Université de Montréal

De la fusion du génie logiciel et d'une bibliothèque à source ouverte pour la modélisation/simulation de processus matériel et logiciel

par

Luc Charest

Département d'informatique et recherche opérationnelle

Faculté des arts et des sciences

Thèse présentée à la Faculté des études supérieures

en vue de l'obtention du grade de Philosophiæ Doctor (Ph. D.)

en Informatique

Septembre, 2004

Université de Montréal

Faculté des études supérieures

cette Thèse intitulée:

De la fusion du génie logiciel et d'une bibliothèque à source ouverte pour la modélisation/simulation de processus matériel et logiciel

présentée par :

Luc Charest

a été évaluée par un jury composé des personnes suivantes :

Jean-Pierre David
président-rapporteur

El Mostapha Aboulhamid
directeur de recherche

Guy Bois
codirecteur

Yvon Savaria
membre du jury

Résumé

Avec la complexité des systèmes et le coût de fabrication des puces électroniques sans cesse croissants, avec les délais de recherche et de conception de plus en plus courts, l'industrie se doit d'encourager la recherche des méthodologies de conception de systèmes qui, tout en validant l'intuition des ingénieurs, assurent que le prototype construit fonctionnera en respectant les spécifications prévues.

En septembre 1999, SystemC, une nouvelle bibliothèque de simulation logiciel/matériel basée sur le langage C++ vit le jour, ouvrant ainsi la porte à une possibilité d'amélioration des techniques du codesign. Le but de ce travail est donc d'étudier ce nouveau venu, délimiter sa place dans le domaine du codesign et, finalement, déterminer l'amplitude de la vague de nouveaux concepts qu'il apporte avec lui.

Nos objectifs sont clairs ; établir la place de l'approche orientée objet et des motifs de conception dans le domaine de la conception matérielle, analyser SystemC et, enfin, proposer des modifications qui augmenteront, soit la méthodologie, soit la bibliothèque ou encore les deux.

Cet ouvrage inclut également des concepts qui furent présentés lors de conférences d'envergure internationale. Ces concepts sont bien établis dans le domaine du génie logiciel, mais s'avèrent être novateurs en rapport à la conception matérielle, et nos articles furent cités par un bon nombre d'articles par la suite [1, 2, 3, 4, 5, 6].

Mots clés : conception simultanée, génie matériel, génie logiciel, simulation, programmation orienté objet, motifs de conception, introspection, interopérabilité, SystemC, VHDL

Abstract

With the system complexity and the manufacturing cost of the electronic chips unceasingly increasing, with the shrinking of research and designs delays, the industry must support the research of system design methodologies which, while validating the intuition of the engineers, ensures that the built prototype will function according to the original specifications.

In September 1999, SystemC, a new software/hardware simulation library based on the C++ language was born, thus opening possibilities of improving the codesign techniques.

The goal of this work is to study this new environment, to delimit its place in the codesign domain and, finally, to determine the significance of the new concepts it brings.

Our objectives are clear; to establish the place of the object oriented approach and the design patterns in the field of hardware design, to analyze SystemC and, finally, to propose modifications which will enhance, either the methodology, the library or both.

This thesis also includes concepts which were presented in conferences of international scale. These concepts are well established in the field of software engineering, but prove to be significant innovations to hardware design, and our articles were referenced by several articles [1, 2, 3, 4, 5, 6].

Keywords: codesign, hardware engineering, software engineering, simulation, object oriented programming, design patterns, introspection, interoperability, SystemC, VHDL

Table des matières

Table des figures

Liste des tableaux

Liste des sigles et des abrviations

ADT ..... Abstract Data Type

ANSI ..... American National Standards Institute

ASIC ..... Application Specific Integrated Circuits

CASHE ..... Computer-Aided Software-Hardware Engineering

CORBA ..... Common Object Request Brokerage Architecture

cpp ..... C Preprocessor

DATE ..... Design Automation and Test in Europe

DCOM ..... Distributed Component Object Model

DDE ..... Dynamic Data Exchange

DSP ..... Digital Signal Processing

DUT ..... Design Under Test

EDA ..... Electronic design automation

EX ..... Execution

FSM ..... Finite State Machine

GDB ..... GNU DeBugger

GUI ..... Graphical User Interface

HDL ..... Hardware Description Language

HW/SW ..... HardWare/SoftWare

IDL ..... Interface Definition Language

ID ..... Instruction Decode

IF ..... Instruction Fetch

LRM ..... Library Reference Manual

MEM ..... Memory

OSCI ..... Open SystemC Initiative
PLI ..... Programming Interface Language
POO ..... Programmation Orientée Objet
ROM ..... Read Only Memory
RTL ..... Register Transfer Level
RTOS ..... Real-Time OS
RTTI ..... Run Time Type Information
SCV ..... SystemC Verification Library
sed ..... Stream EDitor
SIDL ..... SystemC IDL
SOC ..... System-On-a-Chip
STL ..... Standard Template Library
TDM ..... Time Division Multiplexer
UML-RT ..... UML Real Time
UML ..... Unified Modeling Language
VCs ..... virtual components
VHDL ..... VHSIC HDL
VHSIC ..... Very High Speed Integrated Circuit
WB ..... Write Back
XMI ..... XML Metadata Interchage
XML ..... eXtensible Markup Language

dédicace

Je dédie cet ouvrage à mes parents, à ma famille et enfin à Dieu, seulement, toutefois, si ce dernier existe, ou encore, advenant le cas contraire, au destin, à la chance, au hasard, à l'ironie, au principe de la sélection naturelle qui, ensemble ou séparément, ont su si bien faire les choses...

Remerciements

J'aimerais en premier lieu remercier mes parents pour d'abord avoir eu l'idée, qui fut peut-être regrettée par la suite, de me mettre au monde, et pour m'avoir, ensuite, enduré tout ce temps. J'aimerais remercier tous les scientifiques qui m'ont précédé d'avoir fait le plus gros du travail et enfin, Douglas Adams, qui a su apporter une réponse à la question sur la vie, l'univers et tout le reste[1].

Chapitre 1

Introduction

Avec la complexité des systèmes et le coût de fabrication des puces électroniques sans cesse croissants, avec les délais de recherche et de conception de plus en plus courts, l'industrie se doit d'encourager la recherche des méthodologies de conception de systèmes qui, tout en validant l'intuition des ingénieurs, assurent qu'un prototype construit fonctionnera en respectant les spécifications prévues.

Établis depuis bon nombre d'années, VHDL (VHSIC (Very High Speed Integrated Circuit) HDL (Hardware Description Language)) et Verilog dominaient le monde du génie matériel en offrant la possibilité aux ingénieurs de valider leur modèle avant la mise en œuvre finale.

1.1 Méthode de conception

Pour construire un système électronique, la manière habituelle de procéder est assez simple et consiste à d'abord établir les besoins sous forme généralement de do-

que l'on voudra valider en symbiose avec le système pour s'assurer non seulement de la validité des spécifications des propriétés intellectuelles de ces sources externes, mais également de la validité des spécifications du système que l'on désire construire. On peut également vouloir valider l'application (la partie logicielle) qui s'exécutera sur le système en question.

En guise d'exemple, considérons le cas d'un système embarqué tel un téléphone mobile. Le téléphone, dans notre exemple, constitue l'élément électronique à construire que les ingénieurs voudraient valider avant la production du prototype. Prenons le cas où le processeur central de ce téléphone serait fourni, par une compagnie X, donc déjà disponible sur le marché (le reste de la circuiterie devant être conçu localement). L'application est en fait le bottin téléphonique, l'agenda électronique ou encore la partie du logiciel qui établit la connexion avec le réseau. Le processeur est, dans notre exemple, la propriété intellectuelle dont le fonctionnement peut être tenu pour acquis ; il n'a donc pas à être validé, mais par contre, son interaction avec le reste du système est très importante et doit, par conséquent, être validée.

Le coût lié à la fabrication d'un tel prototype est souvent énorme puisqu'il faut tout prévoir et il n'y a aucune garantie que le tout fonctionnera en fin de compte si rien n'est testé en cours de route. Le cas du téléphone cellulaire n'est peut-être pas approprié pour faire valoir cet argument, mais il s'applique aux processeurs tels Pentium, Duron et autres, où le coût de fabrication de la galette de silicium est gigantesque étant donné les technologies de fabrication de plus en plus microscopiques et onéreuses.

