL requires three primitive combinators to perform delay $x := y\$1\ \texttt{init}\ v$, sampling $x := y\ \texttt{when}\ z$ and merge $x = y\ \texttt{default}\ z$. The equation $x := y\$1\ \texttt{init}\ v$ initially defines the signal x by the value v and then by the previous value of the signal y. The signal y and its delayed copy $x := y\$1\ \texttt{init}\ v$ are synchronous: they share the same set of tags $t_1, t_2, \ldots$. Initially, at t_1, the signal x takes the declared value v and then, at tag t_n, the value of y at tag t_{n-1}.



The equation $x := y\ \texttt{default}\ z$ defines x by y when y is present and by z otherwise. If y is absent and z present with v_1 at t_1 then x holds (t_1, v_1). If y is present (at t_2 or t_3) then x holds its value whether z is present (at t_2) or not (at t_3).



The equation $x := y\ \texttt{when}\ z$ defines x by y when z is true (and both y and z are present); x is present with the value v_2 at t_2 only if y is present with v_2 at t_2 and if z is present at t_2 with the value true. When this is the case, one needs to schedule the calculation of y and z before x, as depicted by $y_{t_2} \rightarrow x_{t_2} \leftarrow z_{t_2}$.



8.3.1 Relating Polychronous Signals with Clocks

In SIGNAL, the presence of a value along a signal x is the proposition noted $\hat{}x$ that is true when x is present and that is absent otherwise. The syntax of

clock expressions e and clock relations E is a particular subset of SIGNAL that is defined by the induction grammar e. The clock expression $\hat{}x$ can be defined by the Boolean operation $x = x$ (i.e. $y := \hat{}x \stackrel{\text{def}}{=} y := (x = x)$). Referring to the polychronous model of computation, it represents the set of tags at which the signal holds a value. Clock expressions naturally represent control, the clock $[x]$ represents the time tags at which the Boolean signal x is present and true (i.e. $y := [x] \stackrel{\text{def}}{=} y := 1\,\texttt{when}\,x$). The clock $[\texttt{not}\,x]$ represents the time tags at which the Boolean signal x is present and false. We write 0 for the empty clock (it has no tags).

$$e ::= \hat{}x \mid [x] \mid [\texttt{not}\,x] \mid e\,\hat{}\texttt{+}\,e' \mid e\,\hat{}\texttt{-}\,e' \mid e\,\hat{}\texttt{*}\,e' \mid 0$$

A clock constraint E is a SIGNAL process. The constraint $e\hat{}=e'$ synchronizes the clocks e and e'. It corresponds to the process $(x := (e = e'))/x$. Composition $E \,|\, E'$ corresponds to the union of constraints and restriction E/x to the existential quantification of E by x. A transitive scheduling constraint $x \rightarrow y\,\texttt{when}\,e$ specifies the order of execution between x and y at the clock e.

$$E ::= () \mid e\hat{}=e' \mid e\hat{}<e' \mid x \rightarrow y\,\texttt{when}\,e \mid E \,|\, E' \mid E/x$$

Each process P corresponds to a clock constraint E defined by the clock inference system $P : E$ of Figure 8.9.

$$x := y\$1\,\texttt{init}\,v : \hat{}x\hat{}=\hat{}y$$
$$x := y\,\texttt{when}\,z : \hat{}x\hat{}=\hat{}y\,[z] \,|\, y \rightarrow x\,\texttt{when}\,z$$
$$x := y\,\texttt{default}\,z : \hat{}x\hat{}=\hat{}y\,\hat{}\texttt{+}\,\hat{}z \,|\, z \rightarrow x\,\texttt{when}\,(\hat{}z\,\hat{}\texttt{-}\,\hat{}y) \,|\, y \rightarrow x\,\texttt{when}\,\hat{}x$$

$$\frac{P : E \quad Q : E'}{P \,|\, Q : E \,|\, E'} \qquad \frac{P : E}{P/x : E/x}$$

Figure 8.9: Clock inference system

8.3.2 Code Generation via Hierarchization

The clock constraints E of a process P hold the necessary information to generate a sequential control-flow graph starting from a multi-clocked synchronous specification by a technique of hierarchization, proposed in [ABL95]. It can be outlined by considering a simple SIGNAL program, Figure 8.10. Process `buffer` implements two functionalities. One is the process `current`. It defines a `cell` in which values are stored at the input clock `^i` and loaded at the output clock `^o`. `cell` is a predefined SIGNAL operation defined by:

```
x:=y cell z init v  def=  (| m:=x$1 init v
                          | x:= y default m
                          | ^x ^= ^y ^+ ^z
                          |) /m
```

The other functionality is the process `alternate` that desynchronizes the signals `i` and `o` by synchronizing them to the true and false values of an alternating Boolean signal `b`.

```
process buffer = (? i ! o)
  (| alternate (i, o) | o := current (i)
   |) where
  process alternate = (? i, o ! )
    (| zb := b$1 init true
     | b := not zb
     | o ^= when not b
     | i ^= when b
     |) / b, zb;
  process current = (? i ! o)
    (| zo := i cell ^o init false
     | o  := zo when ^o
     |) / zo;
```

Figure 8.10: Polychronous specification of a buffer

Clock inference (Figure 8.11) applies the clock inference system of Figure 8.9 to the process `buffer` to determine three synchronization classes. We observe that `b, c_b, zb,` and `zo` are synchronous and define the master clock synchronization class of `buffer`. There are two other synchronization classes, `c_i` and `c_o`, that correspond to the true and false values of the Boolean flip-flop variable `b`, respectively.

This defines three nodes in the control-flow graph of the generated code shown in Figure 8.12. At the main clock `c_b`, `b`, and `c_o` are calculated from `zb`. At the sub-clock `b`, the input signal `i` is read. At the sub-clock `c_o` the output signal `o` is written. Finally, `zb` is determined. Notice that the sequence of instructions follows the scheduling constraints determined during clock inference.

8.3.3 Some More Concrete Syntax

In addition to the core syntax of SIGNAL presented so far, we make extensive use of process declarations and partial equations for the purpose of modeling our case study. In SIGNAL, a partial equation $x ::= y\ f\ z$ `when` e is the partial definition

```
(| c_b ^= b
 | b   ^= zb
 | zb  ^= zo
 | c_i := when b
 | c_i ^= i
 | c_o := when not b
 | c_o ^= o
 | i -> zo when ^i
 | zb -> b
 | zo -> o when ^o
 |) / zb, zo, c_b, c_o, c_i, b;
```

Figure 8.11: Clock analysis of the buffer

```
buffer_iterate () {
  b = !zb;
  c_o = !b;
  if (b) {
    if (!r_buffer_i(&i))
      return FALSE;
  }
  if (c_o) {
    o = i;
    w_buffer_o(o);
  }
  zb = b;
  return TRUE;
}
```

Figure 8.12: Buffer code generation

of the variable x by the operation $y\, f\, z$ at the clock denoted by the expression e. The default equation $x ::= \texttt{defaultvalue}\, v$ defines the value of the variable x when it is present but no corresponding partial equation $x ::= y\, f\, z\, \texttt{when}\, e$ applies (because e is absent). Let x be a variable defined using n partial equations and a default value v:

$$\begin{array}{l} x ::= x_1\, \texttt{when}\, e_1 \\ \vdots \\ \mid\ x ::= x_n\, \texttt{when}\, e_n \\ \mid\ x ::= \texttt{defaultvalue}\, v \end{array}$$

The SIGNAL compiler processes this definition by first checking the clock expressions $e_1, \ldots e_n$ mutually exclusive and then handling the definition as the equivalent equation: $x := (x_1\, \texttt{when}\, e_1)\ \texttt{default} \ldots (x_n\, \texttt{when}\, e_n)\, \texttt{default}\, v$. The declaration of a process P of name f, input signals $x_1..x_m$, output signals $x_{m+1}..x_n$ is noted $\texttt{process}\, f = (?\, x_1, \ldots x_m\, !\, x_{m+1}, \ldots x_n)\, (|\, P\, |)$;. Once declared, process f may be called with its actual parameters $y_1..y_n$ by $(y_{m+1}, \ldots y_n) := f(y_1, \ldots y_m)$ and behave as P with $x_1..x_n$ substituted by $y_1..y_n$. A variant declaration is that of a foreign function f, accessible, e.g. from a separately compiled C library. Its call can be wrapped into SIGNAL by declaring its interface and by declaring an abstraction E of its behavior, which consists of scheduling and clock constraints.

$$\begin{array}{l} \texttt{process}\, f = (?\, x_1, \ldots x_m\, !\, x)\, \texttt{spec}\, (|\, E\, |) \\ \qquad\qquad \texttt{pragmas C\_CODE"\&x} = \texttt{f(\&x}_1, \ldots \texttt{\&x}_m\texttt{)"} \\ \qquad\qquad \texttt{end pragmas}; \end{array}$$

8.4 Translating iSTS into SIGNAL

In this section we describe how to transform the iSTS notation into SIGNAL and vice versa. Any SIGNAL equation $x := f(y, r)$ denotes a relation between the input signals (y, r) and the output signal x by a function or combinator f (Figure 8.13).

$$\begin{array}{lrcl} \text{(process)} & P & ::= & x := f(y,r) \mid (P \,|\, Q) \mid P/x \\ \text{(combinator)} & f & \in & \{\$1\, \mathsf{init}\, v \mid v \in \mathcal{V}\} \cup \{\mathsf{when}, \mathsf{default}, \ldots\} \end{array}$$

Figure 8.13: Signal syntax core

Signal has three primitive operators: the equation $x := y\$1\, \mathsf{init}\, v$ initially defines x by v and then by the previous value of y in time, the equation $x := y\, \mathsf{when}\, z$ defines x by y when z is true and the equation $x := y\, \mathsf{default}\, z$ defines x by y when y is present and by z otherwise. The synchronous composition $P \,|\, Q$ of two processes P and Q consists of the simultaneous solution of the system of equations

$$
\begin{aligned}
\llbracket x := y \,\mathsf{when}\, z \rrbracket &= (z \Rightarrow x := y) \\
\llbracket x := y \,\mathsf{default}\, z \rrbracket &= (\hat{y} \Rightarrow x := y) \,|\, (\hat{z} \setminus \hat{y} \Rightarrow x := z) \\
\llbracket x := y\$1 \,\mathsf{init}\, v \rrbracket &= (x^0 = v \mid x' := y) \\
\llbracket P \,|\, Q \rrbracket &= \llbracket P \rrbracket \,|\, \llbracket Q \rrbracket \\
\llbracket P/x \rrbracket &= \llbracket P \rrbracket / x
\end{aligned}
$$

Figure 8.14: From Signal to iSTS $\llbracket P \rrbracket$

P and Q. Figure 8.14 associates Signal equations with the iSTS notation, showing their close relationship within the polychronous model of computation.

$$
\begin{aligned}
\lfloor x^0 = v \rfloor &= x := x'\$1 \,\mathsf{init}\, v \\
\lfloor l := r \rfloor^e &= l ::= r \,\mathsf{when}\, \lfloor e \rfloor \\
\lfloor x \to l \rfloor^e &= x \to l \,\mathsf{when}\, \lfloor e \rfloor \\
\lfloor P \,|\, Q \rfloor^e &= \lfloor P \rfloor^e \,|\, \lfloor Q \rfloor^e \\
\lfloor e \Rightarrow P \rfloor^f &= \lfloor P \rfloor^{e \wedge f} \\
\lfloor P/x \rfloor^e &= \lfloor P \rfloor^e / x \\
\lfloor v \rfloor &= v \\
\lfloor \hat{x} \rfloor &= \hat{}x \\
\lfloor l = r \rfloor &= l = r \\
\lfloor e \wedge f \rfloor &= \lfloor e \rfloor \,\mathtt{when}\, \lfloor f \rfloor \\
\lfloor e \vee f \rfloor &= \lfloor e \rfloor \,\mathtt{default}\, \lfloor f \rfloor \\
\lfloor e \setminus f \rfloor &= \mathtt{not}\, \lfloor f \rfloor \,\mathtt{default}\, \lfloor e \rfloor
\end{aligned}
$$

Figure 8.15: From iSTS to Signal $\lfloor P \rfloor$

Figure 8.15 shows the other side of the transformation between Signal and the iSTS. It defines an encoding of propositions by using partial equations $x ::= y \,\mathsf{when}\, z$, an additional feature of the Polychrony workbench, whose meaning match that of propositions $z \Rightarrow x := y$. The iSTS supports therefore a direct translation into the data-flow notation Signal of the Polychrony workbench which we will use in the following chapters.

Chapter 9

Behavioral Types for SystemC

Type systems have led to important improvements in software development, productivity, and design quality. In component based design, types are used to detect mismatches in component interfaces and makes sure that components are compatible. Interface mismatches however can happen at different levels. One level is the data type. When component A is expecting an integer at a certain input signal, but the connected component B is sending a string, the data types of the signals do not match and in consequence the system will most probably not work as expected. Most general purpose languages implement data types and can check for such type mismatches at compile time. Another level where interface mismatches can occur is for example the temporal level. If component A is expecting input values at a certain rate, but component B is producing values and sending them to A twice as fast, the system will again not work as foreseen, however conventional data type checks will not find a problem in the component compatibility. There are many more levels of mismatches in component compositions that can occur, such as the value range of signals, the clocks, or the synchronizations of signals and even more complex component interaction patterns. For all of these possible interaction problems, we can describe types and type systems that check if they are respected. In general these advanced types can be summarized under the term *behavioral types*, as they describe not only the data types on component interfaces but to a certain extent the behavior of the component interactions.

With the tools from the previous chapter we are now equipped with the required mathematical framework and formal methodology to address the modeling of GALS architectures described using SystemC. In the following we present a model that is described in terms of a type inference system and extended to the structuring elements of SystemC. The bases of this module system have been developed in [TBS+04a], [TG04], and [TBS+04b]. The framework allows to give a behavioral signature of system components to compositionally check their

correct composition when building a desired architecture. It also optimizes the described software elements by, first, extracting the formal model from the functional architecture description and, second, using the model to regenerate an optimized software, matching the requirements of the execution architecture. As a by-product, associating types with SystemC programs provide a formal denotational semantics implied by the interpretation of types in the polychronous model of computation.

9.1 Example and Overview

To allow for an easier grasp on the proposed behavioral type inference technique, we outline the analysis of a small fragment of a SystemC program, Figure 9.1, the construction of its dynamic behavioral type, Figure 9.2, and the inference of its static abstraction, Figure 9.3. Then we elaborate the notion of proof obligations synthesis by giving a brief outline of the design correctness issues which can be modeled and checked in the framework of our type system.

9.1.1 Static Single Assignment

Figure 9.1 (left) depicts a simple C code fragment consisting of an iterative program that counts the number of bits set to one in the variable idata. While idata is not equal to zero, it adds its right-most bit to an output count variable ocnt and shifts it right in order to process the next bit. In the static single-assignment (SSA) representation of the program Figure 9.1 (right), all variables (idata and ocnt) are read and written at most once per cycle. Each time a variable is assigned a new value, SSA is changing its name, this is to make sure that a variable is only assigned once. In the example in Figure 9.1 we rename the variables that are used multiple time by adding an underscore and a number, for example `idata_1` to the end of the variable name.

Label `L2` is the entry point of the SSA block that represents the while loop. The first instruction is a ϕ-node. In a ϕ-node, several execution strings are merged together into one. In this case, the value of `idata` may come from the block `L1`, or from the block `L3`. The ϕ-node assigns to the new variable `idata_3` the value of the variable that corresponds to the preceding execution string. This is `idata_1` for the first execution, and `idata_2` for all succeeding executions. The same happens for the `ocnt` variable, that also has two possible sources. In the third line of block `L2`, we store the result of the loop condition into the temporary variable `cond`. This is needed as SSA only has three address instructions that read two operands at a time and write a third. The last line of block `L2` then checks the condition for the loop. If it is true, control passes to block `L3`, otherwise it goes on to the following block in the code.

(C source)

```
while (idata != 0)
{
  ocnt= ocnt + (idata &1);
  idata= idata >> 1;
}
```

(SSA code)

```
L1: ...
    goto L2

L3: tmp = idata_3 & 1;
    ocnt_2 = ocnt_3 + tmp;
    idata_2 = idata_3 >> 1;

L2: idata_3 = φ(idata_1,idata_2);
    ocnt_3 = φ(ocnt_1,ocnt_2);
    cond = (idata_3 != 0);
    if cond then goto L3

L4: ...
```

Figure 9.1: Translation of a source program into static single assignment form

While the block `L2` is taking care of the control of the loop, block `L3` contains the actual logic of the loop. It is mostly the same as the C version of the example, only that a temporary variable is used to store an intermediate result for a code line with two operators. Also appropriate indices are added to the variables ocnt and idata in order to comply to the SSA rules and manage the different versions of the variables. At the end of the execution of `L3`, control passes implicitly back to the `L2` which is the following block in the code.

9.1.2 Propositional Behavior

Although verbose, the SSA intermediate representation of an imperative program can present an otherwise arbitrarily obfuscated C program in a form that can be easily manipulated by an automatic program analyzer. Figure 9.2 shows the SSA code of our example on the left along with the corresponding behavioral type on the right hand side. The behavioral type consists of the simultaneous composition of logical propositions that form a synchronous transition system. Each proposition is associated with one instruction: it specifies its invariants. In particular, it tells when the instruction is executed, what it computes, when it passes control to the next statement, and when it branches to another block. The extent of a proposition is for the duration of a reaction. A reaction can last for an arbitrarily long period of time provided that it is finite and that every variable or register changes its value at most once during this period.

In the first line of block `L3` for instance, we associate the SSA instruction `tmp = idata_3 & 1;` to the proposition $x_{L3} \Rightarrow$ `tmp:= idata & 1`. In this proposition, the variable x_{L3} is a boolean that is true if and only if statements of block

(SSA code)

```
1  L1: ...
       goto L2
3
   L3: tmp = idata_3 & 1;
5      ocnt_2 = ocnt_3 + tmp;
       idata_2 = idata_3 >> 1;
7      goto L2

9  L2: idata_3 = φ(idata_1,idata_2);
       ocnt_3 = φ(ocnt_1,ocnt_2);
11     cond = (idata_3 != 0);
       if cond then goto L3
13     else goto L4

15 L4: ...
```

(behavioral type)

```
1  x_L1 ⇒ ...
   x_L1 ⇒ x'_L2
3
   x_L3 ⇒ tmp := idata & 1
5  x_L3 ⇒ ocnt' := ocnt + tmp
   x_L3 ⇒ idata' := idata >> 1
7  x_L3 ⇒ x'_L2

9


11 x_L2 ⇒ cond := (idata != 0)
   x_L2 ⇒ cond⇒ x'_L3
13 x_L2 ⇒ ¬cond⇒ x'_L4

15 x_L4 ⇒ ...
```

Figure 9.2: Propositional behavior of the SSA program

L3are being executed. So, this proposition says that each time the label L3 is executed the temporary variable tmp is assigned the value (idata & 1). If L3 is not active, then another proposition may hold. All propositions corresponding to a statement from the SSA block L3 are conditioned by x_{L3} meaning that they hold when L3is being executed. Another characteristic of this behavioral type is that all propositions of one block can be evaluated in parallel.

In order to avoid collisions, time is advanced between the blocks. $x_{L3} \Rightarrow x'_{L2}$ means that x_{L2} will be activated in the reaction following the activation of block x_{L3}. In the behavioral type we can assign a variable multiple times in the program, as long as it is assigned only once during each reaction. This removes the need for the renaming of the variables. The exclusiveness in time of the reactions makes sure that there is no conflict. As all propositions are evaluated in parallel, the passing of one block to another has to be explicited. While in Figure 9.1 control passes implicitly from block L3 to block L2 and from block L2 to L4, in Figure 9.2 these transitions are noted as they have to be specified for the behavioral type. Branches of control in the propositional behavior are implemented with the help of conditions. Block x_{L2} represents a simple conditional branch. First the condition is evaluated in line 11, then, if this condition is true block x_{L3} is activated in line 12, otherwise the execution continues in block x_{L4} as specified in line 13. As there is no variable renaming, there is also no need for the ϕ statements from the SSA.

9.1.3 Static Abstraction

We have seen that every instruction of an SSA program could be associated with a proposition to render its control-flow and data-flow behaviors. This representation provides a formal and expressive way to model, analyze, optimize, and verify the behavior of ordinary programs for example in C/C++.

To ease both optimization and verification of such programs based on that representation, we abstract it over its control flow, characterized by boolean relations between *clocks*, and its data flow, characterized by scheduling relations between *signals*.

Let us first recall some terminology. A clock $\hat{x}$ is associated with a signal x. The signal x corresponds to the flow of the successive values of a variable, sampled by the discrete periods of time that we call reactions. The clock of $\hat{x}$ denotes that set of periods or instants.

In Figure 9.3, all operations on integers and bits reported in the behavioral type on the left hand side of the figure have been abstracted by boolean relations between clocks in the middle column of the figure, and by scheduling relations on the right hand side. This is in fact sufficient information to reconstruct the entire control and data flow graphs of the program. All the abstracted information essentially consists of computations which can be used to decorate the control and data flow graphs and regenerate the original program.

For instance, the instruction $x_{\mathsf{L3}} \Rightarrow tmp := idata\&1$ is abstracted by the type $x_{\mathsf{L3}} \Rightarrow \hat{tmp} = \hat{idata}$. It means: "when the block L3 is executed, tmp is present iff idata is present". The scheduling constraints give an indication on the dependencies of the variables. In line 6 `cond` is assigned a new value that is used in lines 7 and 8. The scheduling abstraction will reflect this, by noting in line 6 $x_{L2} \Rightarrow \texttt{idata} \rightarrow \texttt{cond}$, meaning that the value of `cond` depends on the value of `idata`, and in line 7 and 8 of the scheduling it is expressed that the values of x'_{L3} and x'_{L4} depend on the value of `cond`, and therefore these instructions have to be scheduled later.

The type associated with the whole loop example uses the clocks denoted by the booleans x_{L1}, x_{L2}, x_{L3}, and x_{L4}. Each clock denotes a branch in the control-flow graph of the program. The other clocks, e.g. $\hat{\mathsf{ocnt}}$, denote the presence of data. They are partially related to the "*label*" clocks such as x_{L2}. This means that each label clock is a superset of all the instants that occur in the variables of this block.

9.1.4 Typed Modules

Chapter 10 develops the use of the dynamic or static information provided by the behavioral type inference system to perform design correctness checks. The

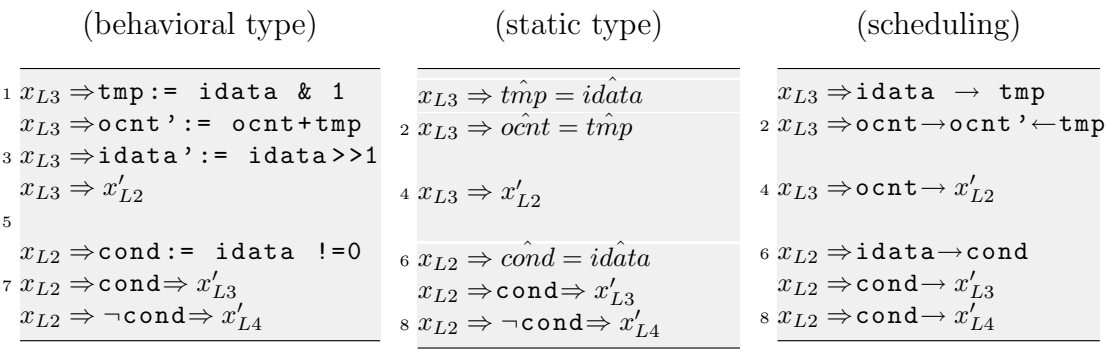


Figure 9.3: Static abstraction of the behavioral type

most salient feature of the behavioral type system is yet the capability to reduce compositional design correctness verification to the validation of synthesized proof obligations. It is presented in the context of the inference system proposed for the SystemC module system, Section 9.4.

As an example, consider a class whose virtual fields are two clocks x and y, and a procedure f. It defines an interface, named m_0, which may be used to type another class. Next, assume an explicit behavioral type declaration $\#\mathsf{TYPE}(f, Q)$ which associates the procedure f with a description of its behavior: the proposition Q. Its aim is to associate the virtual class field f, a method, to the denotation of all possible implementations satisfying an expected functionality.

$$
\begin{array}{l}
\mathsf{class}\, m_0 \,\{ \\
\quad \mathsf{virtual}\, \mathsf{sc\_clock}\, x; \\
\quad \mathsf{virtual}\, \mathsf{sc\_clock}\, y; \\
\quad \mathsf{virtual}\, \mathsf{void}\ f()\ \{\}\ \#\mathsf{TYPE}(f, Q) \\
\};
\end{array}
$$

Next, we associate the interface m_0 with the class parameter m_1 of a template class m_2. The interface m_0 now gives a behavioral type to the method f in the class parameter m_1 expected by the module m_2. Indeed, the template class m_2 uses the class parameter m_1, that implements m_0, to launch a thread $m_1.f$ sensitive to x. The behavioral type Q, which gives an assumption on the behavior of $m_1.f$, is required to provide a guarantee on the behavior of the module m_2, produced by the template class.

$$
\begin{array}{l}
\mathsf{template}\, \langle \mathsf{class}\, m_1 \rangle\ \#\mathsf{TYPE}(m_1, m_0) \\
\quad \mathsf{SC\_MODULE}(m_2)\ \{\mathsf{SC\_CTOR}(m_2)\ \{ \\
\qquad \mathsf{SC\_THREAD}(m_1.f)\, \mathsf{sensitive} \ll x \\
\quad \} \\
\};
\end{array}
$$

9.1.5 Proof Obligations

Let m_3 be a candidate parameter for the template class m_2. It structurally implements the interface m_0, because it provides the clocks x and y and defines the method f by the program *pgm*. Using the type inference technique previously outlined, the program *pgm* is associated with a proposition P that describes its behavioral type, and the class m_3 be decorated with the corresponding type declaration $\#\mathsf{TYPE}(f, P)$.

$$\begin{array}{l}\mathsf{class}\, m_3\,\{\\ \quad \mathsf{sc\_clock}\, x; \mathsf{sc\_clock}\, y;\\ \quad \mathsf{void}\ f()\ \{\ pgm\}\ \#\mathsf{TYPE}(f, P)\\ \};\end{array}$$

Then, let m_4 be the class defined by the instantiation of the template m_2 with the actual parameter m_3. To check the compatibility of the actual parameter m_3 with the formal parameter m_0, we need to establish the containment of the behaviors denoted by the proposition P (the behavioral type of the actual parameter) in the denotation of the proposition Q, the type abstraction declared in m_0.

$$m_2\langle m_3\rangle\, m_4 \text{ is type-safe iff } \models P \Rightarrow Q$$

This amounts to checking that P implies Q. This proof obligation can either be implemented using model checking (if P and Q are *dynamic* interfaces) or using SAT checking, if Q is a *static* interface, by calculating the static abstraction $\hat{P}$ of P and by verifying that $\hat{P}$ implies $\hat{Q}$.

9.2 Formal Syntax of the SystemC Core

We start with the definition of the core of the SystemC syntax relevant to the present study. A system *sys* consists of the composition of classes and modules (Figure 9.4). A class declaration $\mathsf{class}\, m\,\{decl\}$ associates a class name m with a sequence of fields *decl*. It is optionally parameterized by a class with $\mathsf{template}\,\langle\mathsf{class}\, m_1\rangle$. To enforce a strong typing policy, we annotate the class parameter m_1 with $\#\mathsf{TYPE}(m_1, m_2)$ to denote the type of m_1 with the virtual class m_2. A module $\mathsf{SC\_MODULE}(m)$ is a class that defines an architecture component. Its constructor $\mathsf{SC\_CTOR}(m)$ $\{new; pgm\}$ allocates threads (e.g. $\mathsf{SC\_THREAD}(f)$) and executes an initialization program *pgm*. While modules are sequentially declared in the program text, they define threads whose execution is concurrent.

Declarations *decl* associate locations x with native classes or template class instances $m\langle m^*\rangle$ (* being the actual parameter of the template class), and procedures with a name f and a definition *pgm*. For instance, $\mathsf{int}\, x$ defines an integer

variable x while $\mathsf{sc\_signal}\langle\mathsf{bool}\rangle\ x$ defines a boolean signal x. We assume x to denote the name of a variable or signal and to be possibly prefixed as $m :: x$ by the name of the class it belongs to. We assume the relation $\leq$ to denote SystemC sub-typing, e.g., $\mathsf{bool} \leq \mathsf{num}$ or $\mathsf{int} \leq \mathsf{num}$.

$$
\begin{array}{lll}
sys & ::= [\mathsf{template}\,\langle\mathsf{class}\,m_1\rangle\,\#\mathsf{TYPE}(m_1, m_2)]\,\mathsf{class}\,m\,\{decl\} & \text{(class)} \\
& \mid \mathsf{SC\_MODULE}(m)\ \{decl; \mathsf{SC\_CTOR}(m)\ \{new\}\} & \text{(module)} \\
& \mid sys; sys & \text{(sequence)} \\
decl & ::= m\langle m^*\rangle\ x & \text{(field)} \\
& \mid \mathsf{void}\ f()\ \{pgm\} & \text{(thread)} \\
& \mid decl; decl & \text{(sequence)} \\
new & ::= \mathsf{SC\_THREAD}(f)\,\mathsf{sensitive} \ll x^* \mid new; pgm & \text{(constructor)}
\end{array}
$$

Figure 9.4: Abstract syntax for SystemC

The formal grammar of SystemC programs, listed in Figure 9.5, is represented in static single-assignment intermediate form akin to that of the Tree-SSA package of the GCC project [CFR+91].

$$
\begin{array}{lllll}
\text{(program)} & pgm & ::= & L{:}blk \mid pgm; pgm & \\
\text{(block)} & blk & ::= & stm; blk \mid rtn & \\
\text{(instruction)} & stm & ::= & x = f(x^*) & \text{(invoke)} \\
 & & & \mid \mathsf{wait}\,x & \text{(lock)} \\
 & & & \mid \mathsf{notify}\,x & \text{(unlock)} \\
 & & & \mid \mathsf{if}\,x\,\mathsf{then}\,(blk \mid blk\,\mathsf{else}\,blk) & \text{(test)} \\
\text{(return)} & rtn & ::= & \mathsf{goto}\,L & \text{(goto)} \\
 & & & \mid \mathsf{return} & \text{(return)} \\
 & & & \mid \mathsf{throw}\,x; & \text{(throw)} \\
 & & & \mathsf{catch}\,x\,\mathsf{from}\,L & \text{(catch)} \\
 & & & \mathsf{to}\,L\,\mathsf{using}\,L &
\end{array}
$$

Figure 9.5: Abstract syntax for SystemC programs in SSA form

SSA provides a language-independent, locally optimized intermediate representation (Tree-SSA currently accepts C, C++, Fortran 95, and Java inputs) in which language-specific syntactic sugar is absent. SSA transforms a given programming unit (a function, a method or a thread) into a structure in which all variables are read and written once and all native operations are represented by 3-address instructions $x = f(y, z)$.

A program *pgm* consists of a sequence of labeled blocks L:*blk*. Each block consists of a label L and of a sequence of statements *stm* terminated by a return statement *rtn*. In the remainder, a block always starts with a label and finishes with a return statement: $stm_1; L{:}stm_2$ is rewritten as $stm_1;\mathsf{goto}\,L; L{:}stm_2$. A wait is always placed at the beginning of a block: $stm_1;\mathsf{wait}\,v; stm_2$ is rewritten as $stm_1;\mathsf{goto}\,L; L{:}\mathsf{wait}\,v; stm_2$.