La suite de la méthodologie consiste à créer un modèle fonctionnel qui, comme son nom l'indique, modélise la fonctionnalité du système à construire. Ce modèle peut servir à la fois à valider les grandes lignes directrices du document de spécifications

par exemple, vouloir vérifier avec un langage comme Prolog qu'une propriété logique particulière est respectée (que le système n'a pas d'interblocage par exemple).

Une fois le modèle fonctionnel validé, VHDL ou Verilog entrent alors en jeu pour se rapprocher de la couche matérielle. Des ingénieurs matériel doivent donc souvent traduire manuellement le modèle fonctionnel, qui souvent devient alors une référence au même titre que le document de spécifications original, en un modèle synthétisable, qui lui pourra être traduit par les outils de synthèse directement en un schéma matériel constitué de portes logiques ou autres composants électroniques de base. Cette traduction de code source au matériel se fait de manière automatique par une première étape de transformation, puis une seconde étape d'optimisation par des algorithmes bien connus et bien établis dans le domaine. Le programme à traduire doit cependant respecter certaines règles de construction, sans quoi, tout fragment de code dérogeant de ces règles ne pourra être synthétisé sans qu'un être humain ne réécrive le fragment en question.

1.2 Approche ouverte et orientée objet

Le consortium OSCI [7] développa SystemC qui fût introduit en septembre 1999 avec un avenir déjà prometteur, devenant rapidement de plus en plus populaire. Basé sur le langage C++, encourageant donc de facto la philosophie orientée objet, parfois présenté comme un langage, SystemC n'est en fait qu'un ensemble de classes qui permettent la gestion aisée du parallélisme, tout ceci évidemment, afin de mieux modéliser le comportement matériel en mettant à la disposition de l'utilisateur un éventail de classes qui servent des blocs de base pour la construction du modèle. Sans cesse en

offre, avant tout pour la simulation, la possibilité d'utiliser la philosophie orientée objet ; soit pour accélérer le temps de conception, soit pour créer des modèles plus réutilisables et flexibles, ou encore pour améliorer la lisibilité ou augmenter la localité des données et fonctionnalités des composants du système. Un modèle fonctionnel créé en langage C peut plus aisément être converti en langage C++ (SystemC), car le code procédural peut être recyclé souvent tel quel. Évidemment, pour faire un bon travail, un ingénieur devrait idéalement remodeler les constructions C en de plus élégantes structures orientées objet, ce qui est loin d'être une tâche triviale, ou du moins, facilement automatisable.

Le fait que SystemC offre la possibilité de modéliser à un niveau d'abstraction supérieur à VHDL apporte un second avantage. Par exemple ; on peut utiliser des appels de méthodes pour modéliser des transactions entre un processeur et une mémoire sans avoir à spécifier le détail de la communication et le protocole utilisé comme on se doit de le faire au niveau RTL (Register Transfer Level). SystemC encourage donc l'élaboration de modèles à hauts niveaux d'abstraction, soit le niveau « transactionnel »[1] ou tout simplement « fonctionnel ». Lorsque le modèle fonctionnel est validé, il peut s'avérer être une tâche facile que de le transformer en un modèle RTL, puisqu'on utilise le même environnement, la même bibliothèque de programmation, mais à des niveaux d'abstraction différents alors que comparativement, la méthodologie antérieure requérait le passage du C au VHDL qui sont deux langages considérablement différents. Ceci constitue donc un troisième avantage d'utiliser SystemC ; il unifie le flot de la méthodologie du codesign étant donné qu'il est maintenant possible d'utiliser un seul langage, tout au long du processus. N'étant pas une transformation triviale à réaliser, un ingénieur consciencieux pourrait toutefois construire son modèle à haut niveau en

À son apparition en 1999, pour l'ensemble de ces avantages, nous avons choisi SystemC parce qu'il représente à nos yeux le meilleur compromis du domaine du codesign. La meilleure comparaison que nous puissions effectuer est par rapport à un langage existant et bien établi dans le domaine et qui permette à la fois la simulation et la synthèse. Bien que Verilog ait pu être un bon candidat, il est semblable en bien des points à VHDL, le langage que nous avons choisi comme point d'origine de comparaison. Une autre raison justifiant le choix est le niveau d'expertise qui était disponible dans le laboratoire quant à ce langage de modélisation matériel.

Java étant interprété, il fût éliminé ; il s'exécute généralement très lentement (bien qu'il existe aujourd'hui plusieurs technique novatrices pour améliorer les performances comme la compilation dynamique). Bien que SystemC ne soit pas synthétisable à cent pour cent, il n'en demeure pas moins qu'un sous-ensemble, au niveau d'abstraction RTL, l'est. Le projet ODETTE (Object-Oriented co-DEsign and functional Test TEchniques) [9] a pour but de construire un engin pouvant synthétiser des constructions orientées objet dérivées de SystemC. Sur leur site Web, seule leur bibliothèque est disponible, leur outil de synthèse n'étant apparemment qu'un prototype non disponible au public. Java possède également quelques lacunes relatives à des concepts précieux à l'architecte matériel, dont la surcharge des opérateurs, qui est disponible même en VHDL, mais inexistante dans le monde de Java. Un autre point important justifiant notre choix de ne pas utiliser Java était de pouvoir travailler avec la communauté et non pas en vase clos, avec cet intérêt pour SystemC grandissant de jour en jour.

Peut-être existe-t-il d'autres langages qui siéraient mieux à la description du codesign parmi l'impressionnante liste des quelques 80 catégories de langages de pro-

1.3 Motifs de conception

La manière dont l'approche orientée objet est montrée et enseignée utilise souvent des exemples « jouets » pour démontrer avec aise des concepts. Évidemment, le but de l'exemple est de démontrer des concepts comme l'héritage et non de se perdre dans maints détails d'implémentation, mais savoir comment bien construire des systèmes basés sur la philosophie orientée objet requiert souvent beaucoup d'expérience professionnelle.

Tentant de répondre à ce problème, provenant du génie logiciel, introduit par ceux qui sont maintenant appelés « le groupe des quatre » [10], les motifs de conception forment, ensemble, un outil très puissant pour tout architecte logiciel qui désire hausser son art à des niveaux de normes plus élevées.

Un motif de conception se présente sous forme d'un document explicitant une solution à un problème lié à la méthodologie orientée objet. Un motif de conception est constitué en partie d'une définition du problème par le biais d'un court texte, de quelques schémas illustrant la solution, d'une liste dressant le pour et le contre de la méthode, d'exemples avec code, de l'utilisation historique du motif, bref, tout ce qu'il faut pour faire le tour complet du sujet.

Bien que le livre de Gamma et al. constitue, en soi, un premier catalogue de motifs et une source de références primaires pour un architecte, il est important de percevoir cet ouvrage d'abord, et avant tout, comme un tutoriel à une bonne modélisation orientée objet (et non comme une source absolue de références) ; et par la suite, par l'extension de l'expérience acquise, comme une ligne de conduite pour le futur.

De chercher à faire la liste exhaustive ou non de l'applicabilité des premiers motifs, publiés par les quatre, au domaine du génie matériel ne serait pas utile pour plusieurs

une base à certains concepts importants. Les auteurs ont cependant amené l'idée que la liste, n'étant pas complète, et en émettant l'hypothèse qu'elle ne le sera probablement jamais, ne pourrait qu'être qu'un peu « plus » complète si on lui ajoute d'autres motifs. Une liste des motifs qui s'appliquent au design matériel ne serait donc pas exhaustive.

- Définissant plus une idée, une solution à un problème donné, et en partant du principe qu'il est pratiquement impossible de prévoir tous les cas de problèmes auxquels le monde du matériel pourrait faire face, il devient impossible de prétendre qu'un motif donné, en particulier, ne s'appliquera jamais à un domaine précis.
- Étant donné le fait qu'on arrive rarement dans une situation parfaite telle que décrite par les motifs, le motif en tant que tel sert souvent de point de départ pour une combinaison de différents motifs pour parvenir à nos fins. Les motifs se doivent d'être plus une philosophie qu'un guide complet.

Il faut donc regarder les motifs comme étant une collection de principes de base plutôt qu'un ensemble de solutions préconçues et prêtes à être utilisées.

1.4 La synthèse et SystemC

Bien que SystemC encourage une approche orientée objet pour la simulation, à l'heure actuelle, la synthèse requiert le niveau d'abstraction RTL qui est bien spécifique et se trouve très près de la description matérielle pure. Héritage[2], polymorphisme, surcharge d'opérateurs, appels récursifs, templates[3] ; bref, une grande partie de ces outils hérités du C++ et disponibles au moment de la simulation, se retrouvent soit limités

détail dans [11], à l'heure actuelle, il est pratiquement impossible d'utiliser la grande majorité des solutions apportées par les motifs de modélisation qui requièrent ces paradigmes de programmation orientée objet.

Les motifs de conception, étant des solutions cataloguées et documentées à des problèmes existants et concrets, s'avèrent être très pratiques et il serait absurde de s'en passer lorsque vient le temps d'effectuer la modélisation de haut niveau, qui se doit d'être le plus rapide que possible pour accélérer la mise en marché du produit, sous simple prétexte qu'ils deviennent un blocus infranchissable, le temps de la synthèse venu.

L'idéal serait d'avoir une synthèse qui permet toute construction, quelle qu'en soit la forme. À notre avis, ce n'est qu'une question de temps avant que les outils de synthèse ne permettent des éléments plus complexes tels ceux de la philosophie objet. Cette opinion est appuyée par plusieurs faits :

- la bibliothèque SystemC est relativement récente (1999),
- il y a eu très peu de tentatives pour amener l'approche orientée objet au design matériel [12],
- les outils permettant la synthèse pour SystemC n'ont pas encore atteint leur pleine capacité,
- le principe de synthèse de matériel et ses algorithmes existent depuis longtemps.

Étant donné que ces algorithmes existent depuis bon nombre d'années, les gens du domaine n'auraient pas eu le temps de les adapter aux nouvelles technologies. Un exemple flagrant est l'héritage : se basant du simple fait qu'on est capable de synthétiser un module, pourquoi n'est-on pas capable de synthétiser son descendant hiérarchique qui en hérite immédiatement ? A priori, il n'y a pas d'explication logique qui empêche l'apla-

Les articles présentés dans ce rapport furent rédigés dans l'optique d'une amélioration du modèle fonctionnel lié à la simulation et non dans l'optique de l'amélioration du modèle lié à la synthèse. Bien souvent, le thème de la synthèse, bien qu'il soit vital à la méthodologie et au domaine n'y est pas abordé. Globalement, les articles tentent de faire passer le message suivant : « L'approche des motifs de conception ainsi que, par conséquent, l'approche orientée objet sont toutes deux utiles, même dans un domaine de modélisation matériel ».

1.5 Modélisation UML

Depuis le tout début de l'ère orientée objet, les ingénieurs ont créé des schémas représentant leur système. Il n'y avait pas de consensus jusqu'à la venue d'UML (Unified Modeling Language) [13]. La méthode de modélisation UML, ayant été développée par des gens du génie logiciel, n'est pas bien définie pour la modélisation matérielle. Si on désire modéliser le système à très bas niveau, comment peut-on, par exemple, exprimer une connexion par câblage entre deux modules matériels ? Les diagrammes UML ont été conçus en tout premier lieu pour représenter des structures de classes, des appels de méthodes, des propriétés liées à des objets, des séquences d'appels, bref, représenter un programme intangible et non du matériel concret. Une représentation par le biais du diagramme Objet d'UML (Unified Modeling Language) est toujours possible, mais peut-être pas tout à fait souhaitable. Par exemple on peut utiliser une icône de boîte pour représenter un signal et utiliser les liens de relations entre les instances d'objets afin de décrire les connexions entre les signaux et les modules. Cette manière de faire est cependant très lourde graphiquement. Une extension d'UML –

1.6 Introspection

L'introspection, par rapport à un langage, un programme ou une bibliothèque de programmation est de fournir de l'information sur sa structure, ses données. Voici une liste non exhaustive des données que l'on pourrait souhaiter récupérer avec l'introspection de SystemC :

Information structurelle : le nom, la topologie et hiérarchie des modules ; le nom, les connexions entre les signaux et leurs modules associés...

Information de données : le type et la valeur des signaux, ports, canaux, structures personnalisées et événements...

Autres informations : les méthodes (processus) des modules, leur nom, paramètres et contenu (description du comportement) ; la sémantique de plus haut niveau, la nature du composant matériel (bus, processeur, registre...) ou encore du composant logiciel (interface, récursive, RTOS)...

La capacité de réflexion (ou instrospection) des cadres d'applications modernes (Java et .Net) est très intéressante et nous permet d'obtenir des détails aussi précis que les paramètres d'une méthode d'une classe donnée. SystemC étant avant tout basé sur le langage C++, qui est un langage offrant, a priori, peu de mécanismes de réflexion, n'offre malheureusement que peu de ressources. Nous pouvons scinder l'introspection (tout comme l'interopérabilité) en deux classes : intraprocessus et interprocessus.

La première est la plus simple à réaliser, car elle se fera habituellement avec le même langage ; aucun besoin de convertir les données sous un autre format, les processus communicants sont alors compilés sous le même exécutable et ont souvent un couplage assez serré. La deuxième classe, interprocessus, se fera au-dessus d'un moyen

interopérable.

1.7 L'interopérabilité

L'interopérabilité peut être considérée comme étant une métrique qui mesure l'aisance avec laquelle deux programmes (processus) peuvent s'exécuter de manière concurrente, communiquer et coopérer ensemble.

Quelques cas d'utilisation où l'interopérabilité de SystemC pourrait être mise à contribution :

Accéder à une bibliothèque écrite dans un autre langage, effectuer une cosimulation avec un autre modèle SystemC ou avec un modèle écrit dans un autre langage, afficher les résultats de la simulation vers une sortie non usuelle, faire des analyses sur les résultats avec des outils externes...

Pour le cas qui nous intéresse, nous pouvons définir a priori les éléments qui pourraient aider l'interopérabilité de SystemC avec lui-même ou encore avec des outils de tierces parties :

Contrôle de la simulation : démarrer, arrêter, suspendre, remettre à zéro la simulation...

Contrôle de débogage : tracer X, démarrer la trace, arrêter la trace...

1.8 État de l'art d'un point de vue historique

L'apparition des premiers langages orientés objet se réalisa vers les années 1970-1980 [13].

Verilog-XL [16], soit PLI (Programming Language Interface), en y intégrant de nouvelles fonctionnalités d'année en année.

L'année 1990 marque le début du projet « Ptolemy » [17], un environnement de simulation basé sur le langage C++ pour différents types de MoC (Models of Computation) qui seront développés au cours des années subséquentes. Ces représentations capturent la sémantique du modèle sous différents points de vues souvent orthogonaus entre eux.

L'organisation OVI (Open Verilog International) est formée pour gérer l'évolution du langage Verilog qui est maintenant du domaine public. L'interface devient norme avec PLI 1.0.

En 1991, CORBA (Common Object Request Broker Architecture) 1.0 voit le jour [18]. Ce système permet une interopérabilité la plus totale, entre tout langage qui possède une mise en correspondance IDL (Interface Definition Language) bien définie et qui implémente la partie ORB (Object Request Broker), soit l'interface de la norme. La première version de CORBA ne souscrivait qu'au langage C.

En 1993, OVI lance la version 2.0 du PLI, tout en demandant la normalisation de Verilog et de son interface auprès d'IEEE.

Bien qu'il y ait eu de nombreuses méthodes de représentation des structures orientées objet, UML, la plus reconnue maintenant, prit naissance en octobre 1994 pour n'être dévoilée que l'année subséquente.

En cette même année 1994 qui a été fructueuse pour le génie logiciel, Gamma et al. [10] introduisent les motifs de conception. Ce livre est très bien construit et est devenu source de référence du domaine. Les motifs sont répertoriés et classés avec des descriptions exhaustives, allant même jusqu'à décrire les avantages et les inconvénients

elle est scindée en trois parties : les « routines TF », les « routines ACC » et les « routines VPI ». Ces routines sont à deux sens et elles permettent non seulement l'introspection de valeurs en lecture et en écriture, mais également la possibilité de se synchroniser avec le simulateur (pour la cosimulation).

Swamy et al. [12] tentèrent d'apporter au domaine une vision orientée objet qui passa plus ou moins inaperçue, tout en citant d'autres travaux déjà effectués dans le domaine sur le sujet qu'est la POO (Programmation Orientée Objet) pour VHDL. L'article mentionnant le fait qu'un groupe de normalisation existe depuis 1993, la norme orientée objet pour VHDL semble alors utopique et même encore aujourd'hui, soit précisément une décennie plus tard, qu'il y ait eu, ou non, aboutissement à une norme OO-VHDL (Object-Oriented VHDL), on commence à peine à voir apparaître de nos jours des modèles orientés objet alors que les ingénieurs en matériel utilisent toujours des constructions procédurales de bas niveau, que ce soit en VHDL ou en SystemC. Bien qu'il apporte une vision orientée objet, un des schémas de l'article [12] dénote une méthodologie plutôt éclatée en plusieurs outils.