Block instructions consist of native method invocations $x = f(x^*)$, lock monitoring and branches $\mathsf{if}\,x\,\mathsf{then}\,L$. Blocks are returned from by either a $\mathsf{goto}\,L$, a return or an exception $\mathsf{throw}\,x$. The declaration $\mathsf{catch}\,x\,\mathsf{from}\,L_1\,\mathsf{to}\,L_2\,\mathsf{using}\,L_3$ that matches an exception x raised at block L_1 activates the exception handler L_3 and continues at block L_2.

Example 1 *To outline the construction of the intermediate representation of a SystemC program, let us reconsider the example of Section 9.1 and detail the method that counts the number of bits set to* 1 *in a bit-array* `epc.data`. *This time we give an entire SystemC function. It features a lock mechanism. This lock mechanism notifies the method when it is allowed to execute (for example when a new relevant data value is available). Once the execution finishes, the method sends out a notification, signaling to the rest of the program that the output is available and can be read securely.*

```
void epc::ones () {
  sc_int<16> idata, ocnt;
  while true {
    wait (epc.lock);
    idata  = epc.data;
    ocnt = 0;
    while (idata != 0) {
      ocnt = ocnt + (idata & 1);
      idata = idata >> 1;
    }
    epc.cnt = ocnt;
    notify (epc.lock);
  }
}
```

Figure 9.6: Ones-counter method in SystemC

The corresponding SSA code of this method (Figure 9.7) consists of four blocks. The block labeled `L1` *waits for the lock* `epc.lock` *before initializing the local state variable* `idata` *to the value of the input signal* `epc.data` *and* `ocnt` *to* 0. *Then it passes control to the block* `L2`. *Label* `L2` *evaluates the termination condition of*

```
L1: wait(epc.lock);
    idata_1 = epc.data;
    ocnt_1 = 0;
    goto L2;

L3: tmp = (idata_3 & 1);
    ocnt_2 = ocnt_3 + tmp;
    idata_2 = idata_3 >> 1;

L2: idata_3 = φ(idata_1, idata_2);
    ocnt_3 = φ(ocnt_1, ocnt_2);
    cond = (idata_3 != 0);
    if cond then goto L3

L4: epc.cnt = ocnt_3;
    notify(epc.lock);
    goto L1;
```

Figure 9.7: Ones-counter method in SSA

the loop and passes on control accordingly. As there are several possible sources in the control flow for the variables `idata` *and* `ocnt`*, it determines the most recent value with the help of* ϕ *functions (lines 10 and 11). If the termination condition is not yet satisfied, control goes to block* `L3`*, which corresponds to the actual loop contents that shifts* `idata` *right and adds its right-most bit to* `ocnt`*. If the termination condition is satisfied - i.e. all available bits have been counted - control goes to block* `L4` *where the result is copied to the output signal* `epc.cnt` *and* `epc.lock` *is unlocked before passing control back to* `L1` *in order to wait for the next available input vector.*

9.3 Inference

The definition of uniform methodologies for the formal design of GALS architectures has been proposed in [LTL03] and [TGS+03]. After having defined a formal syntax for the SystemC core statements, we are able to infer behavioral types from SystemC descriptions. This is the important step as once we are able to infer formal behavioral types from informal descriptions, we can then automate the process. Once we are able to automatically obtain them, they can be generated for whole libraries of existing IP components and in the following be used to ensure safer component compositions.

The type inference function $\mathcal{I}[\![pgm]\!]$, Figure 9.8, is defined by induction on the formal syntax of *pgm*. To define it, we assume that the finite set $Ł_f$ of program

labels L defined in a given method f respects the order of appearance in the text: $L_1 < L_2$ means that L_1 occurs before L_2.

To each block of label $L \in \mathrm{L}_f$, the function $\mathcal{I}[\![pgm]\!]$ associates an *input clock* x_L, an *immediate clock* x_L^{imm} and an *output clock* x_L^{exit}. The clock x_L is true iff L has been activated in the previous transition (by emitting the event x'_L). The clock x_L^{imm} is set to true to activate the block L immediately. The clock x_L^{exit} is set to true when the execution of the block labeled L terminates. The default activation condition of this block is the clock $x_L \vee x_L^{imm}$ (equation (1) of Figure 9.8). The block L is executed iff the proposition $x_L \vee x_L^{imm}$ holds, meaning that the program counter is at L.

For a return instruction or for a block, the type inference function returns a type P. For a block instruction *stm*, the type inference function $\mathcal{I}[\![stm]\!]_L^{e_1} = \langle P \rangle^{e_2}$ takes three arguments: an instruction *stm*, the label L of the block it belongs to, and an input clock e_1. It returns the type P of the instruction and its output clock e_2. The output clock of *stm* corresponds to the input clock of the instruction that immediately follows it in the execution sequence of the block.

Figure 9.8 defines the behavioral type inference system. Rules $(1-2)$ are concerned with the iterative decomposition of a program *pgm* into blocks *blk* and with the decomposition of a block into *stm* and *rtn* instructions. In rule (2), the input clock e of the block *stm*; *blk* is passed to *stm*. The output clock e_1 of *stm* becomes the input clock of *blk*.

$$
\begin{array}{ll}
(1) & \mathcal{I}[\![L{:}blk; pgm]\!] = \mathcal{I}[\![blk]\!]_L^{x_L \vee x_L^{imm}} \,|\, \mathcal{I}[\![pgm]\!] \\
(2) & \mathcal{I}[\![stm; blk]\!]_L^e = \text{let } \langle P \rangle^{e_1} = \mathcal{I}[\![stm]\!]_L^e \text{ in } P \,|\, \mathcal{I}[\![blk]\!]_L^{e_1} \\
(3) & \mathcal{I}[\![\mathsf{if}\ x\ \mathsf{then}\ L_1]\!]_L^e = \langle \mathcal{G}_L(L_1, e \wedge x) \rangle^{e \wedge \neg x} \\
(4) & \mathcal{I}[\![x = f(y^*)]\!]_L^e = \langle \mathcal{E}(f)(xy^*e) \rangle^e \\
(5) & \mathcal{I}[\![\mathsf{notify}\ x]\!]_L^e = \langle e \Rightarrow (x' = \neg x) \rangle^e \\
(6) & \mathcal{I}[\![\mathsf{wait}\ x]\!]_L^e = \langle e \wedge (x \neq x') \Rightarrow \hat{y} \,|\, e \setminus \hat{y} \Rightarrow x'_L \rangle^{\hat{y}} \\
(7) & \mathcal{I}[\![\mathsf{goto}\ L_1]\!]_L^e = (e \Rightarrow x_L^{exit} \,|\, \mathcal{G}_L(L_1, e)) \\
(8) & \mathcal{I}[\![\mathsf{return}]\!]_L^e = (e \Rightarrow (x_L^{exit} \,|\, x_f^{exit})) \\
(9) & \mathcal{I}[\![\mathsf{throw}\ x]\!]_L^e = (e \Rightarrow (x_L^{exit} \,|\, \hat{x}))
\end{array}
$$

where
$$\mathcal{G}_L(L_1, e) = \text{if } \mathcal{S}_L(L_1) \text{ then } e \Rightarrow x_{L_1}^{imm} \text{ else } \langle e \Rightarrow x'_{L_1} \rangle$$
$$\mathcal{E}(f)(xyze) = e \Rightarrow (\hat{y} \wedge \hat{z} \Rightarrow (\hat{x} \,|\, y \to x \,|\, z \to x)),\ \forall f x y z e$$
$$\mathcal{I}[\![\mathsf{catch}\ x\ \mathsf{from}\ L\ \mathsf{to}\ L_1\ \mathsf{using}\ L_2]\!]_L^e = \mathcal{G}_L(L_2, \hat{x} \wedge x_L^{exit}) \,|\, G_{L_2}(L_1, x_{L_2}^{exit})$$

Figure 9.8: Type inference rules

The input and output clocks of an instruction may differ. This is the case, rule (3), for an $\mathsf{if}\, x\, \mathsf{then}\, L_1$ instruction in a block L. Let e be the input clock of the instruction. When x is false, then control is passed to the rest of the block, at the output clock $e \wedge \neg x$. Otherwise, the control is passed to the block L_1, at the clock $e \wedge x$.

There are two ways of passing the control from L to L_1 at a given clock e. They are defined by the function $\mathcal{G}_L(L_1, e)$: either immediately, by activating the immediate clock $x^{imm}_{L_1}$, i.e., $e \Rightarrow x^{imm}_{L_1}$; or by a delayed transition to L_1 at e, i.e., $e \Rightarrow x'_{L_1}$. This choice is decided by the auxiliary function $\mathcal{S}_L(L_1)$. It checks whether the block L_1 can be executed immediately after the block L. By definition, $\mathcal{S}_L(L_1)$ holds iff $L_1 > L$ (L_1 is after L in the control flow) and $\mathcal{D}(L_1) \cap \mathcal{D}(L) = \emptyset$ (the set of variables defined in L and L_1 are disjoint).

Example 2 *In Example 1,* $\mathcal{D}(\mathsf{L1}) = \mathcal{D}(\mathsf{L2}) = \mathcal{D}(\mathsf{L3}) = \{\mathsf{ocnt}, \mathsf{idata}\}$ *and* $\mathcal{D}(\mathsf{L4}) = \{\mathsf{cnt}, \mathsf{lock}\}$. *Hence, going from* `L1` *or* `L2` *to* `L3` *and from* `L3` *to* `L2` *requires delayed transitions because they all define* `ocnt` *and* `idata`. *Conversely, going from* `L3` *to* `L4` *can be done immediately since* `L4` *does define neither* `ocnt` *nor* `idata`.

Rule (4) is concerned with the typing of native and external method invocations $x = f(y^*)$. The generic type of f is taken from an environment $\mathcal{E}(f)$. It is given the name of the result x, of the actual parameters y^* and of the input clock e to obtain the type of $x = f(y^*)$. On the right, the generic type of 3-address instructions $x = f(y, z)$ at clock e is given by $\mathcal{E}(f)(xyze)$.

The wait-notify protocol (rules $(5-6)$) is modeled using a boolean flip-flop variable x. The notify method, rule (5), defines the next value of the lock x by the negation of its current value at the input clock e.

The wait method, rule (6), activates its output clock $\hat{y}$ iff the value of the lock x has changed at the input clock e: $e \wedge (x \neq x') \Rightarrow \hat{y}$. Otherwise, at the clock $e \setminus \hat{y}$, the control is passed to L by a delayed transition $e \setminus \hat{y} \Rightarrow x'_L$.

Example 3 *Consider the wait-notify protocol at the blocks labeled* `L1` *and* `L4` *in the* `ones` *counter (Figure 9.9). The type of the wait instruction defines the output clock* $\hat{y}$ *if* `L1` *receives control at the clock* x_{L1}, *and if the value of lock has changed (proposition* $\mathsf{lock} \neq \mathsf{lock}'$*). If so, at the clock* $\hat{y}$, `ocnt` *and* `idata` *are initialized and the control is passed to the block* `L3` *by* $\mathcal{G}_{\mathsf{L1}}(\mathsf{L3}, \hat{\mathsf{y}})$. *Otherwise, at the clock* $x_{\mathsf{L1}} \setminus \hat{y}$, *a delayed transition to* `L1` *is scheduled:* $x_{\mathsf{L1}} \setminus \hat{y} \Rightarrow x'_{\mathsf{L1}}$.

All return instructions, rules $(7-9)$, define the output clock x^{exit}_L of the current block L by their input clock e. This is the right place to do that: e defines the very condition upon which the block actually reaches its return statement.

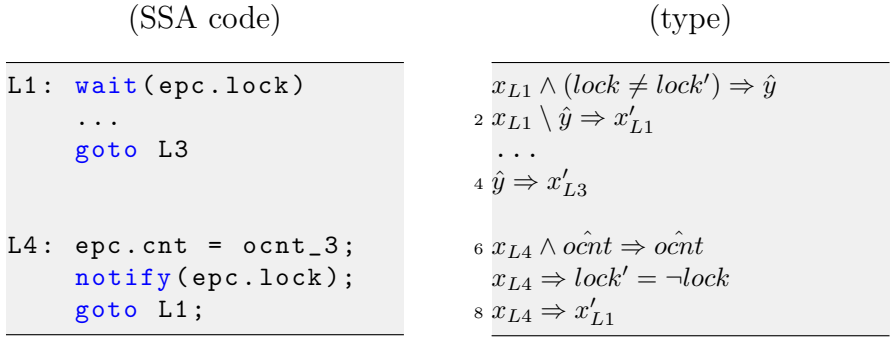


Figure 9.9: Type of the wait-notify protocol

A goto L_1 instruction, rule (7), passes control to block L_1 unconditionally at the input clock e by $\mathcal{G}_L(L_1, e)$. A return instruction, rule (8), sets the exit clock x_f to true at clock e to inform the caller that f is terminated.

A throw x statement in block L, rule (9), triggers the exception signal x at the input clock e by $e \Rightarrow \hat{x}$. The matching catch statement, of the form catch x from L to L_1 using L_2 passes the control to the handler L_2 and then to the block L_1 upon termination of the handler.

This requires, first, to activate L_2 from L when x is present, i.e., $\mathcal{G}_L(L_2, \hat{x} \wedge x_L^{exit})$, and then to pass control to L_1 upon termination of the handler.

9.3.1 Completion of the State Logic

The encoding of Figure 9.8 requires all entry clocks x_L, x_L^{imm} and x_f to be present when the f is being executed. Each signal x_L holds the value 1 iff the block L is active during a transition currently being executed. Otherwise, x_L is set to 0. This default setting of the entry clocks requires a completion of the next-state logic by considering, for all $L \in Ł_f$, the proposition $e_L \Rightarrow x'_L$ implied by the inferred type $P = \mathcal{I}[\![pgm]\!]$ and defines the default rule by $x_f \setminus e_L \Rightarrow \neg x'_L$. Completion is identical for the immediate and exit clocks x_L^{imm} and x_L^{exit} of the block L.

$$x_f \setminus e_L \Rightarrow \neg x'_L \text{ where } e_L \stackrel{\text{def}}{=} \bigvee (e \,|\, P \models e \Rightarrow x'_L)$$

9.3.2 Modular Extension to External Method Calls

The type inference scheme defined for wait, notify, and operations, rules $(4-6)$ can be extended to handle externally defined method calls in a modular and compositional way, depicted in Figure 9.10.

$$(a)\ \mathcal{I}[\![m\ f(x_{1..m})\ \mathsf{raises}\ y\{\ pgm\ \}]\!]{=}\lambda x_{1..m} x_f\, x_f^{exit} y_f.\ \Big(\mathcal{I}[\![pgm]\!]\,|\,x_f \Rightarrow x_{\min \mathrm{L}_f}\Big)\,/\mathrm{L}_f$$

$$(b)\qquad \mathcal{I}[\![L : x_0 = f(x_{1..m})]\!]^e_L{=}e \Rightarrow (\mathcal{E}(f)(x_{1..m} e x y)\,|\,e \setminus (\hat{y} \vee \hat{x}) \Rightarrow x'_L)^{e \wedge \hat{x}}$$

$$(c)\qquad \mathcal{I}[\![\mathsf{return}\ x]\!]^e_L{=}(e \Rightarrow (x_L^{exit}\,|\,x_f^{exit} := x))$$

Figure 9.10: Modular extension of the inference function to separate methods

Consider a method f with formal parameters $x_{1..m}$ (whose data-types are not displayed) and a result of type m, rule (a). Let y be an exception raised by the definition pgm of f and escaping from it. The type of f consists of a lambda abstraction whose arguments are the inputs $x_{1..m}$, the entry clock x_f, the exit clock x_f^{exit}, the return value y_f and the exception y. It is used to parameterize the proposition P, which corresponds to pgm, with respect to these arguments.

The lambda abstraction is instantiated in place of a method invocation $L : x_0 = f(x_{1..m})$, rule (b), which needs to be placed at the beginning of a block (assuming that this block can take several transitions before termination). To model the method call, one just needs to activate the entry clock x_f of the method at the input clock e.

The output signal x is used to carry the value of the result. Its clock determines when the method has reached the corresponding return statement (rule (c)). When the method terminates, the exit clock of the method call is defined by $e \wedge \hat{x}$. Otherwise, if the exception y is raised, a corresponding catch statement handles it. If f has not finished at the end of the transition (at the clock $e \setminus (\hat{y} \vee \hat{x})$), a delayed transition to L is performed $e \setminus (\hat{y} \vee \hat{x}) \Rightarrow x'_L$ in order to resume its execution at the next transition.

9.3.3 Static Interface of SystemC Modules

The construction of a static abstraction from the behavioral type P of a program is automatic, thanks to its clock inference system. Example 4 illustrates how such a static type is inferred and what the notations mean.

Example 4 *Figure 9.11 shows how such a static interface looks like. For instance, the instruction* $x_{\mathsf{L3}} \Rightarrow tmp := idata\&1$ *is abstracted by the type* $x_{\mathsf{L3}} \Rightarrow \hat{tmp} = \hat{idata}$*. It means: "when the block* L3 *is executed,* tmp *is present iff* idata *is present". The scheduling constraints give an indication on the dependencies of the variables. In line 6* cond *is assigned a new value that is used in lines 7 and 8. The scheduling abstraction will reflect this, by noting in line 6* $x_{L2} \Rightarrow idata \rightarrow cond$*, meaning that the value of cond depends on the value of idata, and in line 7 and*

8 of the scheduling it is expressed that the values of x'_{L3} and x'_{L4} depend on the value of cond, and therefore these instructions have to be scheduled later.

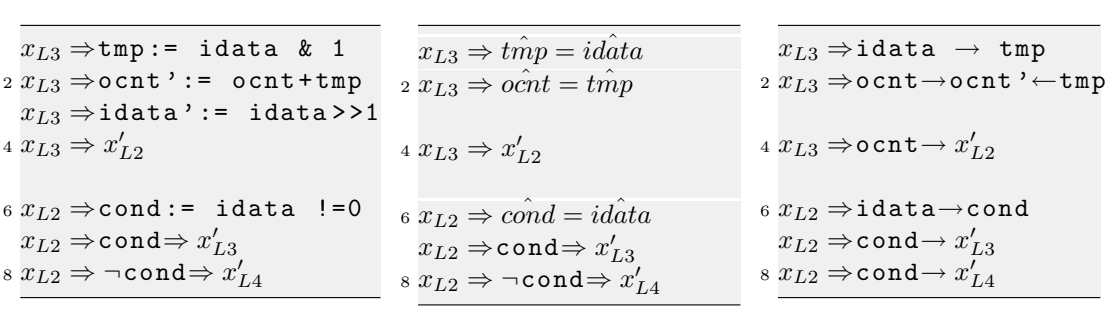


Figure 9.11: Abstraction of the behavioral type of the while loop by a static interface

The type associated with the whole loop example uses the clocks denoted by the booleans x_{L1}, x_{L2}, x_{L3}, and x_{L4}. Each clock denotes a branch in the control-flow graph of the program. The other clocks, e.g. $\hat{\mathsf{ocnt}}$, denote the presence of data. They are partially related to the "label" clocks such as x_{L2}. This means that each label clock is a superset of all the instants that occur in the variables of this block.

9.4 A Behavioral Module System

We define a module system starting from the behavioral type inference function of Section 9.3.

$$\begin{array}{lrcl} \text{(type)} & \mathcal{T} & ::= & \langle \mathcal{E}, P, \mathcal{C} \rangle \mid \mathcal{T} \rightarrow \mathcal{T} \\ \text{(context)} & \mathcal{E} & ::= & [] \mid \mathcal{E}[x : m] \mid \mathcal{E}[f : P] \mid \mathcal{E}[m : \mathcal{T}] \\ \text{(obligation)} & \mathcal{C} & ::= & 1 \mid P \Rightarrow Q \mid \mathcal{C} \wedge \mathcal{C} \end{array}$$

Figure 9.12: Behavioral types for modules

The type $\mathcal{T}$ of a module m, Figure 9.12, consists of an environment $\mathcal{E}$ that associates its methods f and fields x with types, of a type P that denotes the behavior of its constructor, and of a proof obligation $\mathcal{C}$.

The type $\mathcal{T}_1 \rightarrow \mathcal{T}_2$ denotes a template class that produces a module of type $\mathcal{T}_2$ given a parameter of type $\mathcal{T}_1$. A proof obligation is a conjunction of propositions of the form $P \Rightarrow Q$. A proof obligation $P \Rightarrow Q$ is incurred by the instantiation of a template class, whose formal parameter has type P, by an actual class parameter, of type Q.

The type inference function for modules, $\mathcal{I}[\![sys]\!]_{\mathcal{E}}$ assumes a type environment $\mathcal{E}$ that associates names with types. We write $\mathcal{E}(x)$ for the type of the location x and $\mathcal{E}(m.x) = \mathcal{F}(m)(x)$ for the path m to x iff $\mathcal{E}(m) = \langle \mathcal{F}, P, \mathcal{C} \rangle$.

9.4.1 Type Inference for Declarations

Rule (a) sequentially processes the declarations *decl* in a module. Class field declarations contribute to building the type $\mathcal{T}$ of a module: rule (b) associates the location x with the type name m in the class-field $[x : m]$, rule (c) associates the procedure f with the class-field $[f : P]$. The type τ denotes a program that does nothing. It is neutral by composition.

In rule (d), the initialization of a thread $\mathsf{SC\_THREAD}(f)\,\mathsf{sensitive} \ll x$ in the constructor is associated with the behavior $\mathcal{E}(f)$ of the method f it forks and with the type $\hat{x}_f < \hat{x} >$, meaning that x triggers f.

$$
\begin{array}{ll}
(a) & \mathcal{I}[\![decl_1; decl_2\}]\!]_{\mathcal{E}} = \text{let } \langle \mathcal{E}_1, P_1, \mathcal{C}_1 \rangle = \mathcal{I}[\![decl_1]\!]_{\mathcal{E}} \\
 & \qquad\qquad \text{in } \langle \mathcal{E}_1, P_1, \mathcal{C}_1 \rangle \uplus \mathcal{I}[\![decl_2]\!]_{\mathcal{E}\mathcal{E}_1} \\
(b) & \mathcal{I}[\![m\, x]\!]_{\mathcal{E}} = \langle [x : m], \tau, 1 \rangle \\
(c) & \mathcal{I}[\![\mathsf{void}\ f()\ \{pgm\}]\!]_{\mathcal{E}} = \langle [f : \mathcal{I}[\![pgm]\!]_{\mathcal{E}}], \tau, 1 \rangle \\
(d) & \mathcal{I}[\![\mathsf{SC\_THREAD}(f)\,\mathsf{sensitive} \ll x]\!]_{\mathcal{E}} = \langle [], \mathcal{E}(f) \parallel (\hat{x}_f \langle \hat{x} \rangle), 1 \rangle
\end{array}
$$

Figure 9.13: Type inference for declarations

9.4.2 Type Inference for Modules

Rule (e) processes a sequence of module declarations $sys_1; sys_2$. We write $\langle \mathcal{E}_1, P_1, \mathcal{C}_1 \rangle \uplus \langle \mathcal{E}_2, P_2, \mathcal{C}_2 \rangle = \langle \mathcal{E}_1\mathcal{E}_2, P_1 \parallel P_2, \mathcal{C}_1 \wedge \mathcal{C}_2 \rangle$ to merge the types of sys_1 and sys_2. While processing is sequential, the composition of the behavioral type $P_1 \parallel P_2$ is synchronous.

Rule (f) first obtains the type $\mathcal{T}_1 = \langle \mathcal{E}_1, P_1, \mathcal{C}_1 \rangle$ of its class fields. Then, in the environment $\mathcal{E}$ extended with that of the class fields $\mathcal{E}_1$, the body $new; pgm$ of the constructor is processed to obtain its type $\mathcal{T}_2$. The type of the module becomes $m \cdot (\mathcal{T}_1 \uplus \mathcal{T}_2)$. The notation $m \cdot \langle \mathcal{E}, P, \mathcal{C} \rangle = \langle [m : \mathcal{E}], P, \mathcal{C} \rangle$ (resp. $m \cdot (\mathcal{T}_1 \to \mathcal{T}_2) = \mathcal{T}_1 \to (m \cdot \mathcal{T}_2)$) defines the type of the class m from the type $\langle \mathcal{E}, P, \mathcal{C} \rangle$ of its class fields.

Rule (g) determines the type of a template class m_2 whose formal parameter is a class m_1 that implements the virtual class m. The virtual class m provides the type, and hence the expected behavior, of the formal parameter name m_1. It is obtained from the environment $\mathcal{E}$ by $m \cdot \mathcal{T} = \mathcal{E}(m)$. The body of the template (i.e. the field declarations *decl* of the class m_2) is type-checked with the environment

$\mathcal{E}$ extended with the association of m_1 to the type of the class fields $\mathcal{E}_1$ declared in m. This yields the type $m_2 \cdot \mathcal{T}_2$ of the class. The type of the template is defined by associating m_2 with the type $(m_1 \cdot \mathcal{T}_1) \to \mathcal{T}_2$ (and hence m_1 with the type $\mathcal{T}_1$).

Rule (h) performs the instantiation $m_2\langle m\rangle\, x$ of a template class m_2 with an actual parameter m to define the class name x. The type $(m_1 \cdot \mathcal{T}_1) \to \mathcal{T}_2$ of the template class m_2 and the type $m \cdot \mathcal{T}$ of the actual parameter m are obtained from the supplied environment $\mathcal{E}$. Type matching between $\mathcal{T}$ and $\mathcal{T}_1$ requires the resolution of a sub-typing between $\mathcal{T}_1[m_2.m/m_1]$ and $\mathcal{T}_2[m_2.m/m]$. The term $\mathcal{T}_1[m_2.m/m_1]$ stands for the substitution of the name m_1 by the concatenation $m_2.m$ in $\mathcal{T}_1$. The resolution of the type matching constraints reduces to the synthesis of the proof obligation $\mathcal{C}$ by the algorithm $\mathcal{R}$. If $\mathcal{C}$ is satisfied, then the type of the location x is $\mathcal{T}_2[m_2.m/m_1]$.

$$
\begin{array}{lll}
(e) & \mathcal{I}[\![sys_1; sys_2]\!]_{\mathcal{E}} = & \mathsf{let}\ \langle \mathcal{E}_1, P_1, \mathcal{C}_1\rangle = \mathcal{I}[\![sys_1]\!]_{\mathcal{E}} \\
 & & \mathsf{in}\ \langle \mathcal{E}_1, P_1, \mathcal{C}_1\rangle \uplus \mathcal{I}[\![sys_2]\!]_{\mathcal{E}\mathcal{E}_1} \\
(f) & \mathcal{I}\left[\!\!\left[\begin{array}{l}\mathsf{SC\_MODULE}(m)\,\{\ decl;\\ \ \mathsf{SC\_CTOR}(m)\,\{new; pgm\}\}\end{array}\right]\!\!\right]_{\mathcal{E}} = & \mathsf{let}\ \langle \mathcal{E}_1, P_1, \mathcal{C}_1\rangle = \mathcal{I}[\![decl]\!]_{\mathcal{E}} \\
 & & \quad \&\ \ \mathcal{T}_2 = \mathcal{I}[\![new; pgm]\!]_{\mathcal{E}\mathcal{E}_1} \\
 & & \mathsf{in}\ m \cdot (\langle \mathcal{E}_1, P_1, \mathcal{C}_1\rangle \uplus \mathcal{T}_2) \\
(g) & \mathcal{I}\left[\!\!\left[\begin{array}{l}\mathsf{template}\,\langle \mathsf{class}\, m_1\rangle\\ \ \#\mathsf{TYPE}(m_1, m)\\ \ \mathsf{class}\, m_2\,\{decl\}\end{array}\right]\!\!\right]_{\mathcal{E}} = & \mathsf{let}\ m \cdot \mathcal{T} = \mathcal{E}(m) \\
 & & \quad \&\ \ \langle \mathcal{E}_1, \_, \_\rangle = m_1 \cdot \mathcal{T} \\
 & & \quad \&\ \ \mathcal{T}_2 = \mathcal{I}[\![decl]\!]_{\mathcal{E}\mathcal{E}_1} \\
 & & \mathsf{in}\ \langle [m_2 : (m_1 \cdot \mathcal{T}_1) \to \mathcal{T}_2], \tau, 1\rangle \\
(h) & \mathcal{I}[\![m_2\langle m\rangle\, x]\!]_{\mathcal{E}} = & \mathsf{let}\ (m_1 \cdot \mathcal{T}_1) \to \mathcal{T}_2 = \mathcal{E}(m_2) \\
 & & \quad \&\ \ m \cdot \mathcal{T} = \mathcal{E}(m) \\
 & & \quad \&\ \ \mathcal{C} = \mathcal{R}(\mathcal{T}_1[m_2.m/m_1], \mathcal{T}[m_2.m/m]) \\
 & & \mathsf{in}\ x \cdot (\mathcal{T}_2[m_2.m/m_1]) \uplus \langle [], \tau, \mathcal{C}\rangle)
\end{array}
$$

Figure 9.14: Type inference for modules

9.4.3 Proof Obligation Synthesis

The resolution $\mathcal{R}(\mathcal{T}_1, \mathcal{T}_2)$ of sub-typing constraints is defined by induction on the structure of the pair $(\mathcal{T}_1, \mathcal{T}_2)$. It reduces to the proof of a conjunction of

propositions of the form $P_1 \Rightarrow P_2$ (of denotation $[\![P_1]\!] \subseteq [\![P_2]\!]$, see Section 9.5).

$$\begin{aligned}
\mathcal{R}([],[]) &\Leftrightarrow 1 \\
\mathcal{R}(\mathcal{E}_1 \to \mathcal{T}_1, \mathcal{E}_2 \to \mathcal{T}_2) &\Leftrightarrow \mathcal{R}(\mathcal{E}_2, \mathcal{E}_1) \wedge \mathcal{R}(\mathcal{T}_1, \mathcal{T}_2) \\
\mathcal{R}(\mathcal{E}_1[x : t_1], \mathcal{E}_2[x : t_2]) &\Leftrightarrow \mathcal{R}(\mathcal{E}_1, \mathcal{E}_2) \wedge (t_2 \leq t_1) \\
\mathcal{R}(\mathcal{E}_1[f : P_1], \mathcal{E}_2[f : P_2]) &\Leftrightarrow \mathcal{R}(\mathcal{E}_1, \mathcal{E}_2) \wedge (P_2 \Rightarrow P_1) \\
\mathcal{R}(\mathcal{E}_1[m : \mathcal{T}_1], \mathcal{E}_2[m : \mathcal{T}_2]) &\Leftrightarrow \mathcal{R}(\mathcal{E}_1, \mathcal{E}_2) \wedge \mathcal{R}(\mathcal{T}_1, \mathcal{T}_2) \\
\mathcal{R}(\langle \mathcal{E}_1, P_1, \mathcal{C}_1 \rangle, \langle \mathcal{E}_2, P_2, \mathcal{C}_2 \rangle) &\Leftrightarrow \mathcal{R}(\mathcal{E}_1, \mathcal{E}_2) \wedge \mathcal{R}(P_1, P_2) \wedge \mathcal{R}(\mathcal{C}_1, \mathcal{C}_2)
\end{aligned}$$

If P_2 is *static* (i.e. $P_2 \Leftrightarrow \hat{P}_2$) then the problem reduces to checking for satisfaction of the boolean proposition $\hat{P}_1 \Rightarrow \hat{P}_2$). If P_2 is *dynamic* then the problem reduces to verifying that $P_1 \Rightarrow P_2$ is an invariant of P, the type of the program, by using model-checking techniques. Both problems can be expressed and decided using the Polychrony workbench [IRI], as demonstrated in Appendix 9.5, where SystemC programs and their behavioral types are embedded in Signal.