En 1996, la nouvelle version 2.0 de CORBA est lancée, incluant une mise en correspondance avec le langage C++ et Smalltalk.

Février 1996 marque le début de SWIG (Simplified Wrapper and Interface Generator) [19] qui, à l'origine, était un outil permettant d'étendre un langage de script personnalisé créé pour la célèbre « Connection Machine 5 ». Il deviendra plus tard un logiciel qui permet de créer de manière automatique des entités englobantes des fonctionnalités ou des programmes écrits en langage C/C++, et ce, pour différents langages cibles. Nous avons mis à l'épreuve ce logiciel pour tenter d'interconnecter SystemC et notre Système, ESys.Net, écrit en C#, mais ce fut, hélas, sans succès.

complètement dénudé de toute référence quant aux possibilités de modélisation orientée objet. Alors qu'on s'attend à y retrouver une description exhaustive de la structure interne, puisque SystemC est à code source ouvert, la description des fonctionnalités y est sommaire ; le style d'écriture et les exemples utilisent des macros cachant les implémentations orientées objet et contribuent à trahir certaines motivations des concepteurs : on chercher à convertir les ingénieurs, étant alors à l'aise avec VHDL, vers un nouveau langage et non vers une nouvelle méthodologie. Il faudra attendre la version 2.0 de SystemC avant de voir apparaître dans le manuel de spécification fonctionnelle [20] une première mention sur la façon de créer un module sans utiliser ces macros provenant de la bonne, mais vétuste méthode du langage C. Cette version utilise toujours et sans retenue des macros, et bien que le concept de canaux de communication abstraits amène le principe d'utilisation de classes d'interface, le concept général de programmation orientée objet y est toujours mal défini, voire même absent.

SystemC permet une introspection plutôt limitée, car elle passe par des chaînes de caractères pour décrire la hiérarchie des modules, ce qui la rend orientée vers l'introspection interprocessus et rien ne définit la façon d'obtenir l'information sur la connectivité du modèle (relations entre modules, ports et signaux).

En juin 1999, CORBA, qui en est à sa version 2.3, inclut maintenant une mise en correspondance avec Java.

La parution de l'article du chapitre 4 fut très bien accueillie à DATE (Design Automation and Test in Europe) en mars 2001. Elle fut perçue comme novatrice pour le domaine. Les gens qui y assistèrent y gagnaient doublement ; non seulement, certains entraient en contact avec une nouvelle bibliothèque de programmation pour la simulation matérielle basée sur le C++ et d'autres avec un aspect du génie logiciel,

En octobre 2001, un article [22] parut dans ISSS à Montréal portant la même idée que de tenter l'application des motifs de conception au domaine du design matériel. À cette même conférence, Doucet, Shukla et Gupta y ont présenté un système permettant l'introspection et l'instanciation des différentes parties du modèle à travers un langage de script [23, 24, 25]. Les composants de base écrits en langage C sont enveloppés par une interface semblable à celle de CORBA. Leur approche diffère grandement de celle abordée par nos articles du fait qu'elle consiste à amener l'introspection par le biais de la création d'un méta langage propriétaire. Un compilateur spécialisé et adapté au nouveau langage est alors requis et doit être construit, ce qui rend l'approche hors norme si le langage n'est pas soumis à un organisme de normalisation et rend les choses un peu plus complexes si les outils ne sont pas mis à la disposition du public de l'industrie. Heureusement, une version de Balboa est disponible sur leur site [26] ; malheureusement, celle-ci se rapporte à la version 1.0.2 de SystemC.

En mars 2002, Green et Edwards [27] présentent une méthodologie pour la modélisation et le codesign en utilisant UML-RT (UML Real Time) qui permet l'expression des protocoles. Une partie également intéressante de leur approche est de découper la méthodologie en partie « uncommited » (non fixé) et « commited » (fixé).

En juin 2002, emboîtant fort probablement le pas à SystemC, SystemVerilog 3.0 [14] fait son apparition. Cette extension de Verilog, qui n'est pas encore normalisée, promet d'intégrer C et quelques aspects orientés objet à Verilog-2001.

En novembre 2002, la première version de la bibliothèque SCV (SystemC Verification Libray) est rendue publique par OSCI. La bibliothèque permet surtout d'établir une interopérabilité entre SystemC et TestBuilder [28], un outil à code source ouvert qui permet de construire des scénarios de test. La façon de réaliser l'introspection

STMicroelectronics utilise un principe similaire à CORBA pour leur plateforme StepNP [29].

En 2003 OSCI lance la version 1.0 de la bibliothèque SCV.

Vers la fin de 2003, Lapalme, aidé de l'expérience du laboratoire avec les simulateurs à événement discrets, construit un système HDL basé sur le langage C# et le cadre d'applications .Net de Microsoft. L'introspection, implicite à .Net, est bien intégrée avec le mécanisme d'attributs. Comparativement à SystemC, le simulateur, bien que complet et fonctionnel, est assez simple dû au fait que les mécanismes de fils sont déjà présents dans le cadre d'applications .Net et ont pu être utilisés tels quels. Contrairement à SystemC, ESys.Net respecte la méthodologie orientée objet et le modèle de l'utilisateur se trouve très bien isolé du noyau, ce qui permet au simulateur de récupérer, complètement par introspection, l'information structurelle du modèle. La sémantique du niveau supérieur (la nature des composants) reste une fonctionnalité à développer.

À l'été 2004, notre article sur l'unification des types et l'introspection parait à IWSOC 2004 [30].

En résumé, beaucoup de langages de modélisation de matériel ont vu le jour et tenter de les unifier semble une utopie ; même si on réussit ce tour de force qui serait de trouver une sémantique ralliant tous les différents langages, les compagnies de matériels ont toujours éprouvé des difficultés à complètement abandonner leurs anciens modèles. Souvent, leurs modèles précédents sont transformés en versions améliorées pour constituer leurs modèles suivants ; abandonner des modèles écrits dans un langage est l'équivalent en une perte nette d'argent et de main-d'œuvre.

Améliorer la méthodologie de conception :

- Apporter au domaine du génie matériel une partie de la vision du génie logiciel.
- Présenter des solutions orientées objet qui facilitent la flexibilité et la réutilisation des modèles des utilisateurs.
- Introduire les motifs de conceptions qui proviennent du domaine du génie logiciel et qui apportent des esquisses de solutions basées sur l'expérience.
- Encourager le passage du niveau de modélisation RTL à un plus haut niveau d'abstraction (comme le niveau transactionnel) dans le but de réduire le temps de réalisation du modèle, et ce, afin de valider plus rapidement les premières assertions des spécifications.
- Analyser l'impact des choix de méthodologies de conception par rapport à la synthèse.

Valider le choix d'utilisation de SystemC :

- Analyser le code source du noyau de l'engin de la bibliothèque.
- Vérifier la performance.
- Tester la variabilité dimensionnelle.
- S'assurer du bon fonctionnement du système et de la stabilité du produit.

Améliorer et modifier SystemC :

- Moderniser la bibliothèque dans le but de faciliter la méthodologie de développement.
- Revoir la structure orientée objet de la bibliothèque.
- Parfaire les mécanismes d'introspections et d'interopérabilité de la bibliothèque.

...et ce, tout mettant en œuvre **quelques principes de base** (sauf lorsque cela nuit à l'avancement et l'évolution de SystemC ou de la méthodologie) :

de préserver la rétrocompatibilité.

- Préserver la performance qui est l'une des motivations principales, pour un utilisateur, d'utiliser un langage d'aussi bas niveau que le C++.

Cet ouvrage inclut un document qui introduit à DATE un sujet qui n'avait alors jamais été couvert dans le domaine matériel, soit les motifs de conception appliqués à la modélisation matérielle, plus précisément à SystemC.

Nous emmenons des points comparatifs entre la méthodologie SystemC (paradigmes de l'orienté objet) par rapport aux méthodologies des anciens langages de descriptions de matériel, plus précisément VHDL.

Nous discutons également de la performance en rapport avec la variabilité dimensionnelle de SystemC et montrons que les pertes de performance sont alors faibles.

Tel qu'il a été montré avec l'historique, bien qu'il existe plusieurs façons de réaliser l'introspection et l'interopérabilité, nous visons une introspection et une interopérabilité au niveau le plus bas afin de permettre un gain maximal de performance. Nous présentons donc diverses façons de réaliser cette introspection et cette interopérabilité.

1.10 Plan de la thèse

Le projet effectué à STMicroelectronics fut présenté en détail dans l'article du chapitre 3 où on évalue comparativement la performance de l'ancienne version de SystemC avec la toute dernière version qui était disponible à l'époque.

Le travail effectué par l'article du chapitre 4 introduit une méthodologie pour établir une interopérabilité entre SystemC et des outils provenant de tierces personnes. L'idée d'utiliser des motifs de conception pour résoudre des problèmes auxquels fait

nouveaux concepts orientés objet, souvent plus puissants ou plus flexibles, y sont présentés en opposition avec une solution plus ou moins équivalente provenant du VHDL.

Bien que quelques figures et exemples aient été réutilisés du précédent article pour présenter SystemC dans le chapitre 6, qui est en fait un chapitre de livre, il reprend l'idée première de la thèse en traitant des motifs de conception appliqués au design matériel, non pas comme des solutions appliquées aux outils de modélisation, comme le voulait le premier article, mais plutôt comme solutions appliquées aux modèles eux-mêmes.

Le chapitre 7 présente un article étendu de la version de l'article IWSOC 2004 de Banff. Le sujet qui y est traité est assez intéressant du fait qu'il apporte une nouvelle vision orientée objet à la modélisation SystemC. Cet article propose une clarification sémantique des constructions SystemC avec la séparation plus nette du modèle et du simulateur. Il propose également l'application du motif « Composite » aux structures de données personnalisées ainsi qu'à la structure même du modèle utilisateur.

Finalement, le chapitre 8 conclut cette thèse et le chapitre 9 présente quelques avenues de recherches futures en rapport avec cette thèse.

Chapitre 2

Aperçu des contributions

2.1 SystemC Performance Evaluation Using A Pipelined DLX Multiprocessor

(chapitre 3 à la page 31)

2.1.1 Objectifs

- Justifier le choix de SystemC comme ;
  1. engin de simulation,
  2. comme cadre de travail.
- Montrer les avantages de SystemC par rapport à VHDL, Verilog, Java, etc.

2.1.2 Contributions

2.1.3 Résumé

Le projet effectué à STMicroelectronics fut présenté en détail dans cet article où on évalue comparativement la performance de l'ancienne version de SystemC, avec la toute dernière version qui était disponible à l'époque. Le but du projet pour STMicroelectronics était de comparer la performance de SystemC par rapport à un de leur outil propriétaire. Du même coup, il fut décidé de tester la toute dernière version de SystemC par rapport à sa version précédente.

Nous avons donc modifié un modèle d'un processeur DLX déjà existant pour qu'il puisse être connecté à un pont contrôlant l'accès à des périphériques d'entrées-sorties, par l'intermédiaire de registres projetés en mémoire. Nous avons utilisé la dérivation et le polymorphisme afin de réaliser une interface commune à tous les périphériques et une fonctionnalité différente en fonction du périphérique.

Nous avons, par la suite, réalisé un bus en anneau unidirectionnel en interconnectant des périphériques d'entrées-sorties qui établissaient une communication entre les différents processeurs, ceci, afin de créer un modèle d'un multiprocesseur. Étant donné la flexibilité du design et de SystemC, nous pouvions dynamiquement (sans avoir besoin de recompiler) créer un nombre arbitraire de processeurs DLX et un nombre équivalent de périphériques de communication associés par le biais d'un paramètre qui était fixé lors du lancement de la simulation, prouvant ainsi que SystemC est un outil qui permet l'exploration d'architecture plus efficacement que ses prédécesseurs.

2.1.4 Résultats

Nous avons montré que SystemC accepte une méthodologie orientée objet tout en

modèle d'un seul processeur en SystemC pour être ensuite instancié assez facilement en plusieurs processeurs qui forment alors, un multiprocesseur. Nous avons aussi caractérisé les performances d'environnement SystemC pour des systèmes modélisés au niveau « cycle-précis ». Nous avons déduit que nous pouvions nous attendre à des cadences d'un peu moins que 100K-instructions de la machine simulée par seconde.

2.1.5 Contexte

Bien que le prochain article fût publié à une époque antérieure au présent article, ce dernier vint valider les hypothèses émises lors de l'écriture du premier article.

2.2 A Methodology for Interfacing Open Source SystemC with a Third Party Software

(chapitre 4 à la page 41)

2.2.1 Objectifs

- Constituer une méthodologie pour établir une communication entre la simulation de SystemC et un quelconque outil provenant d'une entité indépendante au consortium SystemC, ceci, tout en minimisant les modifications à apporter :
  1. à la bibliothèque SystemC,
  2. à la syntaxe du « langage » déjà en place.
- Établir le niveau d'interopérabilité de SystemC.
- Prouver que l'aspect « source ouverte » de SystemC est bénéfique

2.2.2 Contributions

Une méthodologie novatrice pour le domaine fut introduite ; les motifs de conception. Bien que ce concept existe depuis quelque temps dans le domaine du génie logiciel, l'élaboration d'une méthodologie pour le design matériel constitue la contribution majeure de l'article.

2.2.3 Résumé

Le travail décrit par cet article introduit une méthodologie pour établir une interopérabilité entre SystemC et des outils de provenant de tierce personne. L'idée d'utiliser des motifs de conception pour résoudre des problèmes auxquels fait face le monde du matériel fut en premier démontré par cet article.

Nous avons remarqué, lors de la toute première analyse de SystemC, que la bibliothèque se trouvait complètement dénudée de tout système d'introspection. Nous avons donc tenté de combler le vide par un enrichissement d'un système qui serait à la fois utile et qui engendrerait le moins possible de modifications.

Le motif en question est celui de l' « observateur » qui apporte une solution à un problème où une source de données appelée « sujet » doit être partagée entre plusieurs éléments susceptibles de modifier le sujet, appelés « observateurs ». La manière de résoudre le problème est de créer deux classes abstraites d'interface, soit une pour le sujet et une pour les observateurs, qui définissent le protocole de communication à utiliser entre le sujet et l'observateur. On doit ensuite dériver de ces classes, d'autres classes qui implémenteront des fonctionnalités diverses et utiliseront le protocole de communication établi à la base.

2.2.4 Résultats

Nous avons construit un prototype de l'interface de communication ainsi qu'un prototype d'une interface graphique qui réalisent ensemble une communication entre le noyau de simulation de SystemC et notre outil de visualisation de signaux et de contrôle sur simulation, démontrant ainsi le bien-fondé de notre approche et l'interopérabilité potentielle de SystemC.

Un système d'introspection fait encore défaut présentement à la bibliothèque SystemC. Notre approche pourrait donc encore être appliquée avec un peu d'analyse et de modification à la dernière version de SystemC. Advenant le cas où notre approche serait choisie, elle pourrait mener à une norme évolutive et flexible pouvant établir une communication avec les outils externes. Étant donné la méthodologie et, tel que traité dans l'article, la migration et l'évolution au fil du temps se feraient avec aise.

2.2.5 Contexte

Après que nous ayons prouvé que SystemC soit interopérable dû au fait qu'il est à source ouverte, la suite normale consiste à comparer les performances et les apports de ce dernier en rapport avec des langages déjà existant dans le domaine, et c'est ce que nous avons fait avec le prochain sujet.

Caldari et al. [31] se basèrent en partie sur notre article pour démontrer que l'utilisation du C++ était profitable pour le développement de modèles.

2.3 A VHDL/SystemC Comparison in Handling Design Reuse

(chapitre 5 à la page 58)

2.3.1 Objectifs

- Situer SystemC par rapport au domaine.
- Comparer les constructions SystemC à celles d'un langage bien établi dans le domaine (VHDL).
- Prouver que la méthodologie orientée objet fonctionne pour le design du matériel.
- Montrer que SystemC apporte une réutilisation du code plus aisément que VHDL.

2.3.2 Contributions

Nous apportons des solutions orientées objet qui n'ont bien souvent pas d'équivalent en VHDL

2.3.3 Résumé

Pour ce qui est de l'article du chapitre 5, il présente de manière comparative les liens entre les nouveaux paradigmes engendrés par l'utilisation du C++ avec SystemC et les paradigmes de l'« ancienne méthode » qui sont ceux de VHDL ou Verilog. De nouveaux concepts orientés objet, souvent plus puissants ou plus flexibles, y sont présentés en opposition avec une solution plus ou moins équivalente provenant du VHDL.

de cette même interface, mais la réutilisation entre différents niveaux de hiérarchie d'héritage n'a pas d'équivalent en VHDL.