9.5 Behavioral Types in Polychrony

The Polychrony workbench [IRI] supports the synchronous multi-clocked data-flow programming language Signal. As we have defined now a way on how to define behavioral types starting from SystemC descriptions, in order to be able to express these behavioral types in Polychrony we just have to define a way on how to translate the iSTS of the behavioral type system we just defined into Polychrony.

$$\begin{aligned}
\mathcal{C}[\![L{:}blk; pgm]\!] &= \mathcal{C}[\![L{:}blk]\!]_L^{x_L} \,|\, \mathcal{C}[\![pgm]\!] \\
\mathcal{C}[\![stm; blk]\!]_L^e &= \text{let } \langle \mathtt{P} \rangle^{e_1} = \mathcal{C}[\![stm]\!]_L^e \text{ in } \mathtt{P} \,|\, \mathcal{C}[\![blk]\!]_L^{e_1} \\
\mathcal{C}[\![rtn]\!]_L^e &= \mathcal{T}[\![\mathcal{I}[\![rtn]\!]_L^e]\!] \\
\mathcal{C}[\![stm]\!]_L^e &= \text{if } (stm \neq \text{"}x = f(x_{1\ldots n})\text{"})) \\
&\quad \text{then let } \langle P \rangle^{e_1} = \mathcal{I}[\![stm]\!]_L^e \text{ in } \langle \mathcal{T}[\![P]\!] \rangle^{e_1} \\
&\quad \text{else let } \mathtt{e} = \mathcal{T}[\![e]\!] \text{ and } \mathtt{P} = \mathcal{T}[\![\mathcal{E}(f)(x x_{1\ldots n} e)]\!] \\
&\quad \text{in } \left\langle \begin{array}{l} \mathtt{spec\ P} \\ \mathtt{pragmas\ CPP_CODE\ "if\ e\ \{\ x = f(x_{1\ldots n})\ \}"} \\ \mathtt{end\ pragmas} \end{array} \right\rangle^e
\end{aligned}$$

Figure 9.15: Embedding the intermediate representation in Signal

We recall from Section 8.4 the translation from the iSTS notation into Signal, Figure 8.15, and the encoding of Signal by behavioral types, Figure 8.14. The behavioral type system therefore supports a direct translation into the data-flow notation Signal of the Polychrony workbench [IRI]. This translation allows for

the complete embedding of SystemC modules into Polychrony, Figure 9.15, to perform global design transformations, such as hierarchization or distribution and to perform a correct-by-construction design exploration towards the mapping of system functionalities onto a target execution architecture.

The translation consists of representing guarded commands $e \Rightarrow (x' = v)$ and $e \Rightarrow \hat{x}$ by partial equations `x ::= v when e` and `^x ::= when e`, of identical meaning, and of embedding native method invocations $x = f(x_{1...n})$ in wrappers (the `CPP_CODE` part) visible from Signal via a behavioral type (the `spec` part). Since the previous value of a signal x is noted $x\$1$ in Signal, we assume the default equation $x := x'\$1$ to define the current value x of a variable (of type bool or int in SystemC).

Example 5 *As an example, embedding the* ones *counter into Signal consists of emulating control by partial equations and of wrapping computations using typed* pragma *statements (Figure 9.16).*

(SSA code)

```
L1: wait(epc.lock);
    idata_1 = epc.data;
    ocnt_1 = 0;
    goto L2;


L3: tmp = (idata_3 & 1);
    ocnt_2 = ocnt_3 + tmp;
    idata_2 = idata_3 >> 1;
    goto L2;

L2: idata_3= φ(idata_1,idata_2);
    ocnt_3 = φ(ocnt_1, ocnt_2);
    cond = (idata_3 != 0);
    if cond then goto L3

L4: epc.cnt = ocnt_3;
    notify(epc.lock);
    goto L1;
```

(SIGNAL type)

```
1  x1 := (lock≠lock') when xL1
   x'L1 ::= when not x1
3  idata ::= data when x1
   ocnt ::= 0 when x1
5  x'L2 ::= when x1

7  tmp := f1(idata) when xL3
   ocnt ::=ocnt$1 + tmp when xL3
9  idata ::= f2(idata$1) when xL3
   x'L2 ::= when xL3
11
   cond := idata != 0 when xL2
13 x'L3 := when cond
   xL4 := when not cond
15


17 cnt ::= ocnt when xL4
   lock ::= not lock when xL4
19 x'L1 ::= when xL3

21 xL1 := x'L1$1 init true
   xL2 := x'L2$1 init false
23 xL3 := x'L3$1 init false
   x^f1 = x^f2 = x^L1 = x^L2 = x^L3 = xL4
```

Figure 9.16: Signal type of the ones counter

This embedding allows to operate global architectural transformations on the initial program, such as hierarchization or distributed protocol synthesis, using the Polychrony platform [IRI] or perform both static (SAT-checking) or dynamic (model-checking) verification of its design properties, whose spectrum is outlined next.

The SIGNAL code on the right hand side of the figure can be obtained from the SSA mostly by line to line transformations. Lines 21 to 23 of the SIGNAL code establish a connection between the next value of the blocks and the current value. Only x_{L1} is initialized by `true`*, the others get activated once the execution advances. The completion of the state-logic is implemented by the aggregation of partial state equations. Notice that* `L1`*,* `L2`*, and* `L3` *are always activated by a delayed transition, whereas* `L4` *is always immediate. These partial equations can be integrated into complete equations easily. This is done implicitly by the SIGNAL compiler, but is sometimes useful in order to improve readability. Below are two examples of how partial equations can be merged into one:*

```
x'_L1 := true when not x_1 default x_L4 default false
x'_L2 := true when x_1 default x_L3 default false
```

Native operations have been inlined into the Signal code except the call to bitwise & operation the method >>. These are defined as external functions $f1$ and $f2$ defined by:

```
function f1 = (? i ! o)
  spec(| ^i = ^o | i → o |)
  pragmas C_CODE "o = i & 1" end pragmas;

function f2 = (?i !o)
  spec(| ^i = ^o | i → o |)
  pragmas C_CODE "o = i >> 1" end pragmas;
```

Chapter 10

Applications

We have introduced a type system allowing to model the control and data flow graphs of a given imperative program in SSA intermediate form. It is shown, that the expressive capability of the type system's semantics matches that of *de-facto* standard design languages such as SystemC and as well as that of related multi-clock synchronous formalisms (in particular SIGNAL). In order to really take advantage of such a type system, we need applications that extend its use and exploit its possible advantages. General applications of behavioral type systems such as optimization, verification, and encapsulation may be obvious, however we use this section to highlight details of some example applications. In the following we talk about the scalability of the approach and about its modularity. These are essential in treating large scale systems. Then we show how it can be used for design checking, design exploration, formal verification, and conformance checking. All of these applications are presented in this chapter briefly in order to get an idea of the impact of the approach in different areas.

10.1 Scalability

Just as the theory of interface automata [dAH01], types allow to scale the level of abstraction to be automatically obtained starting from the type inferred from a SystemC module within the simple formalism of Figure 8.2. Behavioral types share with interface automata the capability to define *static* interfaces (boolean relations) and *dynamic* interfaces (a transition system). Behavioral types relate a given proposition P to a more abstract one, Q, in several ways:

- a transition $e \Rightarrow x' = y$ can be abstracted by a clock relation between e, $\hat{x}$ and $\hat{y}$;
- a bound signal x in P/x can be abstracted by any proposition Q which contains P and does not reference x;

- a free signal x whose clock is not fast (because it appears at a lower level in the clock hierarchy) can be abstracted by this clock in any Q containing P.

All these examples are instances of a more generic abstraction pattern. In general, checking a user-specified abstraction Q consistent with the type P inferred from a given program amounts to the satisfaction of the containment relation $[\![\hat{P}]\!] \subseteq [\![Q]\!]$ (i.e. the denotation of P is contained in the denotation of Q). If P is a *static interface*, this amounts to the satisfaction of a boolean equation. Similarly, checking that a *dynamic interface* Q is an abstraction of a process P amounts to verifying that Q simulates P by model-checking.

10.2 Modularity

The main advantage of formulating a behavioral type system for SystemC is the formal foundation it offers to investigate modular and compositional design methodologies using separate compilation techniques. For instance, suppose that the declared type P of a SystemC class template provides sufficient information about its formal parameters that it can be determined if the body if the class template is controllable and can be compiled.

One may then provide it with an actual class parameter, of type Q, satisfying $Q \Rightarrow P$, without having the burden of fully instantiating the template code and recompile its code for that given instance.

Example 6 *To exemplify the benefits of behavior type inference for SystemC modules, let us first consider the dynamic type P of a counter modulo 2. It generates an output event y upon two occurrences of the signal x. Its state s is initially false. The signal x triggers the calculation of the next value s' of s defined by the negation $\neg s$ of its current value. When s is true then y is triggered.*

$$P \stackrel{\text{def}}{=} \left(\neg s^0 \,|\, \hat{x} \Rightarrow (s' = \neg s) \,|\, s \Rightarrow \hat{y}\right)$$

Let $Q = (\hat{x} > \hat{y})$ be the static interface type of P declared for the virtual class of the introductory example, Section 9.1. It just requires the clock $\hat{y}$ to be a sampling of $\hat{x}$. Let us associate the type P and Q with the procedure f using the

#TYPE *annotation.*

$$
\begin{array}{l}
\mathsf{class}\, m_0 \,\{ \\
\quad \mathsf{virtual}\, \mathsf{sc\_clock}\, x; \\
\quad \mathsf{virtual}\, \mathsf{sc\_clock}\, y; \\
\quad \mathsf{virtual}\, \mathsf{void}\ f()\ \{\}\ \#\mathsf{TYPE}(f, Q)\,\}; \\
\\
\mathsf{template}\, \langle \mathsf{class}\, m_1 \rangle\, \#\mathsf{TYPE}(m_1, m_0) \\
\quad \mathsf{SC\_MODULE}(m_2)\ \{\mathsf{SC\_CTOR}(m_2)\ \{ \\
\quad\quad \mathsf{SC\_THREAD}(m_1.f)\, \mathsf{sensitive} \ll x \\
\quad \}\};
\end{array}
$$

Let us now briefly recall the example of Section 9.1. First, an interface m_0, with virtual class fields x, y, and f is declared with the annotation #TYPE(f, Q)*. The template class m_2 uses a class parameter m_1 that implements m_0 to launch a thread $m_1.f$ sensitive to x. The class m_3 implements the interface m_0 and defines the method f by the program pgm of type P. Finally, class m_4 is defined by the instantiation of the template m_2 with the parameter m_3.*

$$
\begin{array}{l}
\mathsf{class}\, m_3 \,\{ \\
\quad \mathsf{sc\_clock}\, x; \mathsf{sc\_clock}\, y; \\
\quad \mathsf{void}\ f()\ \{\ pgm\}\ \#\mathsf{TYPE}(f, P)\,\}; \\
m_2\langle m_3 \rangle\, m_4
\end{array}
$$

Checking the guarantees of the actual parameter m_3 satisfy the assumptions of the formal parameter m_0 of the template m_2 amounts to verifying that P implies Q. This is done by calculating the static abstraction $\hat{P} = \exists s.\, (\hat{s} = \hat{x} \,|\, \hat{y} = s))$ of P and by checking that it implies $\hat{x} > \hat{y}$.

10.3 Design Checking

The proposed type system allows to easily formulate properties pertaining to common design errors the analysis of which has been the subject of numerous related works. Most of these approaches consist of proposing an ad-hoc type system for analyzing a specific pattern of design errors: race conditions, deadlocks, threads termination; and in a given programming language: Java, C, SystemC. By contrast, our behavioral type system provides a unified framework to perform both static verification via satisfaction checking or dynamic verification via model checking of behavioral properties of embedded systems described using imperative programming languages. The inference technique itself is language independent and the semantical peculiarities of language-specific runtime features and libraries can be modeled in the polychronous model of computation and its supportive type system.

Termination. One common design error found in embedded system design is the unexpected termination of a thread due to, e.g., an uncaught exception.

In our behavioral model of SystemC, the termination of the infinite loop of a thread f can be represented by the property $x_f^{exit} = 1$. Unexpected termination can hence be avoided by model-checking the property that $x_f^{exit} = 0$ is an invariant of f:

$$P \models x_f^{exit} = 0$$

Deadlocks. Another common design error is a wait statement that does not match a notification and yields the thread to block. Let $x_{L_{1\ldots n}}$ be the clocks of the blocks $L_{1\ldots n}$ in which the lock x is notified. Waiting for x at a given label L eventually terminates if

$$P \models x_L \wedge \neg(\wedge_{i=1}^{n} x_{L_i}) = 0$$

Race conditions. Similarly, concurrent write accesses to a variable x shared by parallel threads can be checked exclusive by considering the input clocks $e_{1,..n}$ of all write statements $x = f(y, z)$ by verifying that

$$P \models (e_i \wedge (\vee_{j \neq i} e_j)) = 0, \forall i = 1, ..n$$

Example 7 *For instance, consider checking exclusion between the transitions of the* ones *counter. The type P of the counter implies the equations $x'_{\mathsf{L2}} = (\hat{y}_1 \vee \hat{y}_2)$ and $x'_{\mathsf{L1}} = x_{\mathsf{L3}}$. Verifying exclusion between them amounts to proving that $(\hat{y}_1 \vee \hat{y}_2) \wedge x_{\mathsf{L3}} = 0$ is an invariant of P. By construction of $\overline{P}$, we have: $\hat{y}_1 = x_{\mathsf{L1}} \wedge (\mathsf{lock} \neq \mathsf{lock}')$, $\hat{y}_2 = x_{\mathsf{L2}} \setminus \mathsf{T0}$ and $x_{\mathsf{L3}} = \mathsf{T0}$. The property follows by observing that $\hat{\mathsf{T0}} = x_{\mathsf{L2}}$.*

10.4 Design Exploration

Just as the multi-clocked synchronous formalism of SIGNAL which it is based upon, our type system allows for the refinement-based design methodologies considered in [TGS+03] to be easily implemented.

Checking the correctness of the refinement of an initial SystemC module, of type P, by its refined version, of type Q, amounts to verifying that the final type Q satisfies the assumptions made by the initial specification. In the spirit of the refinement-checking methodology proposed in [TGS+03], this can be implemented by checking the refinement Q to be finitely flow preserving the initial design P.

In general, Q may differ from P by the insertion of a protocol between two components of P, by the adaptation of the services provided by P with a new functionality implemented in Q. Along the way, one may abstract from the type Q

the signals and state variables introduced during the refinement in order to accelerate verification. In most cases, such refinements may be checked incrementally, by checking the static containment relation between the static abstractions of P and Q: $\hat{Q} \Rightarrow \hat{P}$.

10.5 Systematic Formalization of Specification-Level Behavior

This approach consists, first, of decomposing the syntactic structure of a program into an intermediate representation that renders the imperative structure of the original program together with its most characteristic features (use of locks, interrupts, etc.). In this structure, each thread consists of a sequence of blocks delimited by wait and notify synchronization statements.

An example implementation of this has been published in [TGB+03]. It consists of a Polychrony plugin that translates multi-threaded real-time JAVA programs into SIGNAL. The JVM real-time runtime system is modeled using the ARINC library of SIGNAL [GG02]. This library gives a generic model of real-time operating systems APIs in SIGNAL. The translator entirely models the behavior of a multi-threaded real-time JAVA component and reuses and reconfigures its package of real-time thread classes according to a given target architecture.

Within such blocks, basic control structures are then encoded. A method call or a basic operation, e.g. $x = y + 1$ with y declared as int $y = n$, is encoded by an equation, e.g. either $x = \mathsf{pre}\ n\ y + 1\ \mathsf{when}\ c$ (when y references a value computed during the previous transition in this block) or $x = y + 1\ \mathsf{when}\ c$ (if it has already been computed in the same transition), conditioned by an activation clock c. A conditional statement, e.g. $\mathsf{if}\ x\ \mathsf{then}\ P\ \mathsf{else}\ Q$, is encoded by constraining the clock of P by x and that of Q by $\mathsf{not}\ x$. While loops are encoded by over-sampling. Interrupts are rendered by events. An interrupt conditions the activation clock of subsequent equations in the control flow graph; if it escapes the scope of the method in which it is raised, it becomes an output signal of the process that contains the method in order to function outside of the context of that method.

In the specification-layer of the behavior ones, there is only one critical section, delimited by a wait and a notify. It is encoded much like the polychronous specification of the previous section, with the noticeable addition of the wait-notify protocol and the simulation scheduling tick. The process is activated when it obtains the lock on istart. Then, at its own rate (now conditioned by the clock c), it determines the count. When it is finished, it sends the notification.

10.6 Conformance Checking

Another application domain for the polychronous model is conformance checking. Conformance consists basically in formally comparing two versions of a model and deciding if the second model conforms to the description given by the first. This is specifically useful when performing model refinements and describing systems at different levels of abstraction. It is important to show that a refinement of a model still contains all the behavior described in the original model, that is *conforms* to the previous description.

In [TGS+03] the theory of refinement checking has been elaborated for examples in the system level design language SpecC and their SIGNAL equivalents. The SpecC design process inherently uses several levels of abstraction such as the specification level, architecture level, communication level, and implementation level. These different design layers can similarly be found in all ESL design methodologies. While SpecC and others guarantee a correct by construction refinement process, this often is less obvious than they claim. The formalism built around polychrony permits to make formal checks on the conformance of these refinements. Several design properties have been identified and formally defined such as relaxation, flow equivalence, flow invariance, and controllability [TLS+04] that permit to perform formal conformance checks between refinement steps.

Fourth Part

Using Formal Methods for Embedded Systems

Chapter 11

Introduction

In Chapter 7 we have presented some general ideas on how to integrate formal methods in a codesign flow. In this chapter we illustrate the details and benefits of such an integration, and show how much of the complexity of formal methods can be hidden without losing the gain in design accuracy.

SystemC [OSC] is one of the most popular languages for Electronic System Level (ESL) design. It has the big advantage that is is based on C++; any SystemC program can be compiled with a standard C++ compiler such as GCC. Its syntax is relatively easy to understand and to program for the large base of available engineers that do know C++ or Java. SystemC is adding some additional concepts to C++ that are needed for the development of hardware systems. The main concepts added in SystemC are true concurrency, hierarchical modules, signals, communication channels, and specific data types. All of this additional functionality is added with the help of a class library and macros, so as to not to alter the language itself. This is one big advantage it has over SpecC [GZD+00], which is based on the C language but where language constructs have been added, necessitating specific tools for any manipulation of the code.

While the strength of SystemC is its ease of expression and the large available code base, one major flaw however is the lack of any formal support, which is the case for most ESL design languages. In order to check if a portion of SystemC code is doing what it is intended to do, the only tool we can rely on is simulation. By simulation we mean in this context to compile an executable binary of the code in question, to execute it with a certain amount of input values, and finally to check if the output values correspond to the expected result. The significance of simulation results largely depends on the number and choice of input vectors. Also systems can be designed to contain test routines that help to discover problems in the design. However, simulation can never verify correct functioning of the system for all input values. And as the complexity of models increase, the percentage of possible input vectors verified decreases exponentially.

Formal methods have the ability to verify the correct behavior of a specific functionality for all possible states of the system. It would therefore be desirable to have an ESL design language that is based on a formal semantic, in order to be able to apply formal correctness checks on the design. The complexity of formal languages, however, make it very inconvenient to express system behavior and hence prevent their use by design engineers. We were thus looking into the possibility to translate existing non formal ESL design models into a formal language, and then verify their functioning using formal methods. This translation process may be time consuming and error prone when performed by hand. But once the translation scheme is defined properly it can be automated and deliver a correct formal model with few additional effort.

As output formalism we choose the synchronous language SIGNAL [BLJ91]. SIGNAL is a data flow oriented synchronous language, able to represent polychronous models, i.e. models with multiple independent clock domains. This enables a maximum liberty in modeling systems, especially large distributed systems and does not impose synchronization of all components prior to implementation. SIGNAL is part of the Polychrony framework [IRI]. Polychrony comprises other tools such as a graphical editor and viewer and the symbolic model checker SIGALI [MBLL00].

When trying to represent a SystemC program in SIGNAL, we face two challenges. The first and most obvious one is to correctly translate the SystemC behavior into SIGNAL syntax. From such an automated straight forward translation, we obtain a SIGNAL program that represents the original SystemC description. However it will hardly represent the original structure of the SystemC model with its modules and connections. A simple automated translation is ignoring the structure of the program and producing a flat program. This is still a formal representation of the system and can be used for verification purposes. However, not having the structure of the system prevents benefit from the whole potential of the transformation. The structure is needed for modular verification, indispensable when the system size is getting too big. In addition to that it also is a key to optimizations that the SIGNAL compiler could perform for example in scheduling. Another advantage of having structural information is that it makes the resulting code more readable and enables manual changes or optimizations in the formal model directly. Being in the possession of structural information can lead as well to other uses such as visualization, automated test generation, or introspection. We therefore cut down the problem into two parts, the flat translation of SystemC code into SIGNAL and the extraction of the structure of SystemC projects, its translation into SIGNAL and its exploitation for other uses. Handling these two problems separately reduces the complexity, and while the respective results can be put together easily in the end, each part has a substantial benefit on its own.

11.1 Translating SystemC Behavior using SSA

For the translation of SystemC behavior we benefit from the fact that any SystemC program is a valid C++ program. As SystemC is a subset of C++, it is from a semantic point of view almost the same to translate SystemC or to translate C++. In order to avoid the complex task of creating an own C++ parser, we use an existing open source compiler front end from the GNU Compiler Collection (GCC). GCC is transforming the input C++ into a simpler structure, the Single Static Assignment (SSA) intermediate format. As the control structure of GIMPLE/SSA is less complex than that of C++ code, it is much easier to translate it into a different formalism.

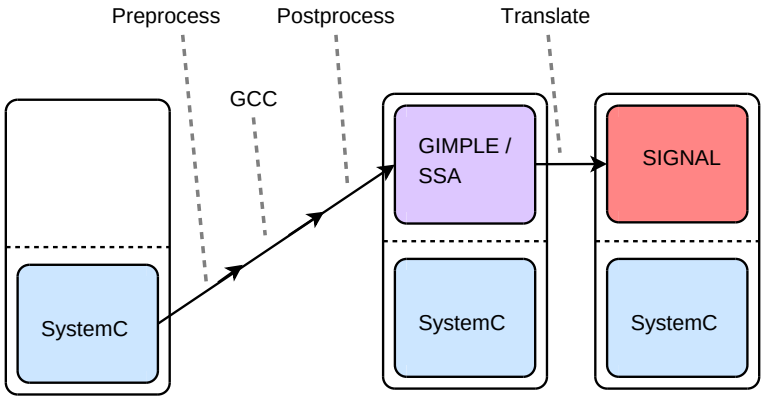


Figure 11.1: Translation of SystemC modules into SIGNAL

Figure 11.1 depicts the design flow for this approach. In a first step we use GCC to transform the model into an intermediate SSA form. Since GCC is not aware of the meaning of SystemC macros such as *sc_module* or *wait(signal_x)*, it simply expands them and then handle the resulting code. To keep the representation of these SystemC commands also in the SIGNAL code, we perform a preprocessing step to prevent GCC to expand them, and make them reappear in the SIGNAL code during a post processing step after the transformation. Then, where necessary, the corresponding macros have to be implemented in SIGNAL in order to keep the resulting SIGNAL code syntactically close to the original SystemC implementation. Once in SSA form, the model is translated into SIGNAL code with the help of the translation scheme specified in Section 12.1

[BTSG04] shows how, based on the theory of [TBS+04a], formal behavioral types can be automatically generated. With the help of a case study about a finite impulse response filter (FIR), we demonstrate the whole flow to transform SystemC components into a SIGNAL description. It also shows how GCC and its intermediate format GIMPLE/SSA can be used to facilitate this transition.

This approach is then presented more in detail in [TBS+04b]. What is missing in these publications, however, is the preservation of the program architecture, which is completely lost or only very hard to recover. The work is therefore useful mainly for treating single components.

11.2 SytemCXML

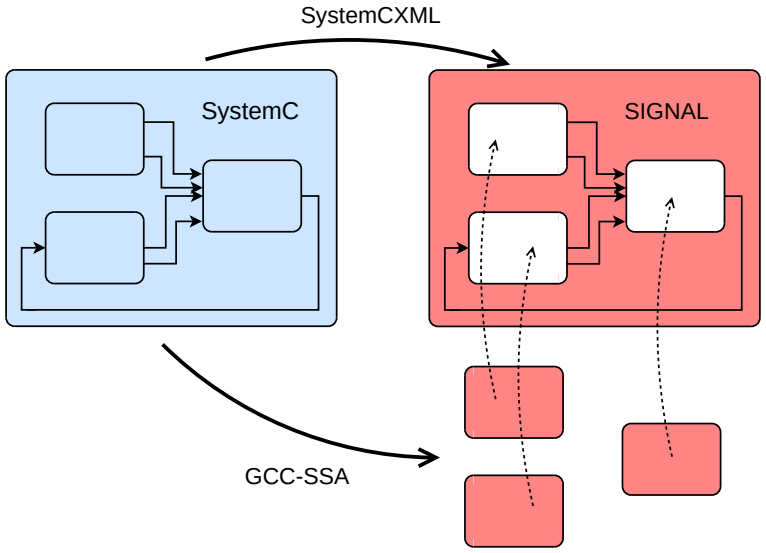


Figure 11.2: Methodology for translating SystemC models into SIGNAL

In order not to lose the structural information of the original SystemC code such as the module organization and their connections, we developed a methodology in which in a first step the structural information is directly extracted from the SystemC code. We then translate the modules one by one with the help of the translation scheme described in Chapter 12, and finally we generate a SIGNAL architecture skeleton. The empty boxes in the skeleton code can then be filled with the results from the module translation (Figure 11.2). Using this high level control information the SIGNAL compiler can perform high level optimizations such as a more efficient scheduling.

The structural model obtained from SystemC programs is extracted into an XML structure with the list of modules, their hierarchy and their connections. This is precious information - even without conversion into a formal framework - and can be used for numerous other purposes such as the visualization of certain aspects of SystemC projects and the automated generation of test vectors.

Our solution is illustrated in 13, where we are presenting a front end for SystemC that can be used to analyze entire SystemC projects, extract their structural information and expose this information in an easily accessible XML format. Starting from this XML structural description we generate a formal SIGNAL description, representing the structure of the SystemC code. With the help of GCC and the SSA to SIGNAL translation scheme defined earlier we can then fill the empty module declaration with the module behavior. We also show, how the XML form containing the structural information can be easily used for other applications such as the visualization of designs or the automated generation of tests.

11.3 Contributions

One of the main contributions of this part consists in considering the applicability of the polychronous model of computation [LTL03] in the context of system-level design languages such as SystemC [OSC, GLMS02]. We provide imperative system components with formal behavioral interfaces that can be abstracted to different levels of detail. These interfaces can be used to prove formal properties of the components as well as on the composition of components. We propose a technique to generically obtain formal abstractions of SystemC components and use the power of the synchronous paradigm to formally verify the correctness of component compositions. In particular we present a case study detailing all steps from the analysis and translation of SystemC components to building behavioral types for these components and synchronously composing them to detect possible flaws. As a side note we show how these interfaces can be used to verify formal properties of the components and their composition.

Another contribution is the demonstration of how the extraction of structural information from IPs can open a path for a plethora of validation applications. We specifically show for the SystemC language how with the use of open source tools such as Doxygen and XML, the model architecture can be relatively easily obtained and used for applications such as visualization, automated test generation, design browsing, and runtime reflection.

11.4 Related Work

The capture of the behavior of a system through a type theoretical framework relates our technique to the work of Rajamani et al. [RR01], and many others, on abstracting high-level and concurrent specifications, for example the π-calculus, by using a formalism, e.g. Milner's CCS, in which, checking type equivalence is decidable. This work contrasts from related studies by the capability to capture

scalable abstractions of the type-checked system. In our type system, scalability ranges from the capability to express the exact meaning of the program, in order to make structural transformations and optimizations on it, down to properties expressed by Boolean equations between clocks, allowing for a rapid static-checking of design correctness properties.

We share the aim of a scalable and correct-by-construction exploration of abstraction-refinement of system behaviors with the work of Henzinger et al. on interface automata [dAH01]. Our approach primarily differs from interface automata in the data-flow formalism used in the Polychrony workbench where partial clock and scheduling relations express the multi-clocked synchronous behavior of the system. Compared to a common automata-based approach, our declarative approach allows to hierarchically explore abstraction capabilities and to cover design exploration with the methodological notion of refinement along the whole design cycle of the system, and can be used from the requirements specification up to synthesis and code-generation [TLS+04, LTL03].

11.4.1 Ptolemy

Ptolemy [BHLM94, LL98] is another an example for a design environment that allows hardware and software components to be integrated from the specification through the synthesis, simulation, and evaluation phases. Its focus is on the assembly of concurrent components and the key underlying principle in the project is the use of well-defined models of computation that govern the interactions between components. A major problem area being addressed is the use of heterogeneous mixtures of models of computation; it supports more than 10 different domains including Communicating Sequential Processes (CSP), Process Networks (PN), Synchronous Data Flow (SDF), and Discrete Event (DE). Ptolemy II [LX01] is a more recent incarnation written in Java. It has a more extensive user interface and works on multiple platforms.

In comparison to the approach presented here, Ptolemy is lacking however an interface to existing model components written in a popular design language such as SystemC.

11.4.2 POLIS, Metropolis

Another framework for Hardware-Software CoDesign of embedded systems is POLIS [BGJ+97]. The POLIS system is centered around a single Finite State Machine-like representation called Co-design Finite State Machine (CFSM). The difference to a classical FSM is that the synchronous communication model is replaced in the CFSM model by a finite, non-zero, unbounded reaction time. This model of computation can also be described as Globally Asynchronous, Lo-

cally Synchronous. The specification is a priori unbiased towards a hardware or software implementation. Each machine can then be independently mapped into software or into hardware by means of ad-hoc synthesis tools. POLIS uses the formal specification language Esterel [Ber00, BG92] for design capture. Starting from a whole unified design flow with formal verification, co-simulation (using Ptolemy), design partitioning, hardware synthesis, and software synthesis. Interfaces between different implementation domains are automatically synthesized. POLIS is one of the pioneering tools for the separation of functionality and architecture, and has been implemented in several commercial applications that are based on the CFSM model of computation of POLIS, Virtual Component Codesign (VCC) of Cadence Design Systems being the most prominent.

The natural successor of Polis, Metropolis [BWH+03] started in 1997 from the idea of freeing the designer from the too strict CFSM model of computation. It is based on a metamodel with formal semantics that can be used to capture designs. It supports simulation, formal analysis, and synthesis. It is based on the same ideas than that of POLIS, but implements them more flexibly. Communication primitives and execution rules can be selected depending on the application. Metropolis can generate SystemC code from the model's abstract syntax tree, which then can be compiled and executed. Also SystemC and Metropolis models can be co-simulated, however currently there is no way to capture existing SystemC IP into the Metropolis framework in order to benefit from the modeling and verification infrastructure.