Un autre exemple est le polymorphisme qui permet de laisser l'objet qui implémente l'interface déterminer quel comportement est le plus approprié à la situation.

Une programmation par « meta-template » fut montrée. Ce type de programmation fournie au compilateur des constructions algorithmiques simples lui permettant une meilleure analyse statique, et par conséquent, d'accélérer l'exécution du code. Le compilateur devient donc à même de résoudre directement appels de fonctions [1] et calculs simples.

2.3.4 Résultats

Nous avons montré qu'avec l'héritage il serait facile de créer une bibliothèque de pièces ayant les comportements de base regroupés en des classes de base et les comportements spécifiques implémentés dans les parties spécialisées par dérivation. Il devient donc évident que l'aspect orienté objet apporte une meilleure réutilisation que le VHDL.

2.3.5 Contexte

Maintenant que nous avons montré que le principe orienté objet est applicable, nous avons étudié dans le prochain article la possibilité d'utiliser les motifs de conception requérant ces constructions objet.

Notre article fut utilisé par [4] qui, brièvement, se rapporta à notre illustration de l'utilisation du motif « Singleton » sur lequel il s'appuya pour démontrer que l'aspect

2.4 Applying Multi-Paradigm and Design Pattern Approaches to Hardware/Software Design and Reuse

(chapitre 6 à la page 78)

2.4.1 Objectifs

- Présenter les motifs de conception au domaine du codesign.
- Établir une liste des motifs qui semblent a priori plus utiles que d'autres pour le matériel.
- Montrer que la création de motifs de conception est possible pour le codesign.
- Prouver que SystemC s'emploie bien avec les motifs.

2.4.2 Contributions

Nous avons montré que les motifs existants s'appliquent bien au design matériel et nous avons créé deux motifs pour démontrer qu'il était possible d'en définir de nouveaux.

2.4.3 Résumé

Bien que quelques figures et quelques exemples fussent réutilisés d'un autre article pour présenter SystemC dans le chapitre en question, qui est en fait un chapitre de livre, ce dernier reprend l'idée première en traitant des motifs de conception appliqués au design matériel, non pas comme solution aux outils de modélisation comme le

suite les motifs de conception qui nous semblent les plus appropriés pour le design du matériel, nommément le « Singleton », le « Composition » ainsi que le « Factory » et « Abstract Factory ».

Cependant, il ne faut point croire qu'il ne s'applique, en la liste des différents motifs présentés par Gamma et al. [10], que seulement ces quelques motifs pour la modélisation du matériel. Les motifs sont en fait des guides et exemples qui illustrent la construction correcte de composants selon une expérience acquise depuis bien des années. Nous illustrons ce principe en apportant de nouveaux motifs de notre cru, tout en respectant la sémantique établie par les auteurs des premiers motifs logiciels.

2.4.4 Résultats

Nous avons créé un motif pour les automates à états finis ainsi qu'un motif pour les « Meta template » tout en démontrant aux gens du domaine du matériel l'utilité des motifs de conception en général.

2.4.5 Contexte

Maintenant que l'on possède un outil qui permet l'approche orientée objet, les motifs de conception, il serait bien de regarder un peu plus en profondeur le cœur de SystemC et d'en trouver les défauts, et ce, afin de l'améliorer. Nous allons analyser à quel point nous pouvons perfectionner la structure orientée objet du noyau, non seulement pour améliorer le code source de la bibliothèque, mais essentiellement pour améliorer la qualité du code de l'utilisateur final. Nous explorerons, de plus, les mécanismes d'introspection et d'interopérabilité de bas niveau de SystemC qui se sont

2.5 Using Design Patterns for Type Unification, Structural Unification, Semantic Clarification and Introspection in SystemC

(chapitre 7 à la page 116)

2.5.1 Objectifs

Améliorer SystemC en utilisant quelques motifs de conception :

- avec de meilleurs mécanismes d'introspection de type et de structure du modèle
- discuter des méthodes d'interopérabilité possible
- faciliter la méthodologie de développent des modèles
- clarifier la sémantique

2.5.2 Contributions

Nous apportons des idées d'améliorations de SystemC qui convergent vers une meilleure modélisation orientée objet.

2.5.3 Résumé

Nous ouvrons le chapitre sur une introduction en profondeur sur notre perception de l'introspection et de l'interopérabilité. Nous enchaînons avec une clarification sur ce que nous appelons « knowledge scope » (portée de la connaissance) et sur la distinction entre une interopérabilité intraprocessus et interprocessus.

Une brève description de quelques mécanismes d'introspection et d'interopérabilité

l'introspection des données personnalisées, en employant le motif « Composite » qui nous permet de rabattre toute donnée à un ensemble bien défini.

À partir de l'idée précédente, nous avons ré-appliqué cette même idée, mais cette fois, à la structure même du modèle créé par l'utilisateur de la bibliothèque SystemC. Ainsi, un module peut mieux s'exprimer sous forme d'arborescence hiérarchique, explorable en tout temps.

Enfin, nous proposons une meilleure classification sémantique des objets SystemC, soit une claire séparation entre les classes du modèle et les classes du simulateur, telle que nous l'avons fait avec ESys.Net.

2.5.4 Résultats

En appliquant les idées présentées, les modèles SystemC deviennent plus structurés et plus facilement analysables.

Les idées contenues dans le chapitre peuvent aussi bien être intégrées dans la bibliothèque de SystemC par OSCI, que servir de base à une couche intermédiaire privée, mais normalisée à l'intérieur d'une même compagnie.

2.5.5 Contexte

Nos prochaines explorations porteront sur la sémantique de haut niveau, telle que décrite dans le chapitre. Il serait intéressant de construire un système qui permettrait de mieux identifier la nature des composants outre que par une banale chaîne de caractères.

Chapitre 3

SystemC Performance Evaluation Using A Pipelined DLX Multiprocessor

Luc Charest[1] El Mostapha Aboulhamid[1] Chuck Pilkington[2]
Pierre Paulin[2]

chareslu@iro.umontreal.ca aboulham@iro.umontreal.ca
chuck.pilkington@st.com pierre.paulin@st.com

[1] DIRO, Université de Montréal
2920 Ch. de la Tour
CP6128 Centre-Ville

[2] System-on-chip Platform Automation
STMicroelectronics, Central R&D
16 Fitzgerald Road, Suite 300

3.1 Introduction

The objective of this work is to evaluate the performance of SystemC [1] in modeling a pipelined multiprocessor at a cycle accurate level. The multiprocessor consists of a number of DLX processors [2] connected through a unidirectional ring (Figure 3.1), where every pair of adjacent nodes can send and receive messages concurrently. The predecessor in a ring can only forward the message to its successor. In a ring of n processors, a message from node i to node j must go through the path (i, $i+1 \bmod n$, $i+2 \bmod n$, ..., $j-1 \bmod n$, j)

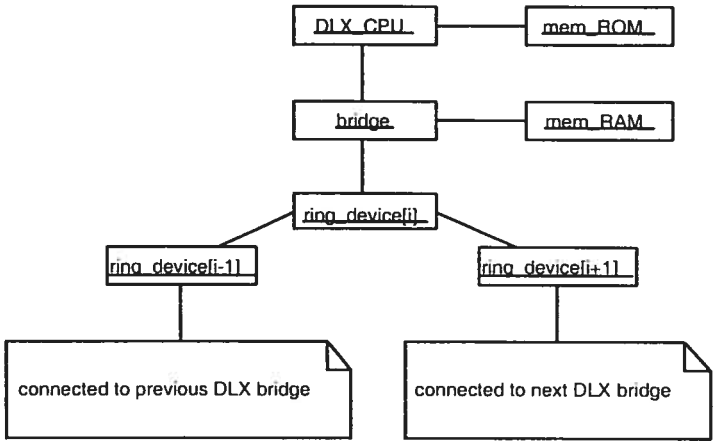


Figure 3.1: *Architecture of a DLX and its adjacent nodes*

3.2 The Model

Figure 3.2 is an UML diagram [3] representing the different classes used to describe the DLX multiprocessor.

getting the instructions out of the ROM (Read Only Memory) (program memory), the second stage is responsible of selecting the operand registers, decoding the instruction and evaluating the branching condition. The third stage is responsible for arithmetic and logical computations as well as memory address calculation. The RAM (data memory) access is performed in the fourth stage. Finally, the results are written back to the destination register if necessary.

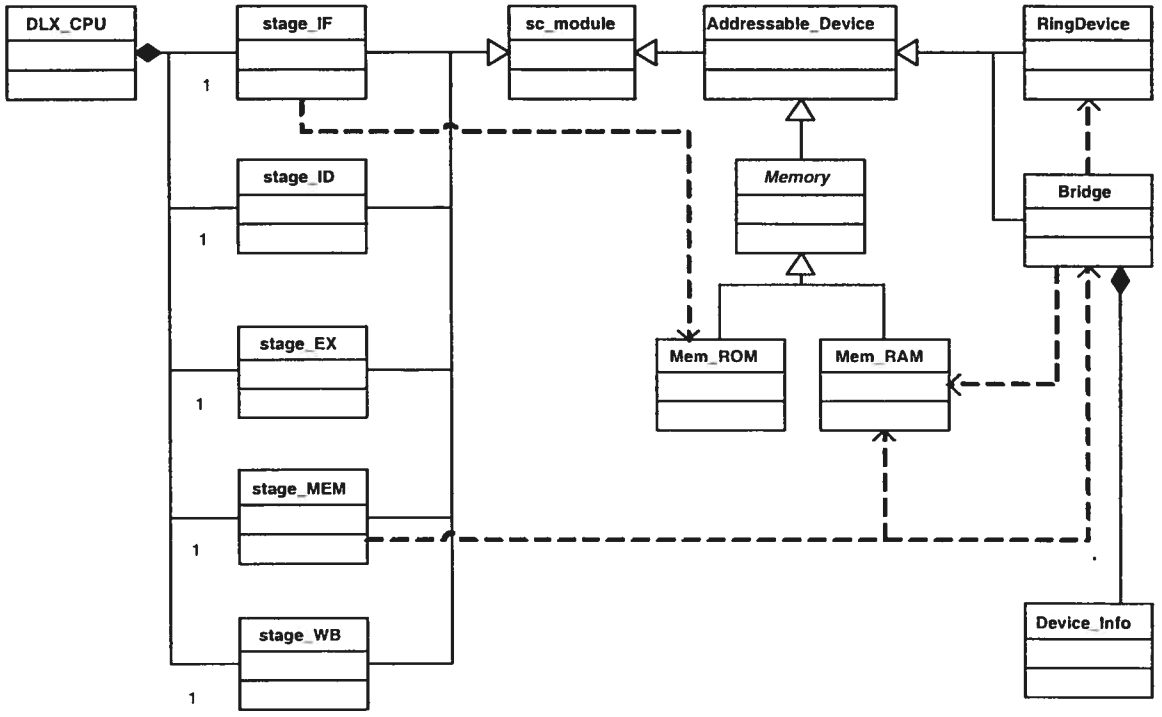


Figure 3.2: *Class diagram of the multiprocessor model*

3.2.2 The Bridge

3.2.3 The Ring Interconnect

At each clock cycle, the ring devices exchange information. The exchange is unidirectional and cannot be blocked. When a message reaches its destination, it is sent to the DLX processor and a free space on the ring is available. Messages issued by DLX i cannot be accepted by the ring device until a free slot (i.e., no message is transiting from node $i-1$ to i during that cycle) is available.

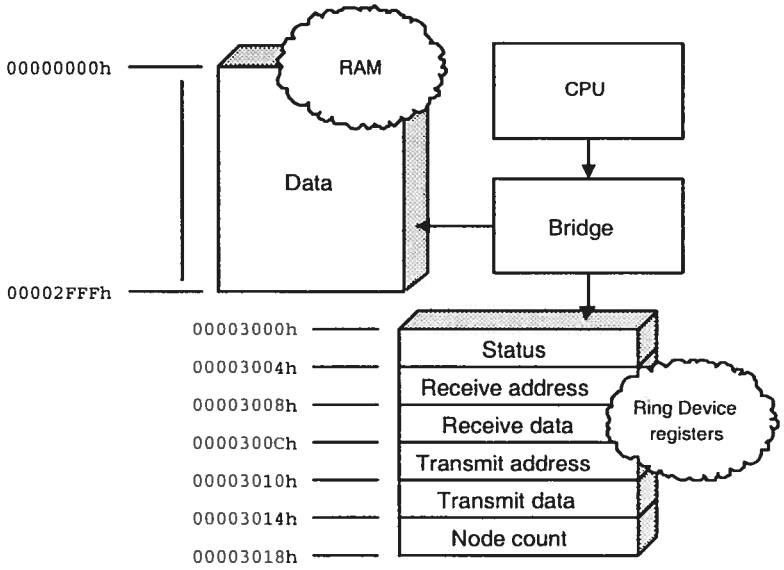


Figure 3.3: *Memory-Mapped Message Exchange*

3.3 The DLX benchmarks

3.3.1 typical6.asm

removed the instructions with invalid codes, and fine-tuned the program until we obtained the instruction coverage described in [2] based on SPECint92 benchmarks; we then added a global loop. The program has wide instruction coverage. It is used to measure the performance of the model of a unique processor; it will be also used in the multiprocessor model.

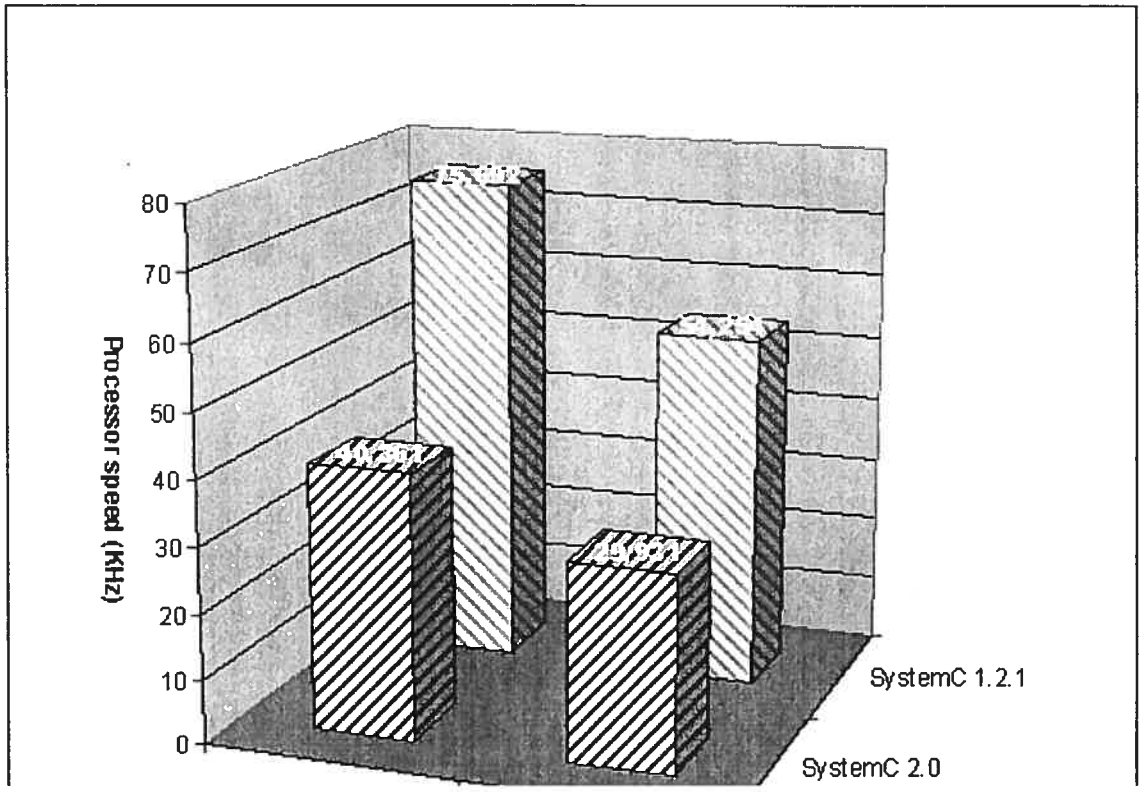
3.3.2 interconnect_test.asm

This program is derived from the typical6.asm program described previously; the difference is that it tries to send a message over the ring, at each iteration of the outmost loop. The message consists of the address of the destination as a 32-bit word and the body of the message on 32 bits too. The program is built to send 1000 messages at a rate of nearly 1 message per 100 cycles. Each Processor i will send all its messages to processor $i-1$. Therefore, each message has to go through $n-1$ hops before reaching its destination. A DLX node cannot issue a new message until the previously issued one has been accepted by the ring device. In our test, we limited the number of simulation cycles to 320,000.