11.4.3 Existing Tools for Structural Reflection

Several tools may be used for implementing structural reflection in SystemC. Some of them are SystemPerl [Sny], EDG [FE], or C++ as in the BALBOA framework [FSG03]. However, each of these approaches have their own drawbacks. SystemPerl and gSysC [EAH05] for instance, require the user to add certain hints into the source file and although SystemPerl handles all SystemC structural information, it does not recognize all C++ constructs. EDG is a commercial front-end parser for C/C++ that parses C/C++ into a data structure, which can then be used to interpret SystemC constructs. However, interpretation of SystemC constructs is a complex and time consuming task, plus EDG is not to be freely used in public domain. BALBOA implements its own reflection mechanism in C++ which again only handles a small subset of the SystemC language. Other approaches such as [GLLA03] require modifications of the SystemC libraries.

In [PMBS06] we position our structural information extraction into a wider context. We describe a service oriented architecture for the introspection and validation of system level designs. The architectural information extracted from

SystemC is used for a platform that offers services such as introspection and reflection, interactive debugging, visualization, and the generation of automated tests. The platform is built in a way that allows the easy addition of additional services.

11.4.4 ESys.NET Framework

ESys.NET [LAN+03] uses an interesting idea where they implement a composite design pattern for datatypes into the original source code. Their work is inspired by the .NET framework's reflection mechanism. ESys.NET enhances SystemC's datatype library by implementing the design pattern with additional C++ classes. This altered datatype library introduces member functions that provide introspection capabilities for the particular datatypes. However, this requires altering the datatype library and altering the original source code to extract structural information.

11.4.5 BALBOA Framework

The BALBOA [FSG03] framework describes a framework for component composition, but in order to accomplish that, they required reflection capabilities of their components. They also discuss some introspection mechanisms and whether it is better to implement reflection at a meta-layer or within the language itself. We limit our discussion to only the approach used to provide reflection in BALBOA.

BALBOA uses their BIDL (BALBOA interface description language) to describe components, very similar to CORBA IDLs [OMG]. Originally IDLs provide the system with type information, but BALBOA extends this further by providing structural information about the components such as ports, port sizes, number of processes, etc. This information is stored at a meta-layer (a data structure representing the reflected characteristics). BALBOA forces system designers to enter meta-data through BIDL which is inconvenient. Our method can directly process the SystemC models.

One characteristic of this framework is the implementation of a dedicated description language, BIDL. The designer writes the BIDL for specifying the reflected structure information which could more easily (and less error prone) be retrieved automatically from SystemC source, as it is done in our approach.

11.4.6 Pinapa and LusSy

Pinapa [MMMC05b, Gre] is a recently released Open Source SystemC front end that builds upon GCC's front end to parse all C++ constructs and infer the structural information of the SystemC model by executing the elaboration phase,

which is a very attractive solution. SystemC's elaboration constructs all the necessary objects and performs the bindings after which a regular SystemC model begins simulation via the `sc_start` function. Instead, Pinapa examines the data structures of SystemC's scheduler and creates its own IR. This solves the SystemC parsing issue, it requires, however, modifications of the GCC source code which makes it (i) dependent on changes in the GCC codebase, and (ii) forbids the use of any other compiler. While this approach in contrast to others such as SystemPerl and gSysC does not require to modify the SystemC models, it requires modifications of the SystemC libraries.

Pinapa is part of LusSy [MMMC05a, Moy05], a toolbox for the formal verification of SystemC models. LusSy is specifically targeting SystemC models on the transaction level. The IR obtained from Pinapa connects a symbolic model checker and an abstract interpretation tool. It is also used as a basis for a series of optimizations and can generate code for the formal frameworks Lustre [HCRP91] and SMV [McM93].

11.4.7 Java, C# .NET Framework, C++ RTTI

Here, we discuss some existing languages and frameworks that use reflection capabilities. There are Java, C# and the .NET framework and C++ RTTI. Java's reflection package `java.lang.reflect` and .NET's reflection library `System.Reflection` are excellent examples of existing reflection concept implementations. Both of these supply the programmer with similar features such as the type of an object, member functions and data members of the class. They also follow a similar technique in providing reflection, so we take the C# language with .NET framework as an example and discuss in brief their approach. C#'s compiler stores class characteristics such as attributes during compilation as meta-data. A data structure reads the meta-data information and allows queries through the `System.Reflection` library. In this infrastructure, the compiler performs the reflection and the data structure provides mechanisms for introspection.

C++'s runtime type identification (RTTI) is a mechanism for retrieving object types during execution of the program. Some of the RTTI facilities could be used to implement reflection, but RTTI in general is limited in that it is difficult to extract all necessary structural SystemC information by simply using RTTI. Furthermore, RTTI requires adding RTTI-specific code within either the model, or the SystemC source and RTTI is known to significantly degrade performance.

11.4.8 Doxygen, XML, Apache's Xerces-C++

Two main technologies we employ in our solution for obtaining and exploiting structural information from SystemC models are Doxygen and XML. Doxy-

gen [vHT] is a documentation system primarily for C/C++, but has extensions for other languages. Since SystemC is simply a library of C++ classes, it is ideal to use Doxygen's parsing of C/C++ structures and constructs to generate XML representations of the model. In essence Doxygen does most of the difficult work in tagging constructs and also documenting the source code in a well-formed XML. By using XML parsers from Apache's Xerces-C++ we can parse the Doxygen XML output files and obtain any information about the original C / C++ / SystemC source. In [BPM+05b, BPM+05a] we describe the front end tool using these techniques that we call SystemCXML. The article [PBMS04] describes a service oriented architecture using that.

Chapter 12

Modular Verification of SystemC Components

In this chapter we show how to provide components with formal interfaces that not only comprise the component's input / output signals and their types but also causal and synchrony relations between signals. This enables more exhaustive checks for the compatibility of components. The description of interface signal dependencies can also be seen as a behavioral interface. Figure 12.1 shows the connection of two components. A normal type description only checks information about the data type of the common signals `x, y,` and `z`. As long as these types match, the type checker will approve the composition. If for example A produces `x` and `y` at the same rate but B consumes two values of `x` for each value of `y`, this would go undetected. Also there is no means to discover a combinational loop over the signal `z`. A behavioral type description that holds information about the synchronization of signals is able to detect these errors. It exhibits part of the internal functioning of the component in a data-flow synchronous formalism that makes it possible to formally reason about these interfaces.

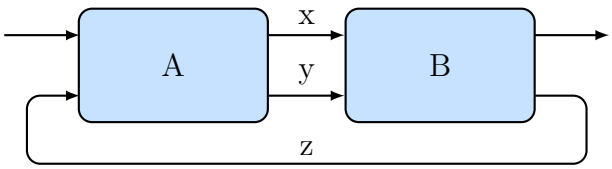


Figure 12.1: Two connected components

The synchronous composition of several of these behavioral interfaces automatically reveals intricate problems in the composition of the components just as a simple type checker would find a signal data type mismatch. Many of these errors would otherwise remain undetected and could cause great costs and delays

later in the design process. The more behavior such an interface captures, the more behavioral errors can be found. A system built from components whose compositions all have been checked with the help of a behavioral type, can therefore be expected to function more reliably and have a much higher overall design quality than compositions with simple data type checks [DSG03] - even after thorough testing.

12.1 Methodology and Tools

Our modeling and verification methodology starts off with a SystemC model of a system. The goal is to provide all system components with a formal behavioral type that can be used to discover errors in the composition of components. The formal type can also be used to formally verify properties of the components and their composition. The methodology consists of several steps. First the SystemC code is analyzed in a preprocessing step and some types are replaced for better conversion results. Then a static single assignment intermediate representation is generated. From this representation, clock and scheduling relations are extracted that serve as a basis for the generation of SIGNAL code. The compilation of the signal code performs static checking for types, dependencies, and clock constraints. This results in a highly reliable connection of components as the synchronous composition - once successfully performed - rules out many sources of error that are not checked for in a common type checking system.

Any SIGNAL program also represents a formal model and can therefore also be used to check for dynamic properties. This opens the gate for formal verification with reasonably additional effort. For the model checker we use, the model has to be abstracted or transformed into a Boolean version. The remainder of this section describes some of the tools used throughout the process. In Section 12.2 all of these steps can be followed in more detail using an example of an FIR filter.

12.1.1 Static Single Assignment Form and GIMPLE

As SystemC programs are written in C++, they can contain very complex constructs, which make it difficult to obtain a corresponding SIGNAL representation. It is obvious that it would be much easier to make a translation from a more low level representation or from a C++ subset that does not make use of all the complex constructs. Not wanting to restrict the input language, we are using an intermediate representation that fits our needs.

GIMPLE [Mer03] is a simple intermediate representation developed at McGill University [HDE+93] and has now been adopted in the GNU Compiler Collection (GCC) [Fre04b]. GIMPLE is a three address C-like language with no high level

control structures. Some of its particularities are that GIMPLE statements - with the exception of function calls - contain not more than three operands and have no side effects, intermediate values are stored in temporary variables, and all control structures are lowered to conditional gotos.

Most of the GCC optimization passes use the data flow information provided by the static single assignment form (SSA) [CFR+91]. This is an intermediate representation in which every variable is assigned exactly once. It is particularly used for high level compiler optimization. This makes it particularly useful for our purpose, since static single assignments have a very regular structure, they can be easily manipulated by automatic program analysis tools, and result in a natural translation to the SIGNAL synchronous formalism.

GIMPLE and SSA are part of the Tree-SSA [Fre04a] changes that have been integrated into the GNU Compiler Collection (GCC) starting from version 4.0, released in April 2005. These changes allow for language independent, higher level optimization passes. By using GIMPLE-SSA as an intermediate representation for the behavioral type generation, we can therefore benefit from all current and future optimization passes implemented in the GCC. GIMPLE-SSA code can be dumped with compiler options corresponding to different levels of optimization, such as `-fdump-tree-ssa`, `-fdump-tree-gimple` and `-fdump- tree-optimized`. The last option gives the best results. When further automating the type extraction process, the tree representation of GCC can be used directly in shared memory without having to dump it to a file and to read it again.

12.1.2 Formal Verification of Component Properties

The SIGNAL compiler is an analysis and transformation tool that tries to generate executable code from the sum of equations the SIGNAL program represents. Before this code can be generated, the SIGNAL compiler checks for static problems such as contradictory clock constraints, cycles, and zero clocks. However, in order to check dynamic properties of the system, the SIGNAL companion model checker SIGALI [MBLL00] can be used. Given a formal model of a system in SIGNAL, SIGALI can verify formal properties of the model. It is an interactive tool specialized on algebraic reasoning in $\mathbb{Z}/3\mathbb{Z}$ logic.

SIGALI transforms SIGNAL programs into sets of dynamic polynomial equations that basically describe an automaton. It can analyze this automaton and prove properties such as liveness, reachability, and deadlock. The fact that it is solely reasoning on a $\mathbb{Z}/3\mathbb{Z}$ logic constrains the conditions to the Boolean data type (true, false, absent). This is practical in the sense that true numerical verification very soon would result in state spaces that are no longer manageable, however it requires, depending on the nature of the underlying model, major or minor modifications prior to formal verification.

For many properties numerical values are not needed at all and can be abstracted away thus speeding up verification. When verification of numerical manipulations is sought, an abstraction to boolean values can be performed, that suffices in most cases to satisfy the needs.

12.2 Case Study of an FIR Filter

This section exemplifies the presented approach with the design of a finite impulse response filter (FIR). It details the decomposition of a full featured SystemC specification into an SSA representation. The different analysis steps are demonstrated down to the final typed SIGNAL representation.

As a starting point, we use the SystemC model of the FIR from the SystemC 2.0.1 distribution [OSC] and translate it into SSA code. We show how this SSA code is analyzed and how clock and scheduling information can be extracted. In Section 12.2.4 the corresponding SIGNAL type is presented and it is shown how to obtain it with the preceding information.

12.2.1 The SystemC Model

In the SystemC model, the filter itself consists of one functional block surrounded by a testbench consisting of a Stimulus that generates input values and a Display module that receives the output and displays it on the screen (Figure 12.2).

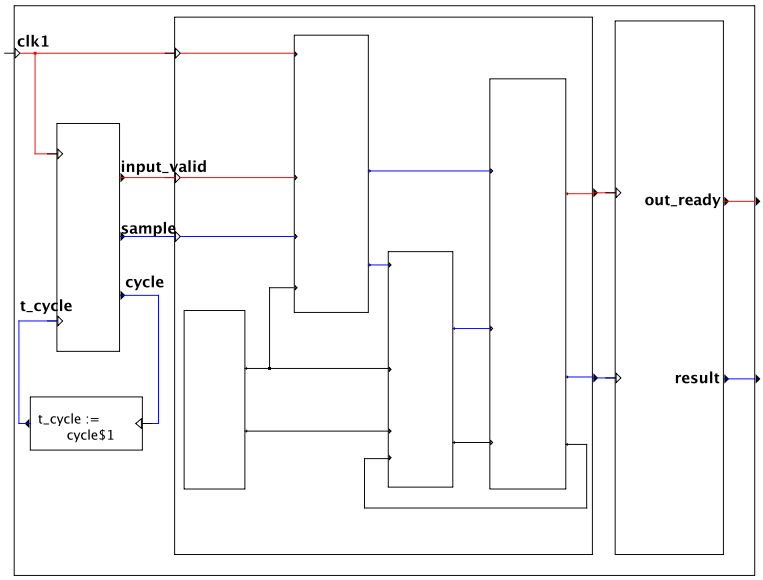


Figure 12.2: Structure of the FIR filter with testbench

The FIR unit is implemented as an SC_THREAD that is triggered on the positive clock edge. The other blocks are SC_METHODS. The left hand side of Figure 12.3 displays the SystemC code of the entry function for the FIR block. The first 10 lines just handle the initialization of variables. Then there is an infinite *while* loop that contains the actual filter functionality.

In short, it waits until there is a valid input available, reads this input, processes it, writes it to an output, and then notifies its environment that the result is available. At the end of each *while* loop it suspends itself until the next positive clock edge. The FIR result is the sum of the last 15 input values weighted with 15 coefficients. This is done in two *for* loops. The first one does the weighting and the second one is shifting the buffer array containing the last inputs.

Communication with the environment is done via *enable* signals. The Stimulus indicates with the signal *in_valid* that a new value is available. In the same way, the Display is sensitive to the variable *output_data_ready* that is set when a new output value is available.

12.2.2 Obtaining a GIMPLE-SSA Representation

The right hand side of Figure 12.3 shows the GIMPLE-SSA code that corresponds to the SystemC FIR. For the generation of a clean GIMPLE-SSA representation we follow three steps. First, preprocessing of the SystemC code, second, translation to GIMPLE-SSA with GCC, and third, post processing of the generated GIMPLE-SSA code. The direct generation of GIMPLE-SSA from SystemC can be done, but it results in very large and hardly readable code. A closer look reveals that most of this bloating is due to the SystemC types and statements, which are implemented as macros and get translated as well. If we replace the SystemC types by corresponding C++ types, e.g. *sc_int* is changed to *int* or *unsigned*, in a simple preprocessing step, the size of the generated code shrinks drastically.

More complex statements such as *wait(signal)*, however, still cause a considerable increase of the code size compared to the original SystemC code. We decide to simply comment these out in the SystemC source so they are ignored by the compiler and can later be taken care of separately in a post processing step.

During post processing we replace the *wait(signal)* statements by corresponding SSA statements. Logically a *wait* statement is similar to an *if* branch. Depending on a condition something is executed, otherwise something else. The condition is the signal that we are waiting for (e.g. *in_valid == true*. If there is no signal given, the process waits for the signal it is sensitive to (this is the positive edge of the clock in this example).

In order to be able to execute the *wait* statement separately, we have to introduce a separate label for it. As we can see on the right hand side of Figure 12.3,

```
void fir::entry() {
  sc_int<8> tmp;
  sc_int<17>pro;
  sc_int<19>acc;
  sc_int<8>shift[16];
  result.write(0);
  out_ready.write(false);

  for (int i=0; i<=15; i++)
   shift[i] = 0;
  wait();

  while(1) {
   out_ready.write(false);
   wait_until
    in_valid.delayed()==true;
   tmp = sample.read();
   acc = tmp*coefs[0];
   for(int i=14; i>=0; i–) {
    pro = shift[i]*coefs[i+1];
    acc += pro;
   };
   for(int i=14; i>=0; i–)
    shift[i+1] = shift[i];
   shift[0] = tmp;
   // write output values
   result.write((int)acc);
   out_ready.write(true);
   wait();
  };
}
```

```
void fir::entry() {
      int shift[16], i, acc, tmp;
      i=0; goto L1;
      this → result = 0;
      this→output_data_ready = 0;
L0:   shift[i]= 0
      i = i + 1;
L1:   t_i = (i<=15)
      if (t_i) goto L0;
      else goto L1a;
L1a:  wait (clk1_pos);
L3:   this→output_data_ready = 0;
L3a:  wait_until(in_valid == true);
L3b:  tmp = this→sample;
      acc=this→coefs[0]*tmp;
      i = 14;
      goto L5;
L4:   acc=acc+shift[i]
        *this→coefs[i+1];
      i = i - 1;
L5:   if (i >= 0) goto L4;
      else goto L6;
L6:   i = 14;
      goto L8;
L7:   shift[i + 1] = shift[i];
      i = i - 1;
L8:   if (i>=0) goto L7;
      else goto L9;
L9:   shift[0] = tmp;
      this→result = acc;
      this→output_data_ready = 1;
L9a:  wait (clk1_pos);
      goto L3;
}
```

Figure 12.3: SystemC and SSA code for the FIR core

for *L1*, *L1a* is introduced since the wait statement is not at the beginning of the block, and for *L3*, there are two additional labels, *L3a* and *L3b* because this wait statement is in the middle of a block.

12.2.3 Extracting Clock and Scheduling Information

Though slightly bigger in size, the SSA representation has several advantages with respect to automated analysis and conversion: it consists of very simple and repetitive statements, it is separated into sequential blocks without branches and where variable are assigned once. The extracted behavioral type information can be separated into two parts, control and data flow.

Figure 12.4 displays this information for the FIR in the form of a synchronous transition system (iSTS, as in [TBS+04a]). It consists of propositions on clocks $\hat{}x$ guarded by block input clocks x_L for implications $x_L \Rightarrow \hat{}x$ and of propositions on state transitions of the form $e \Rightarrow x'_L$ to mean that if e is present then x_L is the next block to be executed.

$$
\begin{array}{ll}
x_{\mathsf{fir}} \Rightarrow \hat{}\,i & x_{\mathsf{L4}} \Rightarrow \hat{}\,acc \\
\quad \hat{}\,result & \quad \hat{}\,shift \\
\quad \hat{}\,out\_ready & \quad \hat{}\,coefs \\
\quad x'_{\mathsf{L1}} & \quad \hat{}\,i \\
x_{\mathsf{L0}} \Rightarrow \hat{}\,shift & \quad i \rightarrow shift \rightarrow acc \\
\quad \hat{}\,i & \quad i \rightarrow coefs \rightarrow acc \\
\quad i \rightarrow shift & \quad x'_{\mathsf{L5}} \\
\quad x'_{\mathsf{L1}} & x_{\mathsf{L5}} \Rightarrow (i \geq 0) \Rightarrow x'_{\mathsf{L4}} \\
x_{\mathsf{L1}} \Rightarrow \hat{}\,t\_i & \quad (i < 0) \Rightarrow x'_{\mathsf{L6}} \\
\quad \hat{}\,i & x_{\mathsf{L6}} \Rightarrow \hat{}\,i \\
\quad i \rightarrow t\_i & \quad x'_{\mathsf{L8}} \\
\quad t\_i \Rightarrow x'_{\mathsf{L0}} & x_{\mathsf{L7}} \Rightarrow \hat{}\,shift \\
\quad \neg t\_i \Rightarrow x'_{\mathsf{L1a}} & \quad \hat{}\,i \\
x_{\mathsf{L1a}} \Rightarrow (clk1 \neq clk1') & \quad i \rightarrow shift \\
\qquad \wedge\, clk1 \Rightarrow x'_{\mathsf{L3}} & \quad x'_{\mathsf{L8}} \\
\quad \neg x'_{\mathsf{L3}} \Rightarrow x'_{\mathsf{L1a}} & x_{\mathsf{L8}} \Rightarrow \hat{}\,i \\
x_{\mathsf{L3}} \Rightarrow \hat{}\,out\_ready & \quad (i \geq 0) \Rightarrow x'_{\mathsf{L7}} \\
\quad x'_{\mathsf{L3a}} & \quad (i < 0) \Rightarrow x'_{\mathsf{L9}} \\
x_{\mathsf{L3a}} \Rightarrow \hat{}\,in\_valid & x_{\mathsf{L9}} \Rightarrow \hat{}\,shift \\
\quad in\_valid \Rightarrow x'_{\mathsf{L3b}} & \quad \hat{}\,tmp \\
\quad \neg in\_valid \Rightarrow x'_{\mathsf{L3a}} & \quad \hat{}\,result \\
x_{\mathsf{L3b}} \Rightarrow \hat{}\,tmp & \quad \hat{}\,acc \\
\quad \hat{}\,sample & \quad \hat{}\,out\_ready \\
\quad \hat{}\,acc & \quad tmp \rightarrow shift \\
\quad \hat{}\,coefs & \quad acc \rightarrow result \\
\quad \hat{}\,i & \quad result \rightarrow out\_ready \\
\quad sample \rightarrow tmp & \quad x'_{\mathsf{L9a}} \\
\quad coefs \rightarrow acc & x_{\mathsf{L9a}} \Rightarrow (clk1 \neq clk1') \\
\quad tmp \rightarrow acc & \qquad \wedge\, clk1 \Rightarrow x'_{\mathsf{L3}} \\
\quad x'_{\mathsf{L5}} & \quad \neg x'_{\mathsf{L3}} \Rightarrow x'_{\mathsf{L9a}}
\end{array}
$$

Figure 12.4: Clock and scheduling relations for the FIR

In order to understand how these clock relations are obtained, we have to take a look at the SSA form in Figure 12.3. For instance, $x_{\mathsf{L0}} \Rightarrow \hat{}\,shift$ means that whenever block L0 is entered, the signal *shift* has to be present. Transitions from one block to another are represented like this: $x_{\mathsf{L4}} \Rightarrow x_{\mathsf{L5}}$. However, if in the following block a signal is assigned that has already been assigned in the current block, it cannot be executed in the same cycle. The time has to be advanced, this is expressed in $x_{\mathsf{fir}} \Rightarrow x'_{\mathsf{L1}}$, where the $'$ indicates the next value for this signal. For *if* statements - such as in block L1 - the value of a Boolean signal decides which of the two targets is taken. Figure 12.5 graphically details this control flow. There are several small loops, such as the one between L1 and L2, representing the manipulation of an array of values.

The big loop between L3 and L9 represents the actual program execution loop. Everything before that deals with initialization. After initialization the program waits for the next positive clock edge. At the beginning of the execution it is waiting for a valid input value. Then the calculations are executed and it subsequently waits for the next positive clock edge before resuming execution at block L3.

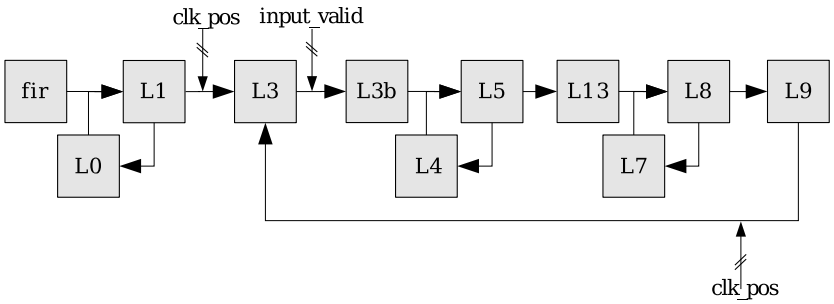


Figure 12.5: Control flow of the FIR filter

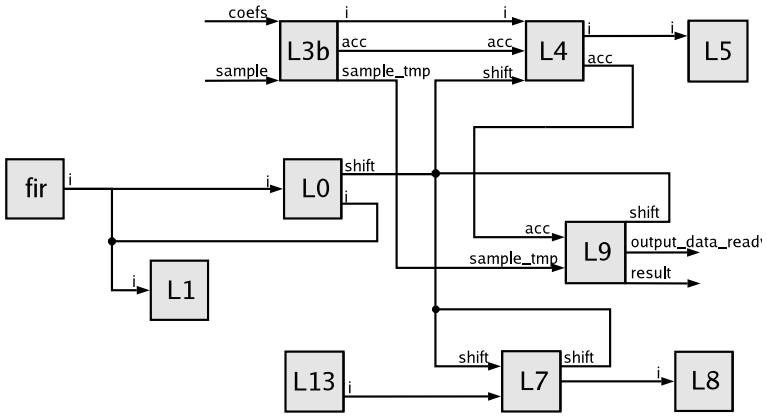


Figure 12.6: Data Flow of the FIR

Data flow dependencies for the FIR are displayed on the right hand side of Figure 12.4. The structure of these dependencies is very simple, the arrow (a→b) showing that a has to be present before b can be evaluated. Overall we see that for the FIR example, the control part largely outweighs the data flow part. Figure 12.6 illustrates the data flow of the example. We see that for the execution loop, the major data activity takes place in blocks L3b, L4, L7, and L9. L3b reads the external inputs *coefs* and *sample*, L9 eventually produces the outputs *out_ready* and *result*.

12.2.4 The Equivalent SIGNAL Program

As described earlier, the combination of control and data flow can be expressed by SIGNAL equations. In order to obtain such equations, it is helpful to have the clock and scheduling information particularized earlier, but they can also be obtained directly from the SSA representation to reflect all control and data flow information. Figure 12.7 details the SIGNAL equation giving the corresponding abstraction of the FIR behavior.

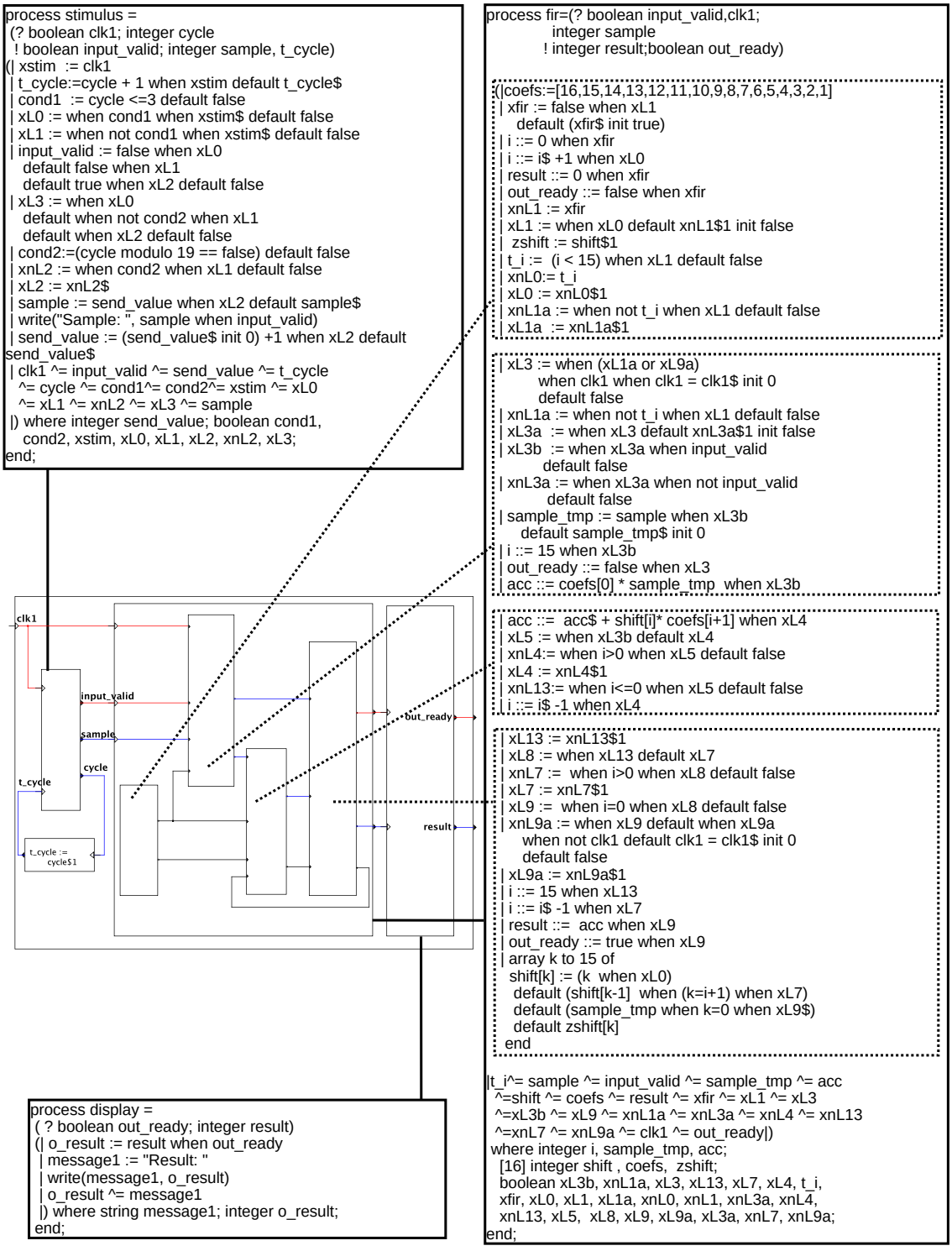


Figure 12.7: Block view of the SIGNAL type for the FIR filter

Translation can be done block-wise, and mostly line by line. The SIGNAL language strictly prohibits multiple assignments of a variable within one block and at the same instant to conform with a purely synchronous execution. Whenever there is the need to advance time before a move from one block to another, the execution of the next block is delayed using a signal delay statement `xL1 := xnL1$1 init false`. Here `xnL1` represents the next value and `xL1` its current value. A variable that gets assigned a value in more than one block would be renamed in SSA.

In SIGNAL it does not have to be renamed, instead we can use partial equations, designated with the "`::=`". A partial equation defines a variable for a certain number of instants. A second or third partial equation then can define additional instants for this variable, as long as it is not defined twice on any instant. Since the instants of the blocks are temporarily disjoint, a variable can be defined once per block with the help of partial equations instead of once per program. Two partial equations for the variable `out_ready` could be distributed anywhere in the program:

```
    out_ready ::= false when xL3
  | out_ready ::= true when xL9
```

Still, partial equations are a source of errors since it is difficult to make sure that they are conflict-free, and to have parts of a variable defined in different parts of the program can also obstruct legibility. This is why we often combine these partial equations afterward into full equations, adding a default statement that otherwise is implied by the compiler:

```
out_ready :=          false when xL3
          default true when xL9
          default false
```

In the code given in Figure 12.7, six partial equations for variable i have been combined. Assembled at one location, it is more clear to see that the definitions do not conflict. In the FIR example, we do not have any complex data manipulations. Would that be the case, it would probably be unreasonably complicated to describe them in SIGNAL. In such cases, they can be wrapped as external functions using SIGNAL's `pragma` directive. With `pragma` statements external code can be used directly. When the type is provided with appropriate signal dependency and clock synchronization relations, these functions are not completely black boxes to the system. This `pragma` mechanism permits the handling of data flow intensive applications without much additional cost.

12.2.5 Making a Boolean Model

As explained Section 12.1.2 the FIR model has to be abstracted to a Boolean model in order to check dynamic properties with the symbolic model checker SIGALI of the Polychrony workbench. Usually, it is not necessary to transform the whole model to a binary form as we might not be interested in all numerical details of the model, but rather some higher level properties such as liveness or deadlock-freedom.

To build such a binary model it has first to be checked whether any float or integer variables are present and how they are used. In the FIR, no float variables are used. However, the actual values of the FIR are integers. They are generated in the stimulus, fed into the FIR and stored in a 15 stage pipeline. The result is calculated numerically and then the output is an integer again.

We reduce the model to binary values in several stages. At first, we reduce the pipeline from fifteen to three stages, representing the stages not by an integer variable, but by three Boolean variables. The input values for the FIR are then reduced to $(0, 1, 2)$ and also represented by Boolean variables. Finally, the most tricky part is the numerical calculation of the result. With the current reduction of values and coefficients, the output of the FIR can never be greater than 15, so we use four Boolean variables as output, interpreted as the bits of a four bit binary number. While the total size of the binary model in number of lines nearly doubled, the state-space for verification only represents a fragment compared to the integer version and is now small enough to be used for formal verification.

12.2.6 Using the Model Checker

Once we have a binary model of the FIR, the model checker can be used to verify formal properties on it. In order to do this, the SIGNAL program has to be compiled with the option `-z3z`, which results in the generation of a file with the extension `.z3z`. This is the input file for SIGALI. It contains a model of the SIGNAL program expressed using polynomial dynamical equations, the data structure manipulated by SIGALI.

As an example on how to define a property for verification, signal *test3*, Figure 12.8 is a Boolean property describing the situation that once the system reaches block *L3* and the signal *input_valid* is true, it will reach block *L9* in at most three steps. We define *test2* as an auxiliary variable that is only true between *L3* and *L9*. The state variable *test3* will be true as soon as test2 has been true for more than three cycles and there was no new value in between.

Figure 12.9 depicts the interaction with the model checker. In the first statement, the z3z file of the design is read. Then internal libraries are loaded. Finally we check for liveness, if property 'test3' is reachable and if it is always false. If

```
test2 :=          true when xL3b
          default false when xL9
          default test2$1 init false
test3 := true when (test2 and test2$1 and
                    test2$2 and test2$3)
              when (input_valid$2 = false)
          default test3 init false
```

Figure 12.8: Example of a formal property definition

test3 is not reachable and always false, then the property holds true.

```
read("top.z3z");
read("Creat_SDP.lib");
read("Verif_Determ.lib");
read("Property.lib");
Alive(S);
Reachable(S,B_True(S,test3));
Always(S,B_False(S,test3));
```

Figure 12.9: Verification of Properties using Sigali

12.2.7 Abstraction of the SIGNAL Model

The SIGNAL program described in Figure 12.7, is the model that implements the FIR filter. It is an exact SIGNAL mirror of the original SystemC implementation. For many purposes, however, all functionality is not needed in order to evaluate the validity of a condition. An abstracted model that does not contain data manipulations is much lighter and can still serve to check conditions such as deadlocks, termination, and race-conditions.

Figure 12.10 depicts the code for a possible abstraction for the FIR model, reduced to the description of control-flow transitions between blocks. The light weight of this model allows for much faster verification of properties, and, therefore makes it possible to check for these properties on more abstract design levels, possibly comprising the whole system. For detailed checks including correctness of data manipulations or range-checks, the complete type can be used.

12.3 Summary

The approach presented shows how to obtain a behavioral type from SystemC components. The passage through the GIMPLE-SSA form allows for a straightforward translation to a formal synchronous model. One important property of this methodology is that it can be applied automatically, such that formal behavioral types can be extracted from a library of existing SystemC components. Depending on the desired accuracy of the behavioral types, the resulting descriptions can be extracted in order to reduce the size of the model for an possible formal verification of design properties. Such formal properties can be checked for with the SIGNAL companion model checker SIGALI. We illustrate the functioning of the methodology with the help of a case study of an FIR filter and work is currently being done in the group to entirely automate this process.

```
process fir (? boolean input_valid, clk1
              ! boolean out_ready)
(| out_ready :=          false when xfir
                default false when xL3
                default true when xL9
                default out_ready$ init false
 | xfir  :=          false when xL3
            default (xfir$ init true)
 | xL3   :=          true when xfir$
                     when clk1
                     when not (clk1 = clk1$)
            default true when xL9a
                     when clk1
                     when not (clk1 = clk1$)
            default false
 | xL3a  :=          true when xL3
            default xnL3a$1 init false
 | xL9   :=          true when xL3a
                     when input_valid
            default false
 | xnL3a :=          true when xL3a
                     when not not input_valid
            default false
 | xnL9a :=          true when xL9
            default true when xL9a
                     when ((not clk1)
                           default (clk1 = clk1$))
            default false
 | xL9a := xnL9a$1
 | input_valid ^= xfir ^= xL3 ^= xL9 ^= clk1
               ^= xnL3a ^= xnL9a ^= out_ready
 |) where integer i; boolean xL3, xnL3a, xL3a,
          xL6, xfir, xL9, xL9a, xnL9a;
end;
```

Figure 12.10: Abstract SIGNAL model

Chapter 13

Automated Extraction of Structural Information from SystemC-based IP

The rising complexity of embedded system design and a widening of the productivity gap have raised the importance of System Level Design (SLD)s languages and frameworks. In recent years, we have seen SLDs such as SpecC and SystemC [GZD+00, OSC] in efforts to raise the level of abstraction in hardware description languages. These SLDs assist designers in modeling, simulation, validation and verification of complex designs. However, the high complexity and heterogeneity of designs make it difficult for embedded system designers to meet the time-to-market. Designers require improved methodologies for verification and validation and tools for debugging and visualization for easier model building to mitigate this productivity crisis. The growing size of common system models is forcing design houses to reuse intellectual property from other designs or from third party companies. The fact that all the different parts of the design have not been conceived in one toolset and by the one actually using it is becoming an additional challenge.

Aside from writing regular testbenches as test-driver modules in SystemC, the SCV library [OSC] is a good medium of writing different types of testbenches allowing features such as randomized testbenches. Unfortunately, the designer must have an understanding of the design, interconnections, datatypes, etc. to generate a testbench using SCV. Furthermore, as the design undergoes changes, the testbench in SCV must be altered. Automating this process of altering the testbench for a design requires access to the structural design information, but most SLD languages and frameworks do not provide a clean mechanism for querying such information. This is why we think it is important that tools are able to automatically extract and exploit structural design information from exist-

ing SLD models in order to further facilitate a realm of design tasks for easier model management, model visualization, automated test generation, improved debugging, etc.

Our approach to extracting structural design information uses a suite of open-source technologies consisting of Doxygen [vHT], Apache's Xerces-C++ XML [Fou], in combination with a C++ library to enable validation tasks exploiting this information. We also do not require any interface description language for entering meta-data. Our approach is based on pre-processing SystemC models through our tools. To show the benefits of having easy access to structural design information, we implement several clients that use it. These clients serve only as examples of exploiting this kind of information and using it for validation purposes. One example is a visualization backend that generates graphical views of the structural information, another is an automated test generator.

Here, we provide details on our approach to the extraction of structural information from existing SystemC designs and describe clients that exploit this information for design validation purposes. We show the benefits and importance of having access to structural design data for the validation of system level designs.

In Section 11.4 we discuss some related work, along with the technologies we employ. We discuss the main contributions of this work in Section 11.3. Section 13.1 then describes how the structural information is extracted, Section 13.2 describes how this information can be used for different validation aspects and we finally give some concluding remarks and future work in Section 13.4.

13.1 Extracting Structural Information

Here, we present details on the infrastructure for the automated extraction of structural information. We only provide small code snippets to present our approach and the concept of using Doxygen, XML, Xerces-C++, and a C++ data structure to perform the extraction and provide the information to use it for validation purposes. For more details on the inner workings of the tool please refer to [BPM+05b].

Doxygen pre-processing: Using Doxygen has the immediate benefit of C/C++ parsing and its corresponding XML representations. However, Doxygen requires declaration of all classes for them to be recognized. Since all SystemC constructs are either, global functions, classes or macros, it is necessary to direct Doxygen to their declarations. For example, when Doxygen executes on just the SystemC model then declarations such as `sc_in` are not tagged, since it has no knowledge of the class `sc_in`. The immediate alternative is to process the entire SystemC source along with the model, but this is very inconvenient when only interested in reflecting characteristics of the SystemC model. However, Doxygen does not

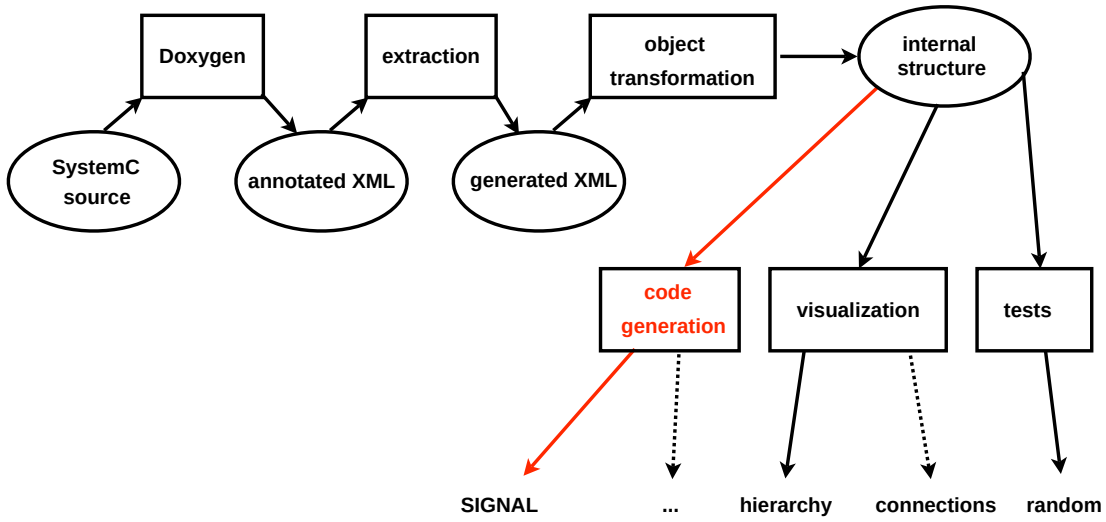


Figure 13.1: Design Flow for the extraction

perform complete C/C++ compilation and grammar check and thus, it can potentially document incorrect C/C++ programs. We leverage this, by indicating which particular classes need to be tagged, by simply adding the class definition in a file that is included during processing. There are only a limited number of classes that are of interest and they can easily be declared such that Doxygen recognizes them. As an example we describe how we force Doxygen to tag the `sc_in`, `sc_out`, `sc_int` and `sc_uint` declarations. We include this description file every time we perform our pre-processing such that Doxygen recognizes the declared ports and datatypes as classes. A segment of the file is shown in Figure 13.2, which shows declaration for input and output ports along with SystemC integer and SystemC unsigned integer datatypes.

```
1  /*! SystemC port classes !*/
   template<class T> class sc_in { };
3  template<class T> class sc_out { };

5  /*! SystemC datatype classes !*/
   template<class T> class sc_int {  };
7  template<class T> class sc_uint { };
```

Figure 13.2: Examples of class declarations

The resulting XML for a small code example is shown in Figure 13.3. Doxygen itself also has some limitations though and it cannot completely tag all the

constructs of SystemC without explicitly altering the source code, which we avoid doing. For example, the `SC_MODULE(arg)` macro defines a class specified by the argument `arg`. Since we do not include all SystemC files in the processing, Doxygen does not recognize this macro when we want it to recognize it as a class declaration for class `arg`. However, Doxygen allows for macro expansions during preprocessing. Hence, we insert a pre-processor macro as: `SC_MODULE(arg)=class arg: public sc_module` that allows Doxygen to recognize `arg` as a class derived from class `sc_module`. We define the pre-processor macro expansions in the Doxygen configuration file where the user indicates which files describe the SystemC model, where the XML output should be saved, what macros need to be run, etc. We provide a configuration file with the pre-processor macros defined such that the user only has to point to the directory with the SystemC model. More information regarding the Doxygen configuration is available at [vHT].

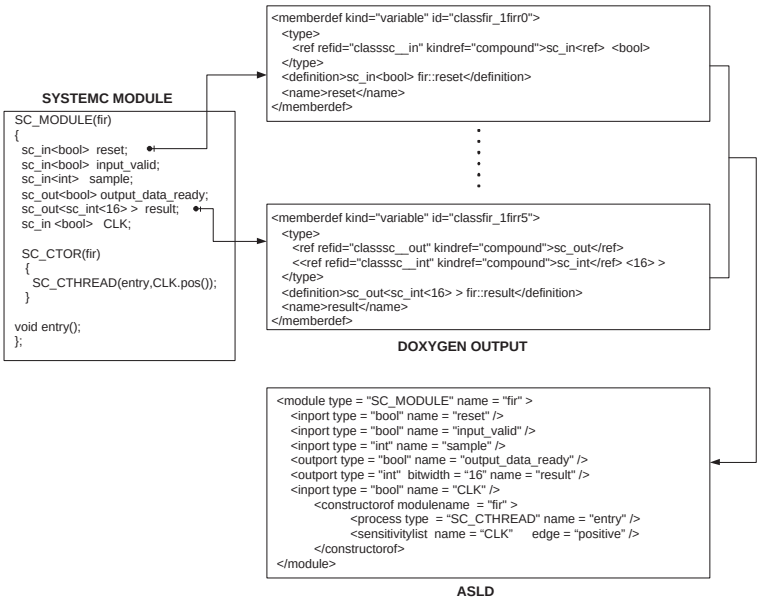


Figure 13.3: Doxygen XML Representation

Even through macro preprocessing and class declarations, some SystemC constructs are not recognized without the original SystemC source code. However, the well-formed XML output allows us to use XML parsers to extract the untagged information. We employ Xerces-C++ XML parsers to parse the Doxygen XML output, but we do not present the source code here as it is simply a programming exercise, and point the readers at [BMPS04] for the source code.

XML Parsers: Using Doxygen and XML parsers we reflect the following structural characteristics of the SystemC model: port names, types and widths, signal

names, types and widths, module names and processes in modules and their entry functions. We reflect the sensitivity list of each module and we also reflect the netlist describing the connections including structural hierarchy of the model. We represent this reflected information in an Abstract System Level Description (ASLD) XML file. The ASLD validates against a Document Type Definition (DTD) which defines the legal building blocks of the ASLD that represents the structural information of a SystemC model. For example, some constraints that the DTD enforces are that two ports of module should have distinct names or all modules within a model should be unique, which verifies that the ASLD correctly represents an executable SystemC model. The main entities of the ASLD are shown in Listing 13.4.

ASLD: In Listing 13.4, the topmost *model* element corresponds to a SystemC model with multiple modules. Each *module* element acts as a container for the following: input ports, output ports, inout ports, signals and submodules. Each *submodule* in a *module* element is the instantiation of a module within another module. This way the ASLD embeds the structural hierarchy in the SystemC model and allows the introspective architecture to infer the toplevel module. The *submodule* is defined similar to a *module* with an additional attribute that is the instance name of the submodule. The *signal* element with its name, type and bitwidth attributes represents a signal in a module. Preserving hierarchy information is very important for correct structural representation. The element *inport* represents an input port for a module with respect to its type, bit width and name. Entities *outport* and *inoutport* represent the output and input-output port of a module. Line 16 describes the *constructorof* element which contain multiple process elements and keeps a *sensitivitylist* element. The *process* element defines the entry function of a module by identifying whether it is an sc_method, sc_thread or sc_cthread. The *sensitivitylist* element registers each signal or port and the edge that a module is sensitive to as a *trigger* element. Connections between submodules can be found in a module or in the *sc_main. Each* connection element holds the name of the local signal, the name of the connected instance and the connected port within that instance. This is similar to how the information is present in the SystemC source code and is sufficient to infer the netlist for the internal data structure.

Using our well-defined ASLD, any SystemC model can be translated into an XML based representation and furthermore models designed in other HDLs such as VHDL or Verilog can be translated to represent synonymous SystemC models by mapping them to the ASLD. This offers the advantage that given a translation scheme from say a Verilog design to the ASLD, we can introspect information about the Verilog model as well.

Data structure: The ASLD file serves as an information base for our reflection capabilities. We create an internal data structure that reads in this information,

```
<!ELEMENT model (module)* >
<!ATTLIST model name CDATA #REQUIRED>
<!ELEMENT module (inport|outport|inoutport|signal|submodule)*>
<!ATTLIST module name CDATA #REQUIRED type CDATA #REQUIRED >
<!ELEMENT submodule EMPTY >
<!ATTLIST submodule type CDATA #REQUIRED name CDATA #REQUIRED
                    instancename CDATA #REQUIRED >
<!ELEMENT signal EMPTY >
<!ATTLIST signal type CDATA #REQUIRED bitwidth CDATA #IMPLIED
                 name CDATA #REQUIRED >
<!ELEMENT inport EMPTY >
<!ATTLIST inport type CDATA #REQUIRED bitwidth CDATA #IMPLIED
                 name CDATA #REQUIRED >
<!ELEMENT constructorof (process * | sensitivitylist) >
<!ATTLIST constructorof modulename CDATA #REQUIRED >
<!ELEMENT process EMPTY >
<!ATTLIST process type CDATA #REQUIRED name CDATA #REQUIRED >
<!ELEMENT sensitivitylist (trigger)* >
<!ELEMENT trigger EMPTY >
<!ATTLIST trigger name CDATA #REQUIRED edge CDATA #REQUIRED>
<!ELEMENT connection EMPTY>
<!ATTLIST connection instance CDATA #REQUIRED
          member CDATA #REQUIRED local_signal CDATA #REQUIRED>
```

Figure 13.4: Main entities of the DTD

enhances it and makes it easily accessible. The class diagram in Figure 13.5 gives an overview of the data structure. The *topmodule* represents the toplevel module from where we can navigate through the whole application. It holds a list of module instances and a list of connections. Each connection has one read port and one or more write ports. The whole data structure is modeled quite close to the actual structure of SystemC source code. All information about ports and signals and connections are in the module structure and only replicated once. Each time a module is instantiated, a *moduleinstance* is created that holds a pointer to its corresponding module.

The information present in the ASLD and the data structure does not contain any behavioral details about the SystemC model at this time, it merely gives a control perspective of the system. It makes any control flow analysis and optimizations on the underlying SystemC very accessible.

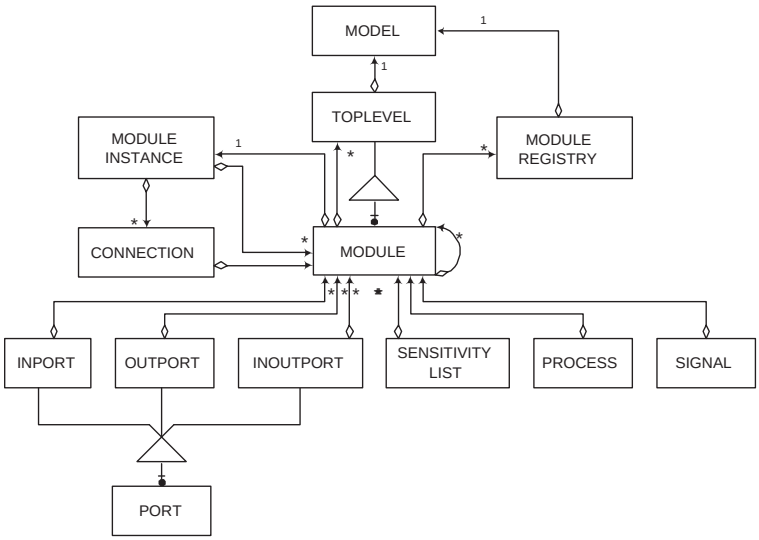


Figure 13.5: Class diagram showing data structure

13.2 Applications for Validation

13.2.1 Visualization

One possible usage of SystemCXML is to facilitate graphical visualization. For large models especially, it is very intuitive to explore a design visually rather than trying to infer the structural aspects of the model by browsing through the code. This problem becomes even more difficult, if the model description is disturbed

over multiple files. Therefore having any form of visualization for the project does ease exploration and debugging capabilities. There are visualization modes at different levels of abstraction that can help to better comprehend a design, such as the netlists displaying module connections on one or multiple levels, the module hierarchy, a layout with blocks whose sizes are mapped to the code size of the corresponding modules. The ability to provide the above visualization modes can be easily achieved through an extraction of the structural information of the model, therefore it is not necessary to understand the behavioral aspects. As visualization can greatly improve productivity, it should be an integral part of any SLDL tool suite. Design visualization tools are especially helpful for design space exploration and semi-automated design refinements. In addition to the above advantages, the automatic generation of a graphical visualization of the design can also be used for documenting different IPs, a step often neglected, leading to better collaboration in terms of IP exchange between different vendors and enhancing reusability of components.

In order to demonstrate the ease of creating such a visualization, we implement a back-end pass that generates a graph of the SystemC module hierarchy. Since, there are many free libraries available for graph rendering, we decided to use the DOT format [GKNV93] from the graphviz [GN00] package to render our graphs. It is a comprehensive and easy to use package, which is used in many Open Source projects.

13.2.1.1 The DOT format.

Figure 13.6 shows the DOT code for the FIR filter example and the resulting graph. We use a *digraph* layout and choose boxed nodes whose width automatically adjusts to the length of the node label. The first occurrence of a node name creates the node. Directed connections are indicated with the "`->`" symbol.

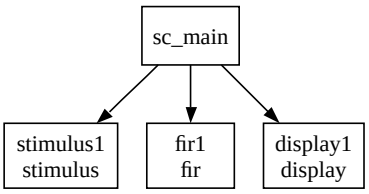
There exist many programs to interactively view DOT files or convert them into various picture formats. Dotty is the standard viewer and part of the Graphviz, but there are better viewers such as [Pie05].

13.2.1.2 Graph generation.

To generate the graph, we start with the list of toplevel modules, these are modules that are not a submodule of any other module. Then we call the recursive function *submod_dot* that writes out the relations to all submodules and successively calls itself for all the submodules. As a node label we give the module name and the name of the instance. In order to keep a strict tree structure with no rejoining branches, all instances have to have different names. However in the SystemC code this is not necessarily the case. For example, we have three modules A, B and C, with $B1$ being an instance of B and a submodule of A.

```
digraph usb
{
  node[shape=box];
  ratio=fill;
  sc_main;
  sc_main->"i_phy\nusb_phy";
  sc_main->"i_top\nusb_top";
  sc_main->"i_test\ntest";
}
```

(a) DOT code



(b) Resulting graph

Figure 13.6: Example DOT code and resulting graph

Now if $A1$ and $A2$ are submodules of C, we get 2 instances of B that have the name $B1$, namely in $A1$ and $A2$. In order to avoid this we keep track of multiple instantiations of a module and distinguish between the respective submodules.

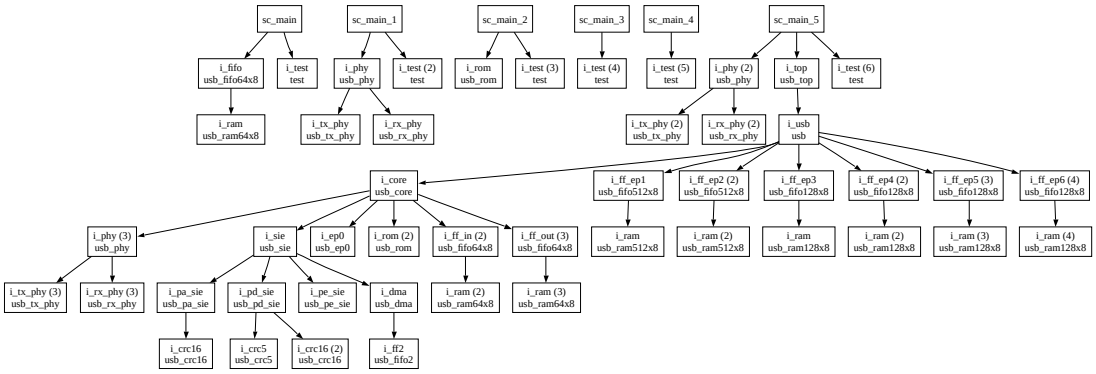


Figure 13.7: Visualization of the extracted module hierarchy of a USB controller

Figure 13.7 shows a part of the module hierarchy of the SystemC implementation of a USB controller from OpenCores [Cor]. In the lower right hand corner you can see four instances of $usb\_fifo128x8$, containing an instance of $usb\_ram128x8$. These have been numbered in order to be able to distinguish them. The figure also shows that there is not only one connected graph but multiple graphs. This is due to the fact that we read in the whole SystemC project as one file containing all source and header files. Larger projects often contain multiple *sc_main* functions, used to individually simulate parts of the design in a separate testbench, which is the case in this example. The visualization of the hierarchy helps to see the different parts of the design and understand their usage.

The code for the back-end pass to generate the graphical visualization illustrating the module hierarchy, took around 60 lines of C++ code. This is small when considering the value add. We assert that given the captured structural information, multiple back-end passes for other visualization modes or transformations can be added with comparable effort, reducing the effort needed in trying different things or implementing a desired functionality. An enhancement to the current version of the our visualization mode would allow the module connections where the user can choose the number of displayed hierarchy levels. We were looking into this option as well, but Graphviz does not natively support this kind of nested hierarchy, so it may be necessary to use of a different graph rendering library.

13.2.2 Design Management

Another important possible use of structural design information is design management. Large designs are getting difficult to maintain - even if they are kept in a file and folder structure that is readily accessible. Browsing a design graphically or through tree and list views can help in managing and maintaining large designs. This makes it easy to view the different components of a design and furthermore it helps in identifying the component of concern, which reduces the effort needed in isolating a particular component of interest to the user.

Suppose the user is interested in knowing the amount of RAM attached to the overall design. This is not obvious to infer from the code since a single RAM module can be instantiated with different sizes at multiple places within the same module as well as across modules. Furthermore, there may be possible RAM instantiations in the design that were done for testing purposes, which is not a part of the actual design. For these cases, the designer can use the module hierarchy visualization, to view all the instances down to the leaf level, from which it is easier to filter out all the RAM instantiations and sum them up to get an idea of how much memory is been used and how much space is needed on the chip. Figure 13.7 illustrates the module structure of a USB controller, it contains two 64k blocks, four 128k blocks, and two 512k blocks, which sums up to 1664 kilobytes RAM. This calculation does not include the 64 kilobytes instantiated in the test of usb_fifo. This valuable information can be easily achieved using our module hierarchy view, which is otherwise very tedious.

Another design management task can be to identify certain functionalities in old designs and package them into a new IP for future use. In a netlist view the graphical selection of a set of modules can then be put into a new module with an automatically generated interface, containing all these modules and their dependent submodules. Again this kind of operation only necessitates knowing the structural information, but otherwise tedious to perform.

13.2.3 Automated Test Generation

We develop a testbench Generator client that uses the structural information to support automated test generation. The test generation client is built using the SystemC verification (SCV) [OSC], which is a library of C++ classes, that provide tightly integrated verification capabilities within SystemC. The testbench Generator interacts with the ASLD by invoking the respective API calls to access the structural information pertaining to test generation. The generator takes as input a SystemC model and generated automatically a SystemC test for the selected part of the model. This testbench Generator uses structural information such as the type, bitwidth of ports and signals to generate test vectors appropriate for this specification of the SystemC model. The generator also has the abilities to generate tests for pre-specified ports or signals of a SystemC model. The client generates different tests based on the mode in which it is set. The different mode can be set during initialization of the client. The *unconstRand*, *simpleRand* and *distRand* are the currently defined modes.

The test generator can create constrained and unconstrained randomized testbenches. In the *unconstRand* mode, the client generates unconstrained randomized tests using objects of the *scv_smart_ptr<T>* class of SCV, which are containers for objects of type T. In the *simpleRand* mode constrained randomized testbenches are created. These tests issue *Keep_out* and *Keep_only* commands to define the legal range of values given in the data file. Similarly in the *distRand* mode, SCV bag objects are used in test environments providing which takes a data file as input with the values and their probability.

Furthermore, the Test Generator uses the SystemC constructs to generate trace file in the format of Value Change Dump (VCD). This provides the user with a trace file with value changes on all the reflected variables, ports and signals of the model, which can used for debugging purposes. To generate a trace file, the Test Generator creates a trace file, registers the reflected variables, ports and signals to be traced and closes the trace file.

Test generation example. We briefly describe the automatic test generation using a small part of the USB example in Figure 13.7. We take a look in particular at the module *usb_crc5*, a simple CRC checksum checker. Figure 13.8 shows the interface of the CRC checker module. It has to input variables, *din* for the data value, and *crc_in*, for the CRC value. For this very simple example we show how to automatically generate test vectors for these inputs.

In the *simpleRand* mode, constrained testbenches are created by initializing SCV smart pointers for the pre-specified port as shown above. Furthermore the random values generated are constrained by defining *keep_out* and *keep_only* constructs with the legal ranges given from the input data file as shown in Fig-

```
SC_MODULE(usb_crc5) {
  public:
  sc_in<sc_uint<5> >  crc_in;
  sc_in<sc_uint<11> > din;
  sc_out<sc_uint<5> > crc_out;
  void update(void);
  SC_CTOR(usb_crc5) {
    SC_METHOD(update);
    sensitive << crc_in << din;
  }
};
```

Figure 13.8: Interface of the module usb_crc5 from the USB example

```
/*! Defining SCV smart pointers !*/
 scv_smart_ptr <int> r_crc_in;
 scv_smart_ptr <int> r_din;

/*! Generating the randomized values !*/
 r_crc_in->next();
 r_din->next();
```

Figure 13.9: Testbench snippet in *unconstRand* mode for a pre-specified port

ure 13.10. If no data file is provided then by default random legal ranges are defined.

```
/*! Defining simple constraints !*/
 scv_smart_ptr <int> r_crc_in;
 scv_smart_ptr <int> r_din;

/*! Defining the legal ranges !*/
r_crc_in->keep_only(10,1000);
r_crc_in->keep_out(100, 300);
r_crc_in->keep_out(600, 900);
r_din->keep_only(1,10000000);
r_din->keep_out(1000, 30000000);
r_din->keep_out(3001000, 9000000);
r_din->keep_out(9001000, 10000000);
```

Figure 13.10: Testbench snippet in *simpleRand* mode for a pre-specified port

In the *distRand* mode, constrained testbenches are generated by defining *SCV_bags* that are given legal ranges and the probabilistic distribution of these ranges from an input data file as shown in Figure 13.11. As in the *simpleRand* mode, if an input data file is not given then a default distribution and its probability is provided.

```
/*! Defining weights for the distribution mode !*/
 scv_smart_ptr <int> r_crc_in;
 scv_bag<pair<int,int> > d_crc_in;

/*! Defining the legal ranges !*/
 d_crc_in.add(pair<int, int> (1, 100), 40 );
 d_crc_in.add(pair<int, int> (5000100, 500700), 30 );
 d_crc_in.add(pair<int, int> (8000600, 800900), 60 );

/*! Setting the distribution mode !*/
 r_crc_in->set_mode(d_crc_in);
```

Figure 13.11: Testbench snippet in *distRand* mode for a pre-specified port

During initialization, if the ports are not specified then the test generating client generates tests with respect to all the ports of the given model in focus.