3.4 Experimental Results

SystemC Version	Time (sec.)	KHz
1.2.1 (SC_METHOD)	3.29	75.69
1.2.1 (SC_THREAD)	4.59	54.25
2.0 (SC_METHOD)	6.17	40.36
2.0 (SC_THREAD)	8.32	29.93

program by using the Unix command `/usr/bin/time`. Each experiment has been run three times; and the result tables contain the average of these three independent runs. We run the monoprocessor DLX model for 249,026 cycles. The pipeline stages were modeled first using `SC_METHOD`s and then by using `SC_THREAD`s. From Table 3.1 and Figure 3.4, we can see that the fastest model can run on a very respectable frequency of 76 KHz. Unfortunately the same model will run at 40KHz for SystemC 2.0, on the same 450-MHz machine. Our results seem to indicate that the increase of capabilities of SystemC 2.0 resulted in a decrease of performance at RTL (or cycle accurate level). We may conclude that the early models cannot be exported directly (without change) from earlier versions if we want to keep the same performances.



n	SystemC 1.2.1		SystemC 2.0	
	SC_METHOD (sec.)	SC_THREAD (sec.)	SC_METHOD (sec.)	SC_THREAD (sec.)
2	8.76	12.86	18.06	23.96
4	19.56	28.35	36.62	46.31
8	46.07	65.24	74.43	95.92
16	105.98	144.10	153.15	204.77
32	230.01	327.43	343.04	484.84
40	296.80	432.08	465.36	656.94
64	524.72	804.61	868.12	1239.99
96	875.36	1403.87	1504.14	2085.94
128	1235.98	2010.82	2063.81	2796.77

Table 3.2: *Performance of N-node multiprocessor model*

We run also different models for the multiprocessor DLX. Table 3.2 summarizes the obtained results, where n indicates the number of processors. As previously, the pipeline stages were modeled using `SC_METHOD`s and `SC_THREAD`s. The usage of `SC_THREAD`s results in a 30% increase in the execution time. Usage of SystemC 2.0 instead of SystemC 1.2.1 results in a less severe increase of execution time compared to the monoprocessor case (about 60%).

n	SystemC 1.2.1		SystemC 2.0	
	SC_METHOD (KHz)	SC_THREAD (KHz)	SC_METHOD (KHz)	SC_THREAD (KHz)
2	73.09	49.78	35.44	26.71
4	65.43	45.15	34.95	27.64
8	55.57	39.24	34.39	26.69
16	48.31	35.53	33.43	25.00
32	44.52	31.27	29.85	21.12
40	43.13	29.62	27.51	19.48
64	39.03	25.45	23.59	16.52

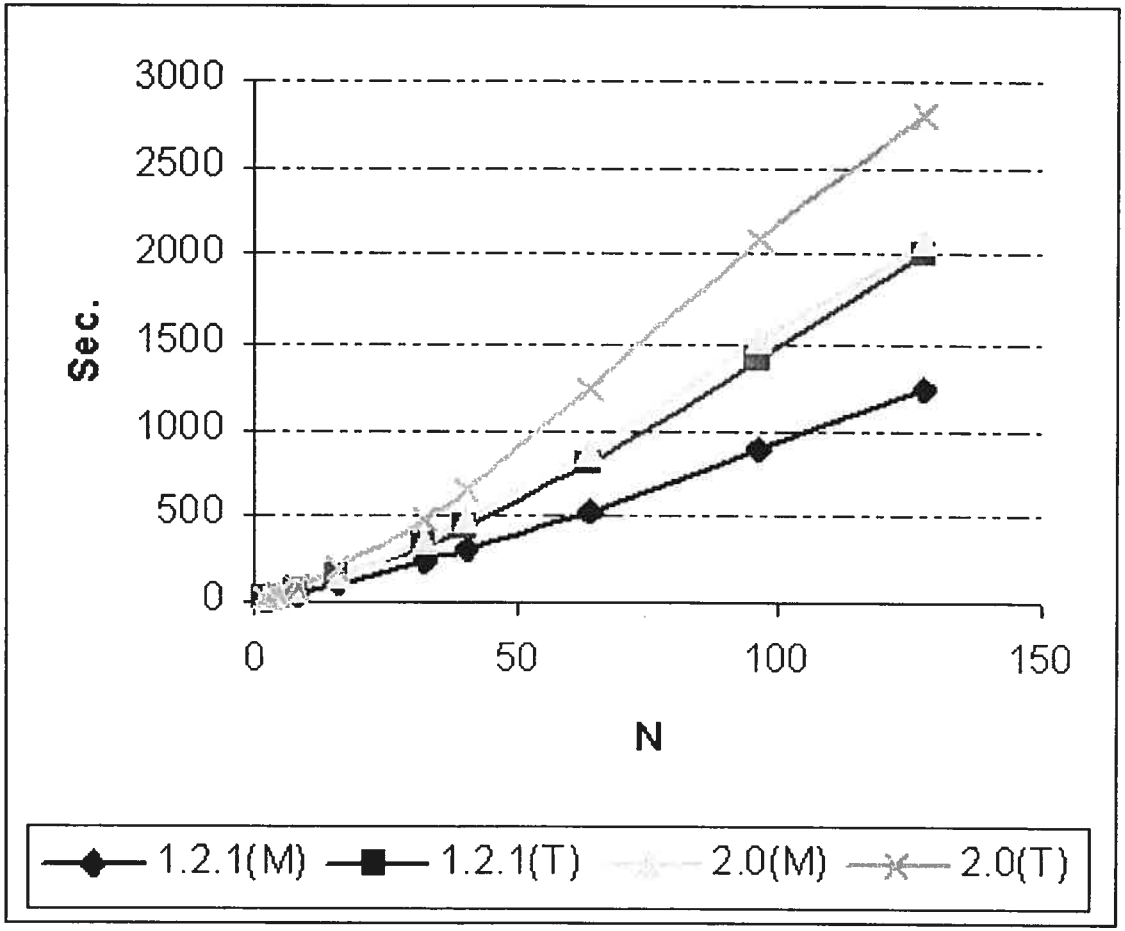


Figure 3.5: *Execution time of the multiprocessor depending on the number of nodes*

to all the messages that should transit through the interconnect is more apparent. As discussed earlier each message should go through $n-1$ hops before reaching its destination, since each processor issues 1000 messages, this results in $1000n^2$ hops to go through. As we limited the number of simulated cycles to 320,000, the effect of this n^2 phenomenon has little impact on the reported results. In our opinion, the nonlinear shape of the curves in Figure 3.5 is due to the simulator overhead in dealing with more complex systems. The model can run as slowly as 14KHz for 128- processor machine, as indicated in Table 3.3. This figure of performance is computed as the number of cycles divided by the execution time, the result being multiplied by the number of processors (Equation 3.1).

$$\begin{aligned} \text{combined\_clock\_frequency} &= \sum_{i=1}^{n} \frac{\text{nb\_processor\_cycles}_i}{\text{processor\_execution\_time}_i} \\ &= \sum_{i=1}^{n} \frac{320000}{\text{simulation\_time}} = \frac{320000n}{\text{simulation\_time}} \end{aligned} \tag{3.1}$$

3.5 Conclusions

We have given some experimental results of SystemC 1.2.1 and 2.0 using a multiprocessor machine modeled at a cycle accurate level. We have shown that models scale harmoniously with the number of processors. Unfortunately, the performances at this level of abstraction of SystemC decreased in the new 2.0 version.

Bibliography

[1] Open SystemC Initiative (OSCI). Functional specification for SystemC 2.0, 2001. `http://www.systemc.org`.

[2] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach, second edition.* Morgan Kaufmann, San Francisco, California, USA, 1995.

[3] Grady Booch, Ivar Jacobson, and James Rumbaugh. *The Unified Modeling Language User Guide 1/e.* Addison Wesley, 1999.

[4] Etienne Bergeron and Eric Lesage. EBEL-DLX, 2001. `http://www.iro.umontreal.ca/~bergeret/EBELDLX`.

Chapitre 4

A Methodology for Interfacing Open Source SystemC with a Third Party Software

Luc Charest[1] Michel Reid[1] El Mostapha Aboulhamid