The code snippet that creates a trace file for the usb_crc5 module with the reflected ports crc_in, din and crc_out is shown in Figure 13.12:

We intend to improve our automated testbench generation capabilities by first implementing additional clients such as coverage monitors and simulation

```
1 /*! Step 1: Creating a trace file !*/
      sc_trace_file* tf = sc_create_vcd_trace_File("trace");
3
  /*! module!*/
5     usb_crc5 crc_inst("crc_inst");

7 /*! Step 2: Register signals and variables to be traced !*/
      sc_trace(tf, crc_inst.crc_in, "crc_input");
9     sc_trace(tf, crc_inst.din, "din");
      sc_trace(tf, crc_inst.crc_out, "crc_output");
11
  /*! Step 3: Close the trace file !*/
13    sc_close_vcd_trace_file(tf);
```

Figure 13.12: Snippet of the testbench showing the trace file creation

performance monitors to better analyze the SystemC model. These additional clients assist the testbench Generator in making more intelligent and concentrated testbenches.

13.3 Current Work

As of now, we can extract the structural information from SystemC projects and generate SIGNAL process skeletons. These structural skeletons can be filled with the behavioral information retrieved from the behavioral C++ to SIGNAL transformation that also has been defined. We are currently working on entirely automating the transformation from SystemC behavior to SIGNAL and to integrate it with the structure. The major part of the work still to be done is to implement the adequate SIGNAL equivalents for certain SystemC constructs such as *sc_fifo* or *sc_semaphore* have to be defined. These can then form a library that would significantly simplify the conversion process.

13.4 Summary

Large scale system designs and increasing component reuse from diverse sources makes design validation a nightmare. Simulation and testing are important validation tools, but they are not sufficient for an increasing number of applications. Formal models and formal verification are one possible answer to this problem, however building correct formal models is difficult and means a lot of additional effort for an already existing design.

In this chapter we show how to automatically transform a SystemC description into the formal language SIGNAL. We describe how SystemC behavior can be translated into SIGNAL using the GCC intermediate SSA representation. In order not to lose the structure of the original model, it has to be obtained separately. In order to do this, we present a methodology for the automated extraction of structural information from already existing SystemC projects. In addition to using structural data for the generation of SIGNAL process skeletons, we illustrate how the data can be easily exploited for other applications such as visualization, design management tasks, and automated test generation.

All parts of the system have been implemented using open source tools such as GCC, Doxygen, and Xerces-C. In the conception and implementation we have been attentive to use open standards and interfaces such as the intermediate SSA representation and XML, and the source code of the SystemC structural extraction tool is published as an open source project at Sourceforge.net [BMPS04] for others to study and use.

When combining the structural and the behavioral transformation results we obtain a formal model that is functionally equivalent to the original SystemC model. This formal model can help to find errors in the original design, e.g. in the connection with other components. If a synchronous composition of several SIGNAL processes is successful, the connection of the corresponding SystemC components is very likely to work. Additional confidence can be gained by verifying formal properties of the components as well as of any composition of components thus increasing certainty on the correctness of the whole system design. To speed up verification or to concentrate on specific functionalities, the formal model can be abstracted to the desired level. If this methodology is applied systematically, a growing library of verified IP components is obtained that helps to substantially reduce development cycles and eventually makes it possible to develop safer and larger systems.

Chapter 14

Incrementally Building Formal Models

14.1 Introduction

In our experience, both in carrying out formal verification for major microprocessor chips, as well as, working on tools and methodologies, we have found that the major challenges faced by industrial formal verification engineers are two fold: (i) Making sure that the natural language specification of the system is translated into a sufficiently complete set of formal properties to be used in model checking of an implementation, (ii) In conformance based formal verification using abstraction techniques, creating an abstract model which satisfies all formal properties intended in the natural language specification. Most of the times, it is hard to validate the sufficiency/completeness of the property suite developed from the natural language, or to make sure that the abstract model is constructed correctly. By "correctly" we mean that the set of behaviors of the abstract model is not only a super set of the set of behaviors of an implementation, but also a subset (in the best case, equal) to the set of behaviors intended/allowed by the natural language specification. Our XFM based methodology addresses these problems, and with two illustrative examples (of a control intensive traffic light controller, and the DLX pipeline) we present this methodology and show the benefits. Our experiments show that this methodology not only constructs abstract models with sufficiently shorter time than the time taken in constructing ad hoc abstract models from implementation or specification, but also provides models which are constructively correct and closer to the intended specification.

Extreme programming (XP) has been popularized in the object oriented software community in the recent years. It introduced novel guidelines and concepts of an agile methodology that seem to increase programming productivity significantly while producing higher quality error free code [WK03] [Wil03]. Some of

the features of extreme programming are use of 'user stories', 'test-first' development, 'refactoring', 'continuous regression' etc., which we have found very useful in creating more dependable software if applied properly. In the spirit of our successful use of extreme programming in software development, we decided to experiment with a parallel methodology in formal model construction.

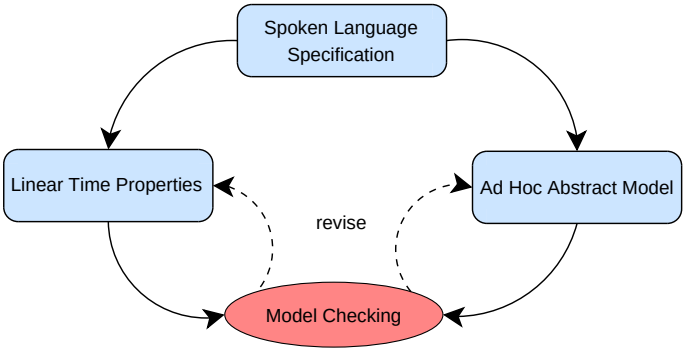


Figure 14.1: State of the art to capture a formal model

Figure 14.1 and 14.2 show our comparison of the current state of the art in capturing formal specifications against our XFM approach. Figure 14.1 shows that an ad hoc abstract model is usually built from an English specification and checked against formal properties with a model-checker. Some times, to make matters worse, the ad hoc abstraction is built from an implementation itself, which is then checked against the implementation for conformance. In our view, that defeats the purpose of formal verification, except that the abstraction step might uncover bugs unknown to the implementors. There are several drawbacks in this approach. First of all, the ad hoc building of both the model and the properties is error prone and the effort of model building and debugging grows exponentially along with the size of the model. Next, as there is no way to control the inclusion of all properties, some may be overlooked, thus reducing the significance of the model. Then, if a property fails, it is tedious to debug the model. Few indications exist where the bug is located. Finally, there is a tendency that the model will include more behavior than the specification will allow. Often implementation detail gets into the abstract model. These tendencies might make the model have undesirable properties and hence the implementation being checked against it may have those too. Also implementation detail in the abstract model may introduce unwanted complexity and may later cause problems in a conformance check.

Figure 14.2 presents XFM's incremental approach to formal modeling. From the English specification, we first derive a simple formal property, then build an abstract model for this property and model check if it holds for the model. Once

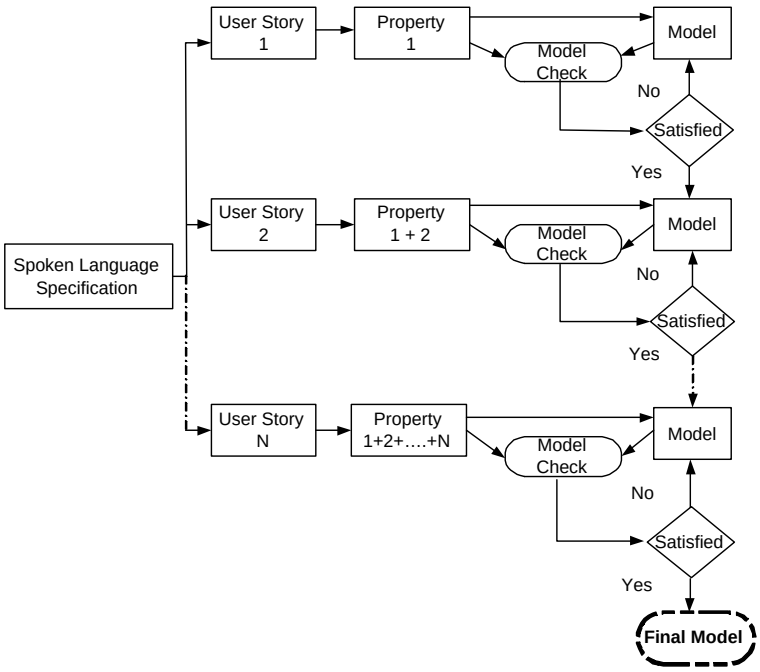


Figure 14.2: Capturing a formal model with XFM

the property is satisfied, we take a second property, extend the model according to this property, and model check for both properties. This procedure is repeated until the abstract model contains all behavior from the English spec (Figure 14.2). One way to make sure that it does is by simulating the model. The controlled and incremental model building results in a compact, structured abstract model. Whenever a property fails to validate, it usually is straightforward to find the bug as it must be related to the latest additions. The complete effort of modeling and bug fixing grows linearly along with the size of the model.

14.2 Contributions

The work presented in this chapter has been conducted in the framework of the INRIA associated teams program with the FERMAT (Formal Engineering Research with Models, Abstractions, and Transformations) laboratory of Virginia Polytechnic Institute and State University. It tempts to lower the difficulty to build formal models from spoken language requirements specification and to create formal models that are correct by construction. One of the main contributions is to use agile methods that are most prominently known from Extreme

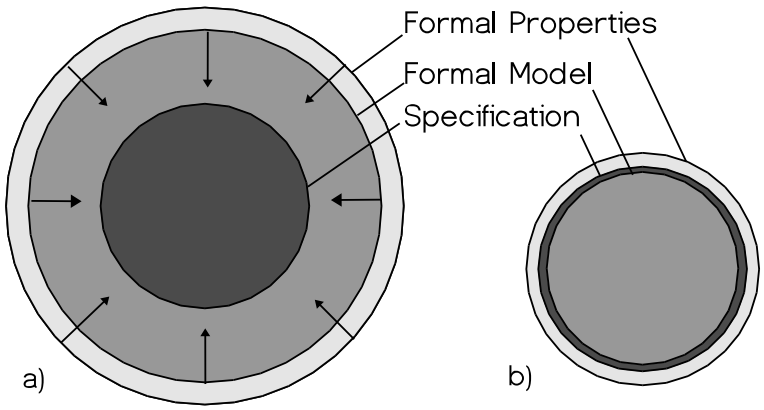


Figure 14.3: Modeling process (a) and modeling result (b)

Programming (XP), to investigate their potential in the field of formal verification, to conceive a methodology that exploits the benefits of using agile methods for the building of formal methods and to validate the methodology with the help of some example projects.

This initial work has been done by Syed Suhaib, Sandeep Shukla, and myself and has been initially published as a technical report [BSSH05].

Later, the work has been conducted further mainly by Syed Suhaib who made this the topic of his masters thesis [Suh04] and Deepak Matthaikutty resulting in more extensive examples, a more rigorous formalization, and an extension to make the results more predictable. This part of the work is presented briefly in Section 14.6.

14.3 Incremental Model Building and Polychrony

The methodology presented in this chapter does not work for all types of programs and specifications. Having said that, it is not restricted to the PROMELA language and SPIN which we use as an example. The formal framework Polychrony is predisposed for the incremental building of formal models. In this section we exhibit certain formal requirements and advantages that a modeling framework has to feature in order to be able to be used for this kind of incrementally model building. Due to a lack of time and resources it has currently not been implemented in Polychrony, however, we show how the approach relates to Polychrony and in what would be the particularities of a possible implementation. In order to successfully build incremental formal models, certain conditions have to be

respected and others, even if they are not necessary, improve the efficiency of the process.

14.3.1 Nondeterminism

Nondeterminism is the property of a computation that may have more than one possible outcome. Most programming languages are deterministic in nature, given a certain input, they generate always the exact same output. Most systems in reality however are nondeterministic. There may be communication delays or hardware effects causing a system to reach states that would not be accessed otherwise. Also programs that contain states depending on user choices or other input that is not known a priori exhibit nondeterministic behavior. For the formal modeling of these systems the formalism used has to be able to capture the systems' inherent nondeterminism in order to be able to explore the complete state space when checking for the properties. If the code is to be executed the nondeterminism has to be resolved such that a deterministic program can be generated. This can be done by making defined or random choices for the values in question, restricting all possible behaviors of the system until a single behavior is left. Both Promela and Polychrony support the description of nondeterministic programs. While in Promela during simulation this is resolved with the help of random choices, in Polychrony, defined choices have to be made before the generation of executable code.

14.3.2 Verification Logic

A verification logic supports formal reasoning about properties of programs in a specific programming language by embedding rules coherent with the semantics of the language. A verification logic has to be expressive, such that the majority of logic properties can be expressed. However it also has to be simple and intuitive such that it is easy to correctly express a desired property. These two requirements are often in conflict. A very expressive verification logic has the tendency to be difficult to use, and a good intuitive logic can restrict the expressiveness. Common temporal verification logics comprise Linear Time Logic (LTL), computation tree logic (CTL), extended temporal logic (ETL), and propositional temporal logic (PTL). For the differences in expressiveness and more or less intuitive expression, the verification logic has to be chosen carefully. SPIN is using LTL as a verification logic, but it supports also the formulation of temporal properties in the form of FSMs.

14.3.3 Compositionality

Compositionality for a specification means, that different parts of the specification can be put together into a specification describing exactly the behavior of the sum of the two. In SPIN compositionality is not very easily accomplished as the integration of two partial specifications represents some effort. There are languages however, that support the composition of two partial specifications with no additional effort. SIGNAL programs for example represent a system of equations that are all evaluated at the same time. As there is no order for the equations, two SIGNAL programs can be added to each other by just concatenating the two. Such a level of compositionality is favoring the incremental specification, as the adding of functionality does not always require to adapt the already existing code.

Another aspect of compositionality is *partial specification*. Partial specification means that different aspects of the system can be described separately. The main idea is that a specification consists of a collection of interlocking partial specifications, each describing the system from a different viewpoint. This is important for large designs where diverse aspects of a system are not always specified within the same team. As SIGNAL descriptions support an easy composition, Polychrony does natively support the expression of partial specifications. It has to be assured however that the partial specifications are consistent between them [BDBS99].

14.4 Methodology

As for any system development, it is important to have a concise and clearly written specification of the system. Some time must be spent on the specs to get an overview of the whole system and maybe visualize its main structure. Both, a clear system specification and a deep understanding of the system are crucial for good LTL properties.

The initial part of our XFM procedure involves breaking down the English specification to user stories. We select a user story that describes basic functionality of the system, and transforms it into an LTL property. The next step is to check if the LTL property correctly expresses the behavior of the user story. LTL 2 BA eases this step by displaying the corresponding FSM. If the property is sound, we start building the model corresponding to this property. It is important that while implementing the model only the behavior of this property is taken into account. If the model checker fails to validate the property, we can locate the error with the help of the trace file generated by the model checker, fix the bug and rerun verification. Once verification is successful we pick the next basic user story, transform it into an LTL property and extend the model obtained from the

previous property to satisfy this one as well. This procedure is repeated until it contains everything that is specified in the English spec. The final model can be simulated to ensure that all specified functionalities are incorporated. If a certain functionality is found missing, we identify the corresponding LTL property and extend the model accordingly. After correct simulation, the model and the list of LTL properties should be complete.

14.4.1 Tools

As our model checking environment we chose the SPIN model checker because it is one of the most popular model checkers in todays modeling of concurrent systems. However, we could use any model checker for the same compelling results. For SPIN, the models are specified in PROcess MEta LAnguage (PROMELA), a system description language. Its basic building blocks are asynchronous processes, message channels, synchronizing statements, and structured data. Once the model is built, the user can simulate it with the built-in simulator and verify formal properties. Verification properties can be entered in LTL or in the form of PROMELA never claims, for properties that are not expressible in LTL.

LTL is the leading technique for specification of temporal rules. It extends propositional logic with the four operators "always" (condition holds always in the future), "eventually" (condition holds sometime in the future), "next" (condition holds in the next cycle), and "until" (condition A holds until condition B, afterwards do not care). All LTL operators are listed in Figure 14.4. As to the expressiveness of LTL, it is complete with respect to first order logic [Hen00]. Temporal expressions that cannot be expressed in LTL, can be provided to SPIN in the form of a never-claim automaton.
Small changes in a LTL property, like the misplacement of a parenthesis, can change the meaning completely. For example [](a→<> (b U c)) represents a simple 2-state automaton, if we move one parenthesis before the <>, it is a 7 state automaton. Even if these kinds of mistakes are hard to detect, it is especially important that properties are correctly defined before checking the model for it. There is a tool called LTL 2 BA [Odd01] (LTL to Büchi automata) that generates a Büchi automaton representing any LTL expression. This visualization is instrumental in verifying that the expression matches the specification. LTL 2 BA also generates PROMELA code from an LTL expression. Therefore it could - in theory - be used to obtain the abstract model directly and automatically from the LTL properties. However, in practice this does not work since it is neither possible to describe concurrent processes in LTL nor to describe implementation details such as initial states or changing state variables.

!A	negation
A→B	implication
A ↔ B	equivalence
A && B	and
A || B	or
[]A	Always
A U B	until
<> A	eventually
X A	next

Figure 14.4: LTL operators

14.4.2 Extreme Programming

As stated earlier, XFM works on the lines of XP. Many of the XP rules can be applied directly and successfully in XFM. For instance one of the main XP rules is to write tests before the actual code. In XFM this rule maps to specifying the LTL property before writing the abstract model. Another important XP technique is to add functionality as late as possible, keeping the model simple for as long as possible. Iterations are small steps in the development process. At the start of each iteration the goals are identified and written down in the form of "user stories" - individual cards that point out specific implementation details and requirements. These user stories act as a detailed guideline for the programmer. To refactor problems as much as possible, to update tests after a bug is found, and to work in pairs are also principles that are as beneficial to the capturing of formal methods as they are for common programming projects. The benefit of other XP techniques such as a stand up meeting in the mornings, collective code ownership and moving people around depends on the type of the project, the size of the team, and on personal and corporate preferences.

14.5 Example

In order to demonstrate the power of XFM we present two examples from different domains and of different complexity. A simple traffic light will illustrate the main steps, tools, and techniques involved. The design of a DLX [HPG02] microprocessor pipeline will show how this works for a bigger model, and how the model evolves with the incremental approach.

never both have a green signal	$[]!(!c\&\&!p)$
If cars cannot go, they will go eventually	$[](c \rightarrow <>!c)$
No button pressed, cars keep going	$[]((!c\&\&!sw) \rightarrow X!c)$
No button pressed the pedestrians cannot walk	$[]((p\&\&!sw) \rightarrow Xp)$
When the switch is pressed while the cars go, pedestrians will go before the switch is turned off.	$[](sw\&\&p \rightarrow swU(!p\&\&sw)\&\&(!p\&\&sw \rightarrow !pU(!p\&\&!sw)))$

Figure 14.5: LTL properties for traffic light (c = cars stop, p = ped stop, sw = button is pressed)

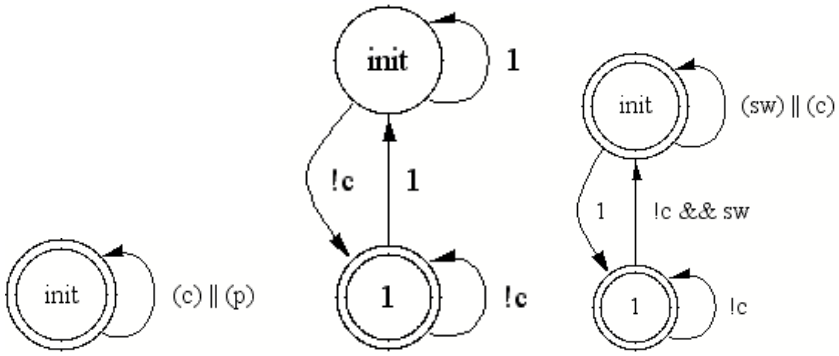


Figure 14.6: FSMs of traffic properties 1 (a), 2 (b), 3 (b)

14.5.1 Traffic light model.

This is a very simple example of a pedestrian crossing with a traffic light. When a pedestrian pushes a button, the lights turn red, and the pedestrians can walk. After one minute the pedestrians get a red light and the cars red light goes off. So this description is the English specification. Now, we construct LTL properties describing this system. We start with the most important property that states that both pedestrian and car, can never get the GO signal at the same time: []!(!p && !c). We verify that the property concurs with the specification with LTL 2 BA. Figure 14.6(a) shows the automaton corresponding to this property. The corresponding model is just as simple, just one state.
The next property we come up with states that whenever the cars stop, they will eventually go (Figure 14.6(b)). Figure 14.5 lists all LTL properties for this example. LTL does not allow to express exact timing, only relative occurrences of events. But in the model we add a timer that counts to 60 before the pedestrians stop and the cars can run again. The model now includes two states, one where

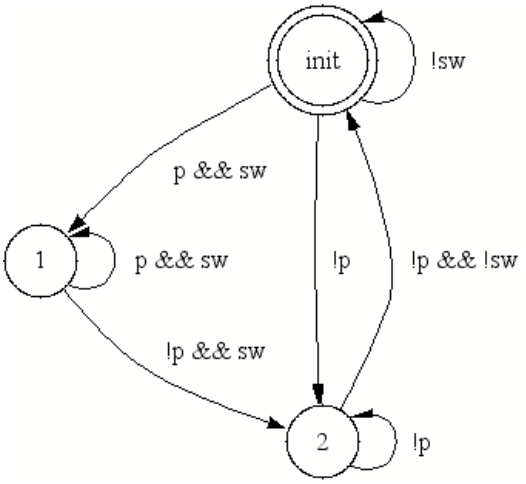


Figure 14.7: Graph for traffic property 5

cars go (!c) and pedestrians stop (p) and the other where pedestrians go and cars stop.
The following two properties state that when no switch is pressed, the cars keep driving and the pedestrians keep stopping (Figure 14.6(c)). As we check these properties against the formal model, we realize that they can be verified without making any modifications to the system, and a closer look at the properties shows that their behavior is already satisfied by property 1 (Figure 14.6(a)).
One functionality that is still missing is the inclusion of the switch. When cars go and the switch is pressed, eventually pedestrians should be allowed to walk before the switch turns off. This property is a bit longer than the others, and without LTL 2 BA it is not easy to figure out if it is correct (Figure 14.7). After implementing the functionality of these properties into the model, simulation shows that it works as specified, so we have found all properties. Figure 14.8 shows the PROMELA code for the traffic light example.

14.5.2 Model of a DLX Pipeline Control

The actual power of the XFM approach develops when working on large systems. The pipeline control of the DLX RISC processor model [HPG02] is a well known and reasonably large example to show the use of XFM. The DLX has a 5-stage pipeline, which means up to five instructions can run concurrently. The cycles for the instructions are instruction fetch (IF), instruction decode (ID), execute (EX), memory access (MEM), and write back (WB). However, not all instruction types use the same cycles in the same order. Figure 14.9 shows the cycle usage for the different instruction types.

```
bool sw, c, p;
int time;
active [0] proctype signal()
{
cargo:
  p=1; c=0;
  if
    :: (1) -> sw =1; time=30; goto pedgo
    :: (1) -> goto cargo
  fi;
pedgo:
  c=1;  p=0; sw =0;
  time = time -1;
  if
    :: (time > 0) -> goto pedgo
    :: (time == 0) -> goto cargo
    fi
}
init {
  p = 1; c = 0; sw = 0;
  run signal();
}
```

Figure 14.8: Promela code for pedestrian crossing

Starting from this system description, we identify the first user story. One of the most basic behaviors states that each instruction executes in a certain order. So, generally speaking, instructions execute in the order $IF \rightarrow ID \rightarrow EX \rightarrow MEM \rightarrow WB$. In LTL this can be expressed as $[](if \rightarrow Xid)$, always ID after IF and then the same for ID and EX, EX and MEM, MEM and WB, and finally WB and IF. The automaton generated with LTL 2 BA (Figure 14.11(a)) shows that the LTL expression is sound. These five properties can be represented with a circular automaton that satisfies our first user story.

The second user story is the fact that this order of execution still has to hold when we consider five concurrent instructions in the pipeline. In order to keep

	IF	ID	EX	MEM	WB
Arithmetic	X	X	X		X
Load	X	X	X	X	X
Store	X	X	X	X	
Branch	X	X	X	X	

Figure 14.9: Cycles for different instruction types

the model small we decide to use five concurrent processes each of which handles one instruction (Figure 14.14). Since the processes run independently, the first property does not hold any more. It is not guaranteed that directly after the first instruction is in the fetch stage it advances to the decode stage, since in the meantime other processes may get execution time. What we can guarantee however, is that we will never go directly into any of the other stages. Now this has to be expressed for each cycle in each instruction, which means we get 25 LTL properties like cat1 in Figure 14.12.

```
proctype instruction1() {
    inst_if:
      if
      :: st1=fet; goto inst_id fi;
    inst_id:
      if
      :: st1=dec; goto inst_ex fi;
    inst_ex:
      if
      :: st1=ex;  goto inst_mem fi;
    inst_mem:
      if
      :: st1=mem; goto inst_wb  fi;
    inst_wb:
      if
      :: st1=wb;  goto inst_if  fi;  }
```

Figure 14.10: PROMELA code for one single instruction

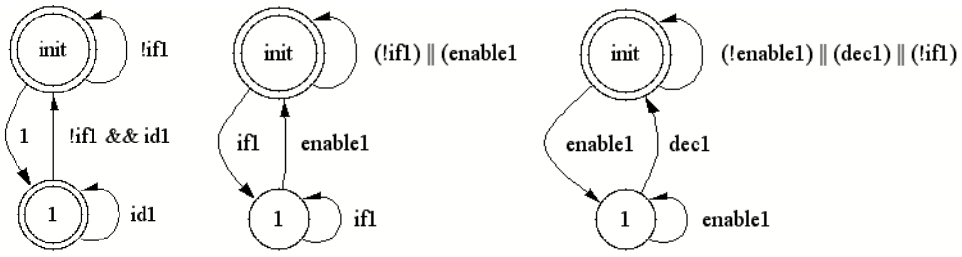


Figure 14.11: Graphs of pipeline properties 1 (a), 3 (b), and 4 (c)

In the next iteration we introduce the possibility to control the instructions from outside. This is done by "enable signals", one for each instruction. The LTL expression will say that an instruction will not advance unless the enable signal is given (Figure 14.11(b)). Again we obtain 25 properties in the style of

cat1	$[](if1 \rightarrow !(Xex1 \| \| Xmem1 \| \| Xwb1))$
cat1b	$[]((ex1 \&\& (load1 \| \| store1 \| \| branch1)) \rightarrow !(Xif1 \| \| Xwb1 \| \| Xdec1 \| \| Xwait1))$
cat2	$[]((if1 \&\& !enable1) \rightarrow (if1 U enable1))$
cat2b	$[]((wait1 \&\& !enable1) \rightarrow (wait1 U enable1))$
cat3	$[](if1 \rightarrow ((enable1 U dec1) \| \| !enable1))$
cat3b	$[]((ex1 \&\& (load1 \| \| store1 \| \| branch1)) \rightarrow ((enable1 U mem1) \| \| !enable1))$
cat4	$[]((if1 \&\& enable1) \rightarrow ((!(if2 \&\& enable2) \| \| (!(if3 \&\& enable3)) \| \| (!(if4 \&\& enable4)) \| \| (!(if5 \&\& enable5)))U !enable1))$

Figure 14.12: Example LTL properties for the DLX pipeline

cat2 in Figure 14.12. The changes in the model for these properties are small, so all of them can be verified without problems.

The following iteration is adding some synchronization. Our user story says that the control enables each instruction in each cycle. Once the instruction advances, it is setting its enable signal to zero, thus signaling the control that it is ready for the next cycle. This category of properties is somewhat more complex, but with the help of the LTL 2 BA tool we finally find cat3 in Figure 14.12. It reads that whenever a stage is true, it will change to the next stage before the enable sign goes down, unless the enable sign is already low (Figure 14.11(c)). Again we get one of these properties for each stage that is 25. Once the correct properties are found, the changes in the model are small, and after all properties verified (including the previous ones), we can check the correct behavior with the builtin SPIN simulator.

Another important behavior of a pipeline is to prohibit the multiple usage of resources. If at no time the fetch, decode, execute, address bus, and data bus units are used by more than one instruction there are no resource conflicts. Cat4 in Figure 14.12 expresses this in LTL for the fetch cycle of the first instruction. Again the category will consist of 25 properties, one for each cycle. In order to satisfy this property in the model we are introducing a control process that in an initialization phase will start each instruction successively, and later makes sure that the every instruction advances in each cycle. Again the verification of all properties and simulation finishes up this iteration step. With only 4 categories of properties the basic functionality of the pipeline is now verified and working.

To make the model of the pipeline a bit more realistic, we select the user story that defines the different instruction types and their different cycle sequences from Figure 14.9. It turns out that this does not result in a new category of properties, but rather implies changes to existing properties. This step illustrates that in the iterative process, not only does the model evolve, but also the properties can evolve and get more complex later in the modeling process. To satisfy the requirement, we extend our basic instruction automaton with a wait stage and

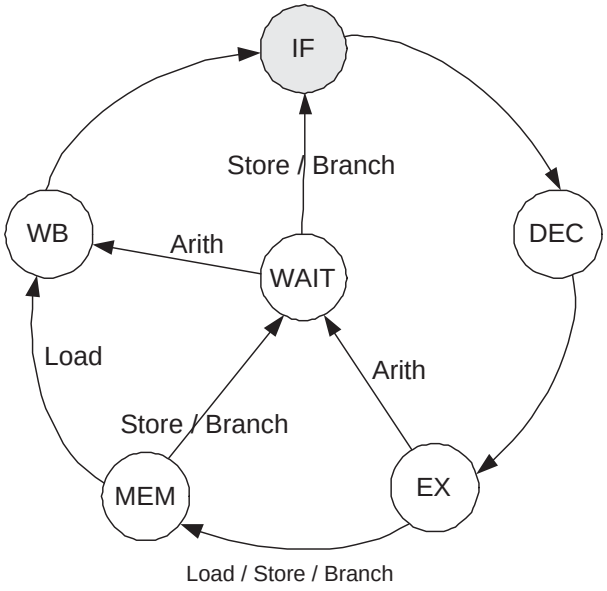


Figure 14.13: Pipeline automaton for one instruction

transitions according to Figure 14.9 (Figure 14.13). This will make sure that an arithmetic instruction for example will now go from EX to WAIT and then to WB. We have to change some properties in category 1 and 3, and add properties in all four categories. Resulting LTL examples are shown in cat 1b and 3b in Figure 14.12. Changes in the abstract model to reflect this are limited to update the FSM description for each instruction to the automaton of Figure 14.13 that means introducing the notion of an instruction type, and adding the transitions to and from the wait stage. For the control these changes are transparent since after the changes still each instruction takes 5 cycles to finish, therefore preventing the occurrence of structural hazards. Figure 14.14 shows the complete structure of the pipeline model. Of course there would still be many more details that missing in this pipeline description, such as data dependencies and forwarding, but as the steps are always the same, we will not detail all of them here.

14.6 Subsequent Work

Following the establishment of the methodology, this work has been continued and elaborated mainly at Virginia Tech. In this section I would like to give a short overview on the developments, extensions and results that were spawned by the original XFM work.

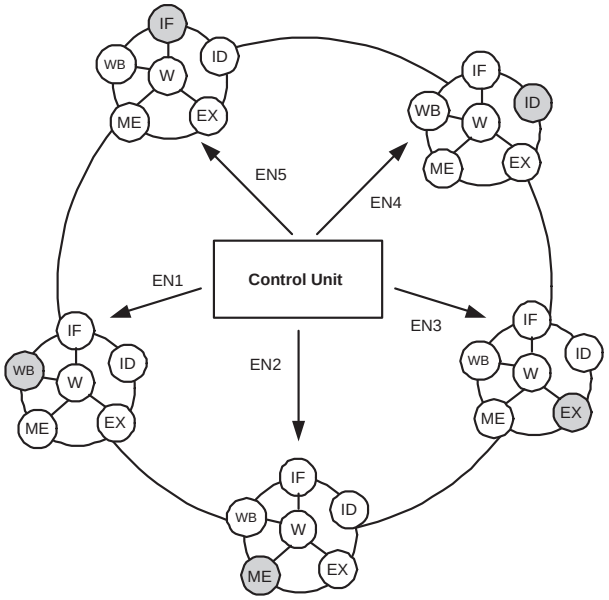


Figure 14.14: Structure of the DLX pipeline

14.6.1 Property Ordering

One further thought was as how the order of including the functionality of properties in the model is effecting the outcome. In the original XFM methodology, the order of properties was not fixed, and during modeling we discovered that the inclusion of certain properties did add more complexity to the model than others. The question arose as how to select the next property in line. One answer is experience. An experiences designer that knows the model well and has a vision of what the system should look like in the end will be more capable of choosing properties first that incur small additions to the model, so that the size of the model is growing slowly in the beginning and is getting large only towards the end. This has the big advantage, that while the model is small, it is easier to keep the overview, to make changes, and perform refactoring.

The FERMAT group then investigated as if it was necessary to be an experiences developer to make that choice, or if there was a pattern that could be automatically determined. They come up with three automated property orderings which they try on some example case studies. The three ordering schemes are:

- **random**: In random ordering, a randomly generated order is chosen. The result of this is very uncertain, as by chance it can turn out to be a very bad but also a very good ordering.

- **property based**: What is called "property based ordering" basically orders the properties depending on the level of entanglement with other properties. The entanglement is measures as a weighted sum of the occurrences of all of a properties predicates in other properties. Predicates are specific states of variables, such as $x = TRUE$.

- **predicate based**: This ordering scheme sorts the properties depending on the absolute frequency of occurrence of predicates in the properties. Properties that contain predicates that do rarely occur are modeled first.

In [SMSB05] it is shown formally and experimentally how the different ordering mechanisms perform. It contains contains some larger case studies suc has the ISA bus architecture and the bus arbitration logic of the Intel Pentium Pro processor. These case studies permit on the one hand to validate the general XFM methodology for larger hardware models, and on the other hand provide experimental material for the evaluation of the property ordering mechanisms. As a result, the experimental data as well as the formal considerations both suggest that a predicate based ordering scheme is most likely to let the model grow late in the model building process. Although this does not mean, that predicate based ordering always gives the best result, it indicates, that using predicate based ordering the state space increase of the models is more controlled than with other ordering schemes such as property based ordering or a random order.

14.6.2 GUI Toolkit

Further work in the FERMAT group also includes the implementation of a GUI toolkit, that facilitates the experimentation with different property ordering rules. Given a set of LTL properties the user can choose one of the three defined ordering rules and have the tool give a sorted list of the properties. As the designer is incrementally building the model, the tool can check the properties for the model, by interfacing to the SMV model checker [McM93].

14.7 Summary

This chapter presents a novel approach to use mechanisms from extreme programming to capture formal models. Instead of building an ad hoc formal model and come up with properties to check it against, we show that when building the model along with the properties, the model will grow linearly and get a natural structure. The major benefits are the faster model building process and the quality of the model compared with the traditional approach. Since we handle small steps, each step does add limited functionality to the model, so the debugging

process is much more directed. Another major benefit resides in the scalability. When ad hoc models are built, they usually result in a monolithic model, difficult to verify formally. Using XFM, complex models get broken down into small problems and can be built as concurrent state machines more easily while at the same time writing the corresponding formal properties.

Chapter 15

Validating Latency Insensitive Protocols

This chapter tackles the difficulties related to the integration of large scale systems on a chip. The underlying work has been performed in cooperation with the FERMAT group at Virginia Tech. I mainly worked with Syed Suhaib on latency insensitive protocols (LIP), which are protocols designed to integrate components on a chip, that can synchronously communicate with each other even if the distance between them is greater than the limit of conventional synchronous communication. Together we implemented formal models for different versions of latency sensitive protocols, starting with the one proposed by Carloni [CMSV99]. During this process I proposed a modified version of the Carloni protocol that is eliminating the need for relay stations and lowering the additional effort for routing. We successfully implemented this version as well and verified its correct functioning with the help of a model checker. Following these results, the work has been mainly continued by the FERMAT group, who made additional formalization efforts, and added a validation framework for LIP based on functional languages. This follow up work is shortly summarized in Section 15.4.

15.1 Introduction

In the current System-on-chip (SoC) based design, reduced time-to-market demands efficient reuse of complex components. This has led to the idea of developing libraries of Intellectual Properties (IPs) or reusable components. The integration of such complex IPs on SoC and communication between them has shifted the performance bottleneck of the system from computation to communication. With clock frequencies of these IPs in the multiple gigahertz range, and the interconnection distances staying almost constant with the chip size, we have hit the limit, where the signal propagation distance during one clock cycle is

shorter than the longest wire. The solution is to distinguish short interconnects from long interconnects, and use the long interconnects preferably on low traffic inter-component communication. Intelligent repeaters on their interconnects make sure that no value sent through them is lost. This approach is known as Latency Insensitive design. The idea is to create a design from a specification without assuming any latency in the interconnects, such that the resulting design is latency equivalent to the specification. Informally, latency equivalence means that given the same input signals, the output signals have the same ordering of events except for interspersed absent events.

While this is solving the fundamental problem, there are still issues that make the application unsatisfactory. We see two main concerns: (i) These repeaters or relay stations are additional components that have to be placed on the floor plan, and consequently require changes in the original placement and routing. Several iterations of a lengthy process may be necessary in order to reach a state where all timing constraints are met. (ii) In such a completely synchronous design all components have to run the same clock frequency. Reusing IPs from different origins however often imply that these IPs are designed to run at varying frequency ranges. Furthermore, only a few components have to run with the fastest clock, others could run at a much slower clock, while still meeting all their performance constraints.

In this chapter, we illustrate a technique with an example to show how to successively eliminate these two constraints, reducing overhead caused to the designer in terms of design time, while at the same time leaving them more room in their power budget. In the first step, we show how to get rid of the relay stations, by putting some extra intelligence into the component interface. In the next step, we show an extension of these interfaces that allow for components to have defined rational clock relations, in contrast to having the same fixed clock for all components. Both of these improvements can be done systematically, while minimizing changes in the actual component and therefore eliminating additional sources of errors. We employ formal verification as our strategy for ensuring that as we gradually refine our protocols, we maintain functional equivalence to the original intent of the latency insensitive design.

15.1.1 Related Work

Latency insensitive protocols (LIP) for systems with long interconnection delays (i.e. greater than one clock cycle) were proposed by Carloni et al [CMSSV99, CMSV01, CMSV99] for single-clock SoCs. All processes with long interconnects are encapsulated in a wrapper to derive a process that is latency equivalent to the actual process, without having to modify the internals of the original IP. Relay stations are added along the long interconnection wires similar to repeaters in

order to ensure successful data transfer. They contain at least two registers and a small control logic. The insertion of these relay stations increases the number of elements to route and requires additional space on the chip for placement. Once it is determined where relay stations have to be added based on the length of the wires, placement and routing of the entire chip design now including the relay stations have to be redone. Several iterations for placement and routing are needed in order to get a configuration that satisfies all interconnection constraints.

All components of such latency insensitive (LI) designs are assumed to operate with the same clock. Singh and Theobald generalize the LI theory for GALS systems [ST04]. In their approach, complex FSMs control all input and output signals. The communication network is implemented as an asynchronous system to connect modules with different clocks. Overall this approach is associated with heavy penalties in terms of implementation costs and performance. Casu and Macchiarulo show how to reduce chip area compared to Carloni's approach [CM04]. They use a smart scheduling algorithm for the functional block activation and substitute the relay stations with simple flip-flops. One disadvantage of this approach is that the schedule has to be computed a priori and depends on the computation in the process. If any change is made in any process, it may result in change of flow of tokens. In this case, the schedule has to be recalculated which is an expensive procedure. In our approach, we do not face the problem of implementing complex FSMs, asynchronous communication protocols, or scheduling computations. Instead we propose to solve the latency problem without using relay stations along the wires. We also generalize the solution for multi-clock systems where communication is done based on a global/communication clock and the process interfaces bridge the global and the local clock to ensure correct functionality of the processes. We formally model the family of protocols that we propose and verify them for latency equivalence with a corresponding synchronous system.

GALS systems, which are a more general approach than the pure LIP, have been subject of research in Polychony as well. In [TG05] and [LTL03] the properties endochrony and endo-isochrony are presented. These express precisely what is needed to natively model GALS architectures. Endochrony as defined in the Polychrony context, means that, given an external (asynchronous) stimulation of its inputs, it still exhibits synchronous behavior. In other words, an endochronous process can be stalled without changing its behavior. This is exactly the class of processes that is required for the application of LI techniques, they are also called stallable or patient processes. The property of endo-isochrony allows for the compositional design of distributed processes. It shows that a correct (i.e. flow preserving) refinement of a GALS specification on a distributed architecture can only consist in the substitution of resynchronization clocks between the components of the architecture. This has been shown by Pascal Aubry [Aub97, AL96]

and implemented in the polychrony platform.

15.1.2 Spin

SPIN [Hol03b] is a model checker used extensively for formal verification of systems. SPIN is used to trace logical design errors and to check the consistency of the specification. Its basic building blocks include asynchronous processes, message channels, synchronizing statements, and structured data. We use these basic blocks to write synchronous models. The communication is done through global variables. Since the processes run asynchronously in SPIN, we synchronize the execution of all processes with a central clock controller in order to make our model behave like a synchronous system. The execution of each component depends on a flag set by the clock controller, which is reset by the component at the end of its execution. The clock controller then waits for all components to finish execution before it starts the next cycle. As in a real synchronous system, the duration of a cycle (i.e. the maximum system frequency) is determined by the slowest component.

15.2 Contributions

The main contributions of this chapter are as follows:

- Formal modeling and validation of existing latency insensitive protocols.
- New refinement-based approach to single-clock latency insensitive systems.
- Formal modeling and validation of our new approach. New approach to latency insensitive systems for multi-clock systems.
- Formal modeling and validation of the multi-clock latency insensitive system.

15.3 Methodology

15.3.1 Carloni's Latency Insensitive Protocol

In the LI approach of Carloni et al each process is encapsulated in a wrapper (called "shell") that is forming the interface to other shells. Logic blocks called relay stations are placed along the wires that are longer than the signal propagation distance during one clock cycle. These relay stations work as pipeline blocks to send data from one shell to another over a long wire. The shell reads all incoming values, filters out empty events, and feeds the process with valid input

events. If any input signal does not have a data available, the process is stalled until all input signals are present. When the process completes computation, the shell writes events to the output wires, but only if the respective connected component is accepting events.

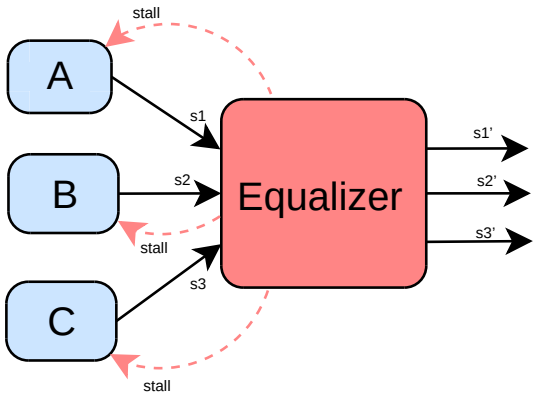


Figure 15.1: Equalizer Example

We present our LI systems based on this approach. So first we made a formal model of the Carloni LIP. Figure 15.2 shows a block diagram of this model. In this implementation we put all functionality of the wrapper into an equalizer process. An equalizer is modeled such that it reads all the input events from the incoming signals. An input is considered only when it is associated with an informative event. In informative event is an event that carries effective information. An informative event is an event that does not carry information. This happens for example when a clock tick occurs but no new value of the signal has arrived since the last clock tick. The equalizer only forwards data when all its input signals provide informative events. If an input signal sends an absent event in one cycle, the equalizer sends a stall signal via a feedback to all processes that had sent an informative event and it suspends the execution of other processes. When the process itself is disabled by its equalizer, it is producing τ events at its outputs, therefore causing the components before to wait until it can resume operation.

Let us consider an example of an equalizer and three processes: A, B, C (shown in Figure 15.1). Each of the processes connect to the equalizer with signals s_1, s_2, s_3. Also the equalizer outputs stall signals $stall_A, stall_B, stall_C$ to the respective processes. Let us assume that processes B and C produce informative events whereas process A outputs an absent event. In this scenario, the equalizer will set $stall_B$ and $stall_C$ signals to stall the processes B and C in the next cycle and output absent events on s_1', s_2', s_3'. Once it receives an informative event on s_1, it will remove the stalls that were set and produce the corresponding outputs.

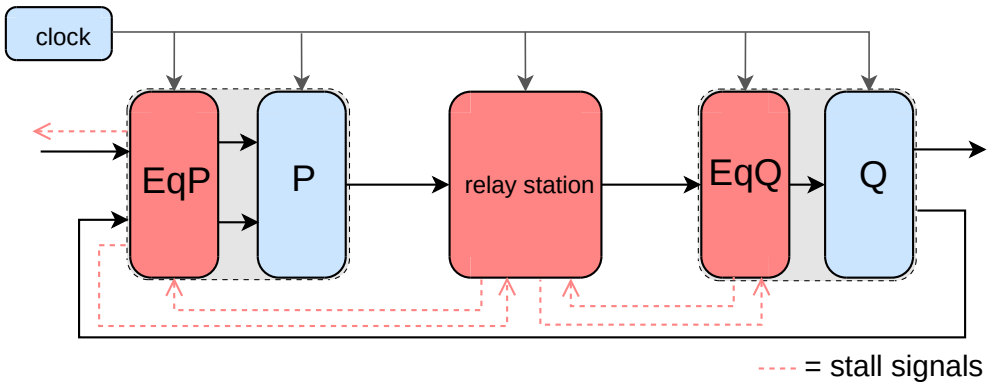


Figure 15.2: LI system proposed by Carloni

The equalizer modeled in the LI implementation also consists of a stall signal generator that connects to the actual process. An example of this LI approach is shown in Figure 15.2. In the example, there are two processes P and Q that are encapsulated in wrappers. The process P has two inputs, one external input and the other is a feedback from the process Q. The shell of process P reads the data from its inputs and performs some computation and sends an output to Q via a relay station. The process Q has two outputs. One is an external output and the other one connects to the input of process P. For simplicity, this connection is considered as a short wire, not necessitating a repeater. The setup is as follows: EqP and EqQ are the equalizers of the corresponding processes. The global clock is connected to all the processes. There is a delay of two cycles on the interconnection between shell P and shell Q. Therefore, one relay station is placed on the interconnect to permit successful transfer of data from process P to process Q. The solid arrows denote the transfer of values where as the dotted lines denote the transmission of events. Each process has enable signals set at the next process station because if a process is not ready to accept data, it can disable the previous processes to stop them from producing more tokens. This feedback mechanism is preventing the usage of infinite buffers. Note that we assume that processes considered here are functions and hence deterministic and they are monotonic as well.

15.3.2 Eliminating Relay Stations

First step to generalize the original LI approach is to eliminate the need for relay stations. There are several advantages to this. First of all, even though we increase the size of certain elements by adding the LIP interface, we reduce the number of elements compared to Carloni, which simplifies routing and placement.

Also, we reduce the process of re-iterating over the routing and placement after estimating the actual delays since we do not have to insert new components in between wires for long interconnections.

The basic idea of our approach is that if a event between two components takes two clock cycles, we only send one value every second clock cycle thereby adapting the communication speed between components to their distance. An outgoing interface takes care of this restriction, and stalls the sending component whenever it delivers values too fast.

While this eliminates all relay stations, it however slows down the system significantly due to an increasing number of stalls. We eliminate this slowdown by additional communication lines. The number of interconnects needed depend on the interconnect delay. For example, if there is an interconnect delay of n cycles, then n interconnects are placed in-between the processes. Note that in most cases n is in the range of 2 or 3. These additional wires an extra cost, but in modern processes it is not very expensive to add parallel wires. Only in rare cases of very sparse communication does it make sense to use a LI implementation for more than three cycle distances, since stalling is bound to slow down the rest of the system otherwise.

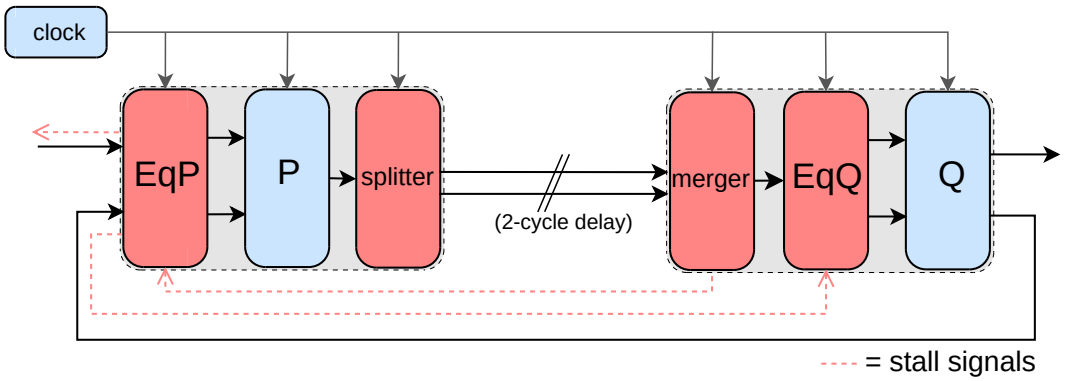


Figure 15.3: LI system without relay stations

To ensure that the events are correctly transferred from one process to another through long interconnects, we add extra interconnects that are bounded by a *splitter* on the source and a *merger* on the sink. The splitter is implemented at the output of a process, and it transfers events on the corresponding interconnect. The splitter only puts one event on one of the output interconnects and absent events are placed on the rest of the signals at a particular time stamp. Complementary to the splitter, we implement a component that is able to merge multiple inputs into one signal depending on their availability. This merger reads the corresponding events from these interconnects based on the placement

of events by the splitter. For each long interconnect in our LI implementation we need a splitter merger combination, that means we insert a splitter on one end and a merger on the other end.

Figure 15.3 shows the structure of this implementation now avoiding the use of relay stations by using splitter and merger processes attached to the respective components. Each component still has an equalizer for the inputs, however on the output side, long interconnects are connected by the bridge. The throughput of the system is exactly the same as the throughput of a system with relay stations.

15.3.3 Components with Different Clocks

An LI system as defined by Carloni is still a synchronous system where all components are connected to the same clock and work with the same speed. We generalize this definition towards a multi-clock implementation where we allow components with different clocks connected via arbitrarily long wires. At this time, however, we are only permitting the use of components with defined, rational clock relations. This approach therefore makes it possible to connect components where one is working with a clock three times faster than the other one, or where the ratio of their clocks is 13:17. The global clock is always the fastest clock in the system, and will have a defined rational relation with the clock of any component. As before, we assume that each process is reading the inputs during the first half of the clock cycle, and writes the outputs during the second half of the clock cycle. This way we are avoiding any read/write conflicts. In order to support systems with multiple clocks we have to modify the equalizer and the merge to take into account different clock ratios. The equalizer now has the information about the clock ratio of corresponding process in relation to the global clock. This is to make sure that the equalizer delivers input values to the system at the rate that it expects them. The process does its processing and in the write phase writes a value to its outputs. This value is read by the Splitter which now also knows about the speed of the attached process. As before, it also knows about the distance to the connected components, however it does not have to know anything about the clock speed of the connected components. The protocol is taking this into account implicitly. This is also why the merger process is unchanged from the single clock version. The merger is still merging together signals from multiple inputs, depending on their availability. The merger always works on the main system clock so for its task it does not have to know anything about the actual clock of the process. The equalizer that is in between is taking care of that part.

Figure 15.4 shows the multi-clock LI implementation of the system. The main processes run based on the local clocks, whereas, the interfaces of the processes run based on the global clock. The basic blocks of the module are the same as

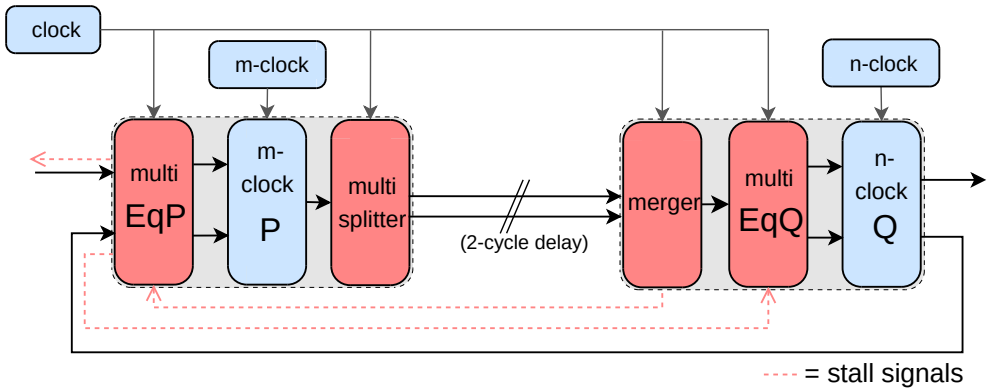


Figure 15.4: LI implementation with different clocks

shown in the previous implementations. Each process has an extended equalizer and an extended splitter. In the read phase of each process, the data is read from extended equalizer. In the writing phase of the process, the data is written on the signals read by the splitter. On long interconnects, the extended splitter controls the writing of the values on its signals

15.3.4 Verification of LI Protocols

To ensure that all members of this family of LIPs are functionally correct, we check for their latency equivalence with a corresponding synchronous system. By latency equivalence, we mean that all the values corresponding to the same set of informative events must be equal. We model the two systems in SPIN and feed the same sequence of tokens to the synchronous model as well as the LI model, and compare the output tokens to be equal. The setup for verification of the two systems is shown in Figure 15.5.

The figure shows that the same input is fed into both systems. In order to test for all cases, we used a random generator giving the same sequence of values to both the systems. The corresponding processes of both the system perform the same computation. The output of the systems is fed to the Comparator. The Comparator is a process that embeds an equalizer that takes the inputs from the signals of the two implementations and has an assertion that checks that the values read are equal at every clock tick. This assertion ensures that the values received at the output are equal. Verification to check for the latency equivalence was done for LIP with relay station and synchronous model, LIP without relay station and components with same clock, and finally LIP without relay station that permit the use of components with different clocks but with a known clock ratio. For verification of the latency equivalence for this LIP, various different

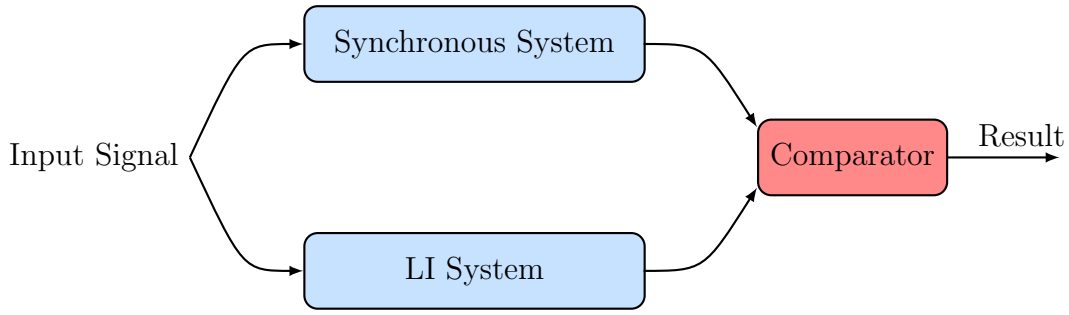


Figure 15.5: Verification

ratios for the clocks were taken into account.

15.4 Subsequent Work

This work has been continued mainly in the FERMAT lab of Virginia Tech by Sandeep Shukla and his students Syed Suhaib and Deepak Mathaikutty. They added two main contributions to this work, the first one being an effort of formalization, that is to make the understanding of different terminologies in LIPs unequivocal. The second addition is the implementation of a functional programming framework, that permits to build and validate LIP models.

Formalization In order to be able to completely rely on the tools and protocols offered by a latency insensitive methodology, it is important that everyone is understanding the exact same thing. This is only possible through rigorous formalization of terminology. In [SMBS06] we find formal definitions of terms and processes such as *splitter*, *merger*, and *equalizer* in order to clarify what exactly they are performing.

Functional Language Framework Formally verifying protocols with a model checker such as SPIN is giving definitive answers about the correct functioning of a protocol for the entire state space, however it is sometimes not easy to correctly describe a system in SPIN. More importantly, complex protocols are getting difficult to verify with a model checker since the state space is growing exponentially with the size of the model, and processors and memory of available machines set a limit to the complexity, which is difficult to overcome. Even if verification finishes in a couple of hours, debugging and modifications are getting extremely

expensive as each and every small modification necessitates the reverification of the model.

In order to overcome this Syed Suhaib et al. developed a functional programming framework, that gives a simulation based approach to the validation of LI protocols. While this is not matching a formal model checker for correctness, it scales much better and can be used to complement a model checking based approach. Functional languages perform computation by function application, and therefore provide an abstract and elegant way to express computation. Experiments show, that SML is quite apt for the modeling of LIP models as many of the definitions can be easily mapped to recursive functions.

15.5 Summary

In this chapter, we propose a framework for the validation of LI systems. The LI systems along with their synchronous idealization can be modeled together and checked for latency equivalence. We show techniques for validation using our framework. We model the entire framework in PROMELA and formally verify using the SPIN model checker. The latency equivalence is expressed as a formal property and verified for equivalence by a direct comparison of the outputs of the original system and the LI system. A possible extension would be to modify the LI protocols to GALS system that contain system components that are completely asynchronous, without any (known) relation between their respective clocks.

Conclusion

The work presented in this document has been performed in the ESPRESSO team of INRIA at IRISA and during several visits at the FERMAT lab of Virginia Polytechnic Institute and State University. Our main interest is to show how the design of embedded systems in general can benefit from the use of formal methods. We illustrate this with the help of several examples addressing difficulties and use cases in a CoDesign flow.

Automated generation of formal models. There are several tracks leading to this goal. One possibility is to automatically build a formal description based on nonformal system models. This is to satisfy an important condition for the widespread adoption of formal methods in the general purpose design flow of embedded systems. The idea is to hide the complexity of formal methods to the user, i.e. the engineers, while still drawing a maximum of benefit from the clear definitions and formal toolsets. For our work on the automated construction of formal models we use SystemC as the language of design entry. The actual work consists in three parts: (i) The definition of a methodology to infer behavioral types from SystemC modules, (ii) a case study that puts this theory to practice and implements the actual toolflow, and (iii) the design of a tool for the extraction of structural information from SystemC in order to preserve the structure of the models.

The definition of the theory for the behavioral type inference is a first important step. Based on the iSTS algebra proposed in [TG04] we represent the behavior of a program by a type or proposition. In order to be able to make the transition to SystemC, we define a formal syntax for a subset of the language. With the help of this formal syntax, behavioral types can be inferred from nonformal SystemC components. After the definition of the inference rules, we make the connection to the POLYCHRONY framework by defining a translation from the iSTS behavioral type system into SIGNAL.

In a second step we try to put this theory to work. We identify tools that help in implementing this process. We use the Static Single Assignment (SSA) as the central format for the process. SSA has a straightforward transformation into SIGNAL and is the intermediate format of the GCC. Therefore GCC can be used

to translate from SystemC into SSA. We then only have to transform the SSA into SIGNAL following the previously defined transitions. A small case study of an FIR filter design exemplifies the whole transformation process and illustrates how the tools are being used. We also show how static and dynamic properties of the model can be verified with the help of the model checker SIGALI. Finally we demonstrate how the models can be abstracted in order reduce complexity, which is important for the verification of large models.

The model resulting of such a transformation does, however, not retain the structure of the original design. This is an important limitation as for large models it is difficult to keep the overview and track errors back to the source. In order to eliminate this concern, we extend the methodology in a third step to extract the structural information from the existing system description first, and transform this structure into the desired target formalism. Then each of the module behaviors can be transformed separately in order to fill in the empty process declarations of the formal structure. We describe a methodology that spans the whole process, starting from a structured SystemC program to an equally structured SIGNAL description. The effort of parsing SystemC is minimized by the smart use of open source tools such as Doxygen, the Xerces XML parser, and GCC. The separate transformation of structure and behavior results in a formal model that on the one hand contains the behavioral detail needed in order to perform desired formal checks, and on the other hand it preserves the control structure of the original model, enabling an easier backtracing of errors, a modular component reuse, and the compiler can do more optimizations for example for scheduling. Beyond this, the structural information obtained can be used for plenty of other applications. We show with some examples how easily this can be exploited for other uses such as design visualization or automated test generation. The front end part of the implementation has been made available as an open source project, so that it can be easily used and modified by others.

Incremental model building. Another possibility for the use of formal methods in the design flow of embedded systems is to start with a formal model in the first place. As this allows specifications to be captured much more clearly and unequivocally, this is by definition a good way to start a system description. However starting off with a formal model also holds many hazards and pitfalls that have to do for example with the nonintuitive understanding of formal expressions and the difficulty to keep track of the size of the state space of the generated model. We present a methodology that tries to simplify the process of formal model building. It borrows techniques from extreme programming to incrementally build formal models along with the associated formal tests. To incrementally build a model, the initial specification is cut down into small self contained parts of the functionality. We start by formally modeling one of these

functional blocks and at the same time write one or several formal properties that verify it. After model checking that the model satisfies these properties, the other functional blocks are added one by one, each time verifying if the model satisfies the corresponding formal properties as well as all the previous ones. The result is a verified formal model containing all of the initial specification and a set of corresponding formal properties. As the model is verified from the beginning and all along the design process, errors are detected earlier and big leaps in the state space are detected at the source and can then possibly be avoided or reduced.

Latency insensitive protocols. With the growing complexity of embedded systems comes the problem that for the integration on chip, connecting wires between components can get longer than allowed for a desired processing speed. One method to go beyond this limit is to insert buffers along these communication wires and add a certain logic that takes care of the missing values and additional delays. This logic is not easy to produce, and may induce major changes in the design of the concerned components or the complete system architecture. The theory of latency insensitive protocols established ways that permit the automated generation of protocol interfaces where needed in the design assuming all components fulfill certain conditions. We present a way of how to formally validate the result of such a conversion against a corresponding system without delays on the wires. We show a method of how existing LIPs can be formally modeled and verified. This method is then applied successfully to the protocol proposed by Carloni by verifying it in an easy to follow test setup. We then introduce a modified protocol that removes the need for repeater stations along long wires without losing system performance and verify its correctness again with our formal setup.

Future Work

The work present in this document describes a methodology with which it is possible to extract formal behavioral types from SystemC descriptions. The possible benefits of such a project are clear, however, the real power of this approach can only be demonstrated with a tool that can perform this task automatically on a sufficiently large subset of the input language. At this point in time there are some steps missing to this goal. While the automated extraction of structure and the generation of the corresponding SIGNAL process skeletons is implemented and working, we are currently working on the automated transformation of the SSA code to SIGNAL. Once the SSA code can be automatically translated, a SIGNAL library has to be built that corresponds to the SystemC library in C++. This library has to model all SystemC constructs such as `sc_signal` or `sc_event` or at

least a reasonably large subset of them. Some of these functions are available from previous work [GGG06], but this will have to be completed and tested. In oder to entirely reflect the original SystemC design, the SystemC scheduler will also have to be modeled in SIGNAL. What equally remains to be done is the integration of the modular model generation with SystemCXML in order to automatically fill the empty process definitions in the generated SIGNAL skeleton.

Once the automated type extraction is working, it will be interesting to see how it scales with the size of the projects, and how it behaves for different types of systems. As the SSA to SIGNAL translation is valid also for other languages supported by a GCC front end such as Fortran, Java, and ADA, it would equally be interesting to see how the behavioral type extraction would work for these languages and how we can control simulation and verification performance by using different levels of abstraction.

The work on incremental formal modeling shows that extreme programming techniques can be used for the process of building formal models. The results are encouraging and seem to indicate that building formal models incrementally is faster and more reliable than using standard techniques. However, there is still missing a representative study that examines the effects of this method in terms of design time, performance, and quality for a number of projects. While such an extensive study could be conceived with the cooperation of design companies, it would require considerable resources and its outcome may largely depend on the type of the projects and the experience of the designers. For extreme programming, number of such studies have been attempted [SASB04, LWC04], but despite its wide adaption there is no prevalent opinion to an empirical advantage of the method [Lay04, LBB+02]. So for incremental formal modeling, as an empirical study is far out, a next step would be its deployment in an industrial environment, to get some feedback from developers and project managers.

The verification of latency insensitive protocols is an important subject as easy to follow verification techniques are missing. A big advantage of the presented technique is the transparency of the verification mechanism, however one concern is the scaling of the method as for large models the verification state space is getting unmanageable. More complex protocol implementations may become difficult or impossible to verify with this method. Another issue is that the formal models for the protocols are not very intuitive to write, and that considerable effort has to be put into correctly modeling one specific protocol implementation. The follow up work at Virginia Tech is addressing these concerns by adding a functional verification framework. Modeling the protocols in a functional language is much easier than building a model in Promela. Also, functional programs scale better, such that even complex implementations do not create difficulties in terms of verification time or memory usage. Functional languages obviously do not allow for formal verification, so in a further step we could imagine a mixture

of the two approaches. While the system level view of the protocol can more efficiently be modeled in a functional framework, core functionalities of the protocols could be modeled in a formal language. This can be done in a way such that the combination of the two assures the core functionality of the protocol as well as the high level functionality.

Bibliography

[ABL95] T. P. Amagbegnon, Loic Besnard, and Paul Le Guernic. Implementation of the data-flow synchronous language signal. In *Conference on Programming Language Design and Implementation*. ACM Press, 1995.

[AL96] P. Aubry and P. Le Guernic. On the desynchronization of synchronous applications. In *Proceedings of the 11th International Conference on Systems Engineering, ICSE'96*, Las Vegas, Nevada (USA), July 1996. University of Nevada.

[Ass06] Semiconductor Industry Association. International technology roadmap for semiconductors (ITRS) 2005 edition. http://www.itrs.net/Common/2005ITRS/Home2005.htm, January 2006.

[Aub97] P. Aubry. *Mises en oeuvre distribuées de programmes synchrones*. PhD thesis, Université de Rennes 1, IFSIC, October 1997.

[BDBS99] E.A. Boiten, J. Derrick, H. Bowman, and M.W.A. Steen. Constructive consistency checking for partial specification in Z. *Science of Computer Programming*, 35(1):29–75, September 1999.

[Bec00] Kent Beck. *Extreme Programming explained: Embrace change*. Addison Wesley, 2000.

[Ber00] G. Berry. *The Foundations of Esterel*. MIT Press, 2000. Editors: G. Plotkin, C. Stirling and M. Tofte.

[BG92] G. Berry and G. Gonthier. The Esterel synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19:87–152, November 1992.

[BGJ+97] F. Balarin, P. Giusto, A. Jurecska, C. Passerone, E. Sentovich, B. Tabbara, M. Chiodo, H. Hsieh, L. Lavagno, A. Sangiovanni-

Vincentelli, and K. Suzuki. *Hardware-Software Co-Design of Embedded Systems, The POLIS Approach*. Kluwer Academic Publishers, April 1997.

[BH95a] J. P. Bowen and M. G. Hinchey. Seven more myths of formal methods. *IEEE Software*, 12(4):34–41, July 1995. Previously available as University of Cambridge Computer Laboratory Technical Report 357, December 1994 and Oxford University Computing Laboratory Technical Report PRG-TR-7-94, June 1994.

[BH95b] J. P. Bowen and M. G. Hinchey. Ten commandments of formal methods. *IEEE Computer*, 28(4):56–63, 1995.

[BHLM94] Joseph Buck, Soonhoi Ha, Edward A. Lee, and David G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogenous systems. *Int. Journal in Computer Simulation*, 4(2):155–182, April 1994.

[BLJ91] Albert Benveniste, Paul Le Guernic, and Christian Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Science of Computer Programming*, 16(2):103–149, 1991.

[BM02] Luca Benini and Giovanni De Micheli. Networks on chips: A new soc paradigm. *Computer*, 35(1):70–78, 2002.

[BMPS04] David Berner, D. A. Mathaikutty, H. D. Patel, and S. K. Shukla. FERMAT's SystemC Parser. http://systemcxml.sourceforge.net, 2004.

[BPM+05a] David Berner, Hiren Patel, Deepak Mathaikutty, Sandeep Shukla, and Jean-Pierre Talpin. SystemCXML: An extensible SystemC front end using XML. Technical Report 2005-06, Virginia Polytechnic Institute, FERMAT Lab, Blacksburg, VA, USA, June 2005.

[BPM+05b] David Berner, Hiren Patel, Deepak Mathaikutty, Jean-Pierre Talpin, and Sandeep Shukla. SystemCXML: An extensible SystemC front end using XML. In *Proceedings of the forum on specification and design languages (FDL)*, Lausanne, Switzerland, September 2005.

[BSSH05] David Berner, Syed Suhaib, Sandeep Shukla, and Harry Foster. XFM: Extreme Formal Method for Capturing Formal Specification into Abstract Models. Technical Report 2003-08, FERMAT Lab Virginia Tech, Blacksburg, VA USA, December 2005.

[BSST04] David Berner, Syed Suhaib, Sandeep Kumar Shukla, and Jean-Pierre Talpin. *Formal Methods and Models for System Design*, chapter Capturing Formal Specification into Abstract Models. Kluwer Academic Publishers, October 2004.

[BST92] David Becker, Raj K. Singh, and Stephen G. Tell. An engineering environment for hardware/software co-simulation. In *Design Automation Conference*, pages 129–134, 1992.

[BTSG04] David Berner, Jean-Pierre Talpin, Sandeep Shukla, and Paul Le Guernic. Modular design through component abstraction. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 202–211, Washington DC, USA, September 2004.

[BWH+03] Felice Balarin, Yosinori Watanabe, Harry Hsieh, Luciano Lavagno, Claudio Passerone, and Alberto L. Sangiovanni-Vincentelli. Metropolis: An integrated electronic system design environment. *IEEE Computer*, 36(4):45–52, 2003.

[Cad05] Cadence Design Systems Inc. Incisive unified simulator. Datasheet 5418C 04/05, June 2005.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. on Programming Languages and Systems*, 13(4):451–490, 1991.

[CM04] M. Casu and L. Macchiarulo. A new approach to latency insensitive design. In *Design Automation Conference*, 2004.

[CMA02] Srihari Cadambi, Chandra S Mulpuri, and Pranav N Ashar. A fast, inexpensive and scalable hardware acceleration technique for functional simulation. In *DAC '02: Proceedings of the 39th conference on Design automation*, pages 570–575, New York, NY, USA, 2002. ACM Press.

[CMSSV99] Luca P. Carloni, Kenneth L. McMillan, Alexander Saldanha, and Alberto L. Sangiovanni-Vincentelli. A methodology for correct-by-construction latency insensitive design. In *ICCAD*, pages 309–315, San Jose, CA, USA, November 1999.

[CMSV99] L.P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli. Latency insensitive protocols. In *11th International Conference on*

Computer-Aided Verification, volume 1633, pages 123–133, Trento, Italy, July 1999. Springer Verlag.

[CMSV01] L. Carloni, K. McMillan, and A. Sangiovanni-Vincentelli. The theory of latency insensitive design. *IEEE Transactions on Computer Aided Design of Integrated Circuits and System*, 20(9):1059–1076, 2001.

[Cor] Open Cores. Free open source IP cores and chip design. http://www.opencores.org.

[dAH01] Luca de Alfaro and Thomas A. Henzinger. Interface theories for component-based design. In *First International Workshop on Embedded Software*, pages pp. 148–165. Lecture Notes in Computer Science 2211, Springer-Verlag, 2001.

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[DSG03] Frederic Doucet, Sandeep Shukla, and Rajesh Gupta. Typing abstractions and management in a component framework. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE Press, January 2003.

[EAH05] Christian J. Eibl, Carsten Albrecht, and Rainer Hagenau. gSysC: A Graphical Front End for SystemC. In *Proceedings 19th European Conference on Modeling and Simulation (ECMS)*, Riga, Latvia, June 2005.

[FE] Edison Design Group C++ Front-End. Edison design group c++ front-end. Website: http://edg.com/cpp.html.

[Fou] The Apache Software Foundation. Xerces C++ validating XML Parser. Website: http://xml.apache.org/xerces-c/.

[Fre04a] Free Software Foundation. The GCC tree-ssa documentation. http://gcc.gnu.org/onlinedocs/gccint/Tree-SSA.html, 2004.

[Fre04b] Free Software Foundation. The GNU compiler collection. http://gcc.gnu.org, 2004.

[FSG03] Frederic Doucet, S. Shukla, and R. Gupta. Introspection in System-Level Language Frameworks: Meta-level vs. Integrated. In *Design and Test Automation in Europe*, 2003.

[GCNCM92] Rajesh Kumar Gupta, Jr. C. N. Coelho, and Giovanni De Micheli. Synthesis and simulation of digital systems containing interacting hardware and software components. In *DAC '92: Proceedings of the 29th ACM/IEEE conference on Design automation*, pages 225–230, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[GG02] Abdoulaye Gamatié and Thierry Gautier. Modeling of modular avionics architectures using the synchronous language. In *Proceedings of the 14th. Euromicro Conference on Real-Time Systems, work-in-progress session*. IEEE Press, 2002.

[GGG06] Abdoulaye Gamatié, Thierry Gautier, and Paul Le Guernic. Synchronous design of avionic applications based on model refinements. *Journal of Embedded Computing (JEC)*, 2006. IOS Press (to appear).

[GKNV93] Emden R. Gansner, E. Koutsofios, S. North, and K.-P. Vo. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering*, 19(3):214–230, March 1993.

[GLLA03] Daniel Große, Rolf Drechsler Lothar, Linhard, and Gerhard Angst. Efficient Automatic Visualization of SystemC Designs. In *Proceedings of the Forum on specification and design languages (FDL)*, Frankfurt, Germany, September 2003.

[GLMS02] T. Groetker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[GN00] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Softw. Pract. Exper.*, 30(11):1203–1233, 2000.

[GNJ+96] Gopi Ganapathy, Ram Narayan, Glenn Jorden, Denzil Fernandez, Ming Wang, and Jim Nishimura. Hardware emulation for functional verification of k5. In *DAC '96: Proceedings of the 33rd annual conference on Design automation*, pages 315–318, New York, NY, USA, 1996. ACM Press.

[Gre] GreenSocs. Pinapa: A SystemC front-end. Website: http://greensocs.sourceforge.net/.

[GZD+00] D. D. Gajski, J. Zhu, R. Domer, A. Gerstlauer, and S. Zhao. *SpecC: Specification Language and Methodology*. Kluwer Academic Publishers, January 2000.

[Hal90] A. Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–19, September 1990.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. In *Proceedings of the IEEE, vol.79(9)*, pages 1305–1320, September 1991.

[HDE+93] Laurie J. Hendren, Chris Donawa, Maryam Emami, Guang R. Gao, Justiani, and Bhama Sridharan. Designing the McCAT Compiler Based on a Family of Structured Intermediate Representations. In *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, pages 406–420. Springer-Verlag, LNCS 757, 1993.

[Hen00] Jesper G. Henriksen. *Logics and Automata for Verification - Expressiveness and Decidability Issues*. PhD thesis, Basic Research in Computer Science (BRICS), University of Aarhus, Denmark, June 2000.

[Hol03a] Gerhard J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison Wesley, Boston, MA, September 2003.

[Hol03b] Gerhard J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison Wesley, Boston, MA, September 2003.

[HPG02] John L. Hennessy, David A. Patterson, and David Goldberg. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, CA, 3rd edition, may 2002.

[IRI] IRISA, project ESPRESSO. The polychrony workbench. http://www.irisa.fr/espresso/Polychrony.

[JVBEK91] Jr. Jack V. Briner, John L. Ellis, and Gershon Kedem. Breaking the barrier of parallel simulation of digital systems. In *DAC '91: Proceedings of the 28th conference on ACM/IEEE design automation*, pages 223–226, New York, NY, USA, 1991. ACM Press.

[Klo05] Debra Klopfenstein. Verifying large models in rtl simulation. http://www.eetimes.com/showArticle.jhtml?articleID=172901509, October 2005.

[KTBB06] Hamoudi Kalla, Jean-Pierre Talpin, David Berner, and Loic Besnard. Automated translation of c/c++ models into a synchronous formalism. In *IEEE International Conference and Work-*

shop on the Engineering of Computer Based Systems (ECBS), Potsdam, Germany, March 2006.

[LAN+03] J. Lapalme, E. M. Aboulhamid, G. Nicolescu, L. Charest, F. R. Boyer, J. P. David, and G. Bois. .NET Framework – A Solution for the Next Generation Tools for System-Level Modeling and Simulation. In *Design and Test Automation in Europe*, 2003.

[Lay04] Lucas Layman. Empirical investigation of the impact of extreme programming practices on software projects. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 328–329, New York, NY, USA, 2004. ACM Press.

[LBB+02] Mikael Lindvall, Victor R. Basili, Barry W. Boehm, Patricia Costa, Kathleen Dangle, Forrest Shull, Roseanne Tesoriero Tvedt, Laurie A. Williams, and Marvin V. Zelkowitz. Empirical findings in agile methods. In Don Wells and Laurie A. Williams, editors, *XP/Agile Universe*, volume 2418 of *Lecture Notes in Computer Science*, pages 197–207. Springer, 2002.

[LL98] Bilung Lee and Edward A. Lee. Hierarchical concurrent finite state machines in ptolemy. In *International Conference on Application of Concurrency to System Design (ACSD)*, pages 34–40. IEEE Computer Society, 1998.

[LTL03] Paul Le Guernic, Jean-Pierre Talpin, and Jean-Christphe Le Lann. Polychrony for system design. *Journal of Circuits, Systems and Computers*, 12(1), April 2003.

[LW98] Jacob Lehraum and Bill Weinberg. Ide evolution continues beyond eclipse. http://www.eetimes.com/, June 1998.

[LWC04] Lucas Layman, Laurie A. Williams, and Lynn Cunningham. Exploring extreme programming in context: An industrial case study. In *Agile Development Conference*, pages 32–41. IEEE Computer Society, 2004.

[LX01] Edward A. Lee and Yuhong Xiong. System-level types for component-based design. In *EMSOFT '01: Proceedings of the First International Workshop on Embedded Software*, pages 237–253, London, UK, 2001. Springer-Verlag.

[MBLL00] H. Marchand, P. Bournai, Michel Le Borgne, and Paul Le Guernic. Synthesis of discrete-event controllers based on the signal environment. *Discrete Event Dynamic System: Theory and Applications*, 10(4):325–346, October 2000.

[McM93] Ken McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[Mer03] Jason Merrill. GENERIC and GIMPLE: A new tree representation for entire functions. In *GCC Developers Summit*, Ottawa, Canada, May 2003.

[MMMC05a] Matthieu Moy, Florence Maraninchi, and Laurent Maillet-Contoz. LusSy: A toolbox for the analysis of Systems-on-a-Chip at the transactional level. In *International Conference on Application of Concurrency to System Design (ACSD)*, pages 26–35. IEEE Computer Society, 2005.

[MMMC05b] Matthieu Moy, Florence Maraninchi, and Laurent Maillet-Contoz. Pinapa: an extraction tool for SystemC descriptions of systems-on-a-chip. In *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*, pages 317–324, New York, NY, USA, 2005. ACM Press.

[Moy05] Mathieu Moy. *Techniques and Tools for the Verification of Systems-on-a-Chip at the Transaction Level*. PhD thesis, Institut National Polytechnique de Grenoble, Grenoble, France, December 2005.

[Nai02] Ravi Nair. Effect of increasing chip density on the evolution of computer architectures. *IBM Journal of Research and Development*, 46(2-3):223–234, 2002.

[NTLG99] David Nowak, Jean-Pierre Talpin, and Paul Le Guernic. Synchronous structures. In Jos C. M. Baeten and Sjouke Mauw, editors, *CONCUR '99: Concurrency Theory, 10th International Conference, Eindhoven, The Netherlands, August 24-27, 1999, Proceedings*, volume 1664 of *Lecture Notes in Computer Science*, pages 494–509, Eindhoven, The Netherlands, August 1999. Springer.

[Odd01] Denis Oddoux. LTL 2 BA : fast algorithm from LTL to buchi automata.
http://www.liafa.jussieu.fr/~oddoux/ltl2ba/, 2001.

[OMG] OMG. OMG CORBA. http://www.corba.org/.

[OSC] The Open SystemC Initiative OSCI. SystemC Reference Manual. Website: http://www.systemc.org.

[PBMS04] Hiren Patel, David Berner, Deepak Mathaikutty, and Sandeep Shukla. Introspective-SystemC: Reflection and introspection in SystemC. Technical Report 2004-22, Virginia Polytechnic Institute, FERMAT Lab, December 2004.

[Pie05] Emmanuel Pietriga. Zgrviewer - a 2.5D graph visualizer for the DOT language. http://zvtm.sourceforge.net/zgrviewer.html, 2005.

[PMBS06] Hiren D. Patel, Deepak A. Mathaikutty, David Berner, and Sandeep Kumar Shukla. Carh: An introspective and service oriented architecture for validation system level designs. *To be published in IEEE Transactions on CAD (TCAD)*, 2006.

[PSS98] Amir Pnueli, Natarajan Shankar, and Eli Singerman. Fair synchronous transition systems and their liveness proofs. In Anders P. Ravn and Hans Rischel, editors, *FTRTFT*, volume 1486 of *Lecture Notes in Computer Science*, pages 198–209. Springer, 1998.

[RR01] Sriram K. Rajamani and Jakob Rehof. A behavioral module system for the pi-calculus. In *Proceedings Static Analysis Symposium (SAS'01)*, Paris, July 2001. Springer Verlag.

[SASB04] Panagiotis Sfetsos, Lefteris Angelis, Ioannis Stamelos, and Georgios L. Bleris. Evaluating the extreme programming system - an empirical study. In Jutta Eckstein and Hubert Baumeister, editors, *XP*, volume 3092 of *Lecture Notes in Computer Science*, pages 227–230. Springer, 2004.

[SBM+05] Syed Suhaib, David Berner, Deepak Mathaikutty, Jean-Pierre Talpin, and Sandeep Shukla. A functional programming framework for latency insensitive protocol validation. In *Proceedings of the International Workshop on Formal Methods for Globally Asynchronous Locally Synchronous Design (FMGALS)*, Verona, Italy, July 2005.

[SMBS04] Syed Suhaib, Deepak Mathaikutty, David Berner, and Sandeep Shukla. Extreme formal modeling for hardware models. In *Proc. of 5th International Workshop on Microprocessor Test and Verification (MTV'04)*, Austin Texas, USA, September 2004.

[SMBS05] Syed Suhaib, Deepak Mathaikutty, David Berner, and Sandeep Shukla. Validating families of latency insensitive protocols. In *To be published in Proceedings of the IEEE International High Level Design Validation and Test Workshop (HLDVT)*, Napa Valley, California, USA, November 2005.

[SMBS06] Syed Suhaib, Deepak Mathaikutty, David Berner, and Sandeep Shukla. Validating families of latency insensitive protocols. *To be published in IEEE Transactions on Computers (TCOMP), Special Issue on Simulation-Based Validation*, October 2006.

[SMSB05] Syed M. Suhaib, Deepak A. Mathaikutty, Sandeep K. Shukla, and David Berner. Xfm: An incremental methodology for developing formal models. *ACM Transactions on Design Automation of Electronic Systems (TODAES) Special Issue on Validation of Large Systems*, 10(4):589–609, October 2005.

[Sny] Wilson Snyder. SystemPerl - a perl library for SystemC. http://www.veripool.com/systemperl.html.

[SR98] Preston G. Smith and Donald G. Reinertsen. *Developing Products in Half the Time*. John Wiley, New York, 1998.

[ST04] M. Singh and M. Theobald. Generalized latency-insensitive systems for single-clock and multi-clock architectures. In *Design, Automation and Test in Europe (DATE'04)*, 2004.

[Suh04] Syed Suhaib. XFM: An incremental methodology for developing formal models. Master's thesis, Virginia Polytechnic and State University, Blacksburg, Virginia USA, May 2004.

[SVMS96] Alberto L. Sangiovanni-Vincentelli, Patrick C. McGeer, and Alexander Saldanha. Verification of electronic systems. In *DAC '96: Proceedings of the 33rd annual conference on Design automation*, pages 106–111, New York, NY, USA, 1996. ACM Press.

[TBS+04a] Jean-Pierre Talpin, David Berner, Sandeep K. Shukla, Paul Le Guernic, Abdoulaye Gamatié, and Rajesh Gupta. A behavioral type inference system for compositional system-on-chip design. In *International Conference on Application of Concurrency to System Design (ACSD)*, pages 47–56. IEEE Computer Society, 2004.

[TBS+04b] Jean-Pierre Talpin, David Berner, Sandeep Kumar Shukla, Abdoulaye Gamatié, Paul Le Guernic, and Rajesh Gupta. *Formal*

Methods and Models for System Design, chapter Behavioral Type Inference for Compositional System Design. Kluwer Academic Publishers, October 2004.

[TG04] Jean-Pierre Talpin and Paul Le Guernic. *Algebraic Theory for Behavioral Type Inference*, chapter Behavioral Type Inference for Compositional System Design. Kluwer Academic Publishers, October 2004.

[TG05] Jean-Pierre Talpin and Paul Le Guernic. An algebraic theory for behavioral modeling and protocol synthesis in system design. *To appear in Formal Methods in System Design. Special Issue on formal methods for GALS design*, 2005.

[TGB+03] Jean-Pierre Talpin, A. Gamatié, David Berner, Bruno Le Dez, and Paul Le Guernic. Hard real-time implementation of embedded systems in java. In *International Workshop on Scientific Engineering of Distributed JAVA Applications (FIDJI), Lectures Notes in Computer Science*. Springer Verlag, November 2003.

[TGS+03] Jean-Pierre Talpin, Paul Le Guernic, Sandeep K. Shukla, Rajesh K. Gupta, and Frederic Doucet. Polychrony for formal refinement-checking in a system-level design methodology. In *International Conference on Application of Concurrency to System Design (ACSD)*, pages 9–19. IEEE Computer Society, 2003.

[TLS+04] Jean-Pierre Talpin, Paul Le Guernic, Sandeep Kumar Shukla, R. Gupta, and F. Doucet. Formal refinement checking in a system-level design methodology. In *Special Issue of Fundamenta Informaticae on Applications of Concurrency to System Design*. IOS Press, August 2004.

[TM91] D. Thomas and P. Moorby. *The Verilog Hardware Description Language*. Kluwer Academic, 1991.

[VHD] VHDL. VHDL. Website: http://www.vhdl.org/.

[vHT] Dimitri van Heesch and The Doxygen Team. Doxygen, an automated code documentation system. http://www.doxygen.org.

[Wil03] Laurie Williams. The XP programmer - the few minutes programmer. *IEEE Software*, 20(3):16–20, May/June 2003.

[Win90] J. M. Wing. A specifier's introduction to formal methods. *IEEE Computer*, 23(9):8–26, September 1990.

[WK03] William A. Wood and William L. Kleb. Exploring XP for scientific research. *IEEE Software*, 20(3):30–36, May/June 2003.

[YHP+97] Joon-Seo Yim, Yoon-Ho Hwang, Chang-Jae Park, Hoon Choi, Woo-Seung Yang, Hun-Seung Oh, In-Cheol Park, and Chong-Min Kyung. A c-based rtl design verification methodology for complex microprocessor. In *DAC '97: Proceedings of the 34th annual conference on Design automation*, pages 83–88, New York, NY, USA, 1997. ACM Press.

List of Figures

1 Prédictions de la SIA pour le nombre de transistors dans des puces 10

1.1 Flot de conception conjointe 14
1.2 Flot de conception conjointe modulaire 15
1.3 Flot de conception conjointe avec réutilisation 17

4.1 Traduction de modules SystemC en SIGNAL 28
4.2 Méthodologie pour la transformation de SystemC vers SIGNAL 29
4.3 Construction classique de modèles formels 30
4.4 La modélisation formelle itérative 31
4.5 Number of transistors on a chip predicted by the SIA 40

5.1 Simple codesign flow 44
5.2 Modular codesign flow 48
5.3 Codesign flow with IP integration 50

8.1 Specification of a counter modulo 2 64
8.2 Formal syntax of iSTS algebra 67
8.3 A behavior in the polychronous model of computation 68
8.4 Scheduling relations between simultaneous events 69
8.5 Relating synchronous behaviors by stretching. 70
8.6 Denotational semantics of clock expressions 71
8.7 Denotational semantics of propositions 72
8.8 Denotational semantics of propositions 72
8.9 Clock inference system 74
8.10 Polychronous specification of a buffer 75
8.11 Clock analysis of the buffer 76
8.12 Buffer code generation 76
8.13 Signal syntax core 77
8.14 From Signal to iSTS $[\![P]\!]$ 78
8.15 From iSTS to Signal $\lfloor P \rfloor$ 78

9.1 Translation of a source program into static single assignment form 81

9.2 Propositional behavior of the SSA program 82
9.3 Static abstraction of the behavioral type 84
9.4 Abstract syntax for SystemC 86
9.5 Abstract syntax for SystemC programs in SSA form 86
9.6 Ones-counter method in SystemC 87
9.7 Ones-counter method in SSA 88
9.8 Type inference rules 89
9.9 Type of the wait-notify protocol 91
9.10 Modular extension of the inference function to separate methods 92
9.11 Abstraction of the behavioral type of the while loop by a static interface 93
9.12 Behavioral types for modules 93
9.13 Type inference for declarations 94
9.14 Type inference for modules 95
9.15 Embedding the intermediate representation in Signal 96
9.16 Signal type of the ones counter 97

11.1 Translation of SystemC modules into SIGNAL 109
11.2 Methodology for translating SystemC models into SIGNAL . . 110

12.1 Two connected components 117
12.2 Structure of the FIR filter with testbench 120
12.3 SystemC and SSA code for the FIR core 122
12.4 Clock and scheduling relations for the FIR 123
12.5 Control flow of the FIR filter 124
12.6 Data Flow of the FIR 124
12.7 Block view of the SIGNAL type for the FIR filter 125
12.8 Example of a formal property definition 128
12.9 Verification of Properties using Sigali 128
12.10 Abstract SIGNAL model 130

13.1 Design Flow for the extraction 133
13.2 Examples of class declarations 133
13.3 Doxygen XML Representation 134
13.4 Main entities of the DTD 136
13.5 Class diagram showing data structure 137
13.6 Example DOT code and resulting graph 139
13.7 Visualization of the extracted module hierarchy of a USB controller 139
13.8 Interface of the module usb_crc5 from the USB example 142
13.9 Testbench snippet in *unconstRand* mode for a pre-specified port 142
13.10 Testbench snippet in *simpleRand* mode for a pre-specified port 143
13.11 Testbench snippet in *distRand* mode for a pre-specified port . . 143

13.12 Snippet of the testbench showing the trace file creation 144

14.1 State of the art to capture a formal model 148
14.2 Capturing a formal model with XFM 149
14.3 Modeling process (a) and modeling result (b) 150
14.4 LTL operators 154
14.5 LTL properties for traffic light 155
14.6 FSMs of traffic properties 1 (a), 2 (b), 3 (b) 155
14.7 Graph for traffic property 5 156
14.8 Promela code for pedestrian crossing 157
14.9 Cycles for different instruction types 157
14.10 PROMELA code for one single instruction 158
14.11 Graphs of pipeline properties 1 (a), 3 (b), and 4 (c) 158
14.12 Example LTL properties for the DLX pipeline 159
14.13 Pipeline automaton for one instruction 160
14.14 Structure of the DLX pipeline 161

15.1 Equalizer Example 169
15.2 LI system proposed by Carloni 170
15.3 LI system without relay stations 171
15.4 LI implementation with different clocks 173
15.5 Verification 174

Résumé

La modélisation du contenu des puces électroniques s'avère de plus en plus difficile puisque le développement des outils et des méthodologies de modélisation n'a pas su accompagner l'explosion de la complexité des systèmes. Les méthodes formelles ont su démontrer dans les dernières années leurs capacités de prévention et de détection d'erreurs durant les phases de modélisation. Malgré cela, leur utilisation reste toujours restreinte à des domaines bien spécifiques comme le militaire ou l'avionique en raison d'un manque de liens avec les méthodes existantes et la difficulté de leur utilisation.

Dans ce document, nous essayons de montrer dans quelle mesure la conception de systèmes embarqués peut profiter de l'utilisation de méthodes formelles. Pour cela, nous proposons plusieurs approches qui démontrent comment l'utilisation des méthodes formelles peut être intégrée dans la conception conjointe tout en cachant - du moins en partie - leur complexité inhérente. Plus spécifiquement nous définissons une méthodologie pour extraire des modèles formels à partir de modèles non formels de systèmes embarqués. Une autre approche proposée utilise les méthodes agiles dès le début du flot de conception pour faciliter la construction de modèles formels à partir de spécifications. Cette méthodologie de construction incrémentale produit des modèles formels correctes par construction avec les propriétés formelles correspondantes.

Abstract

Designing electronic chips is becoming increasingly difficult as the modeling tools and methodologies cannot keep up with the rise in system complexity. In the recent past, formal methods have proved their capability of error detection and prevention in the different phases of system modeling. However, due to their complexity and a lack of integration with existing design methods, their use is still restricted to specific domains such as avionics and military.

In this document we show how the design of embedded systems can benefit of the use of formal methods. We give several examples on how formal methods can be integrated in a co-design flow while hiding - at least to some extent - their inherent complexity. More specifically we define a methodology to extract formal models from non formal embedded system descriptions. Our approach separates the extraction of the structure from the transformation of the behavior. Another approach that we propose aims at the use of formal models early in the design flow. It uses agile methods to facilitate the construction of formal models from natural language specifications. The presented methodology for the incremental building of formal models produces correct-by-construction models along with the corresponding formal properties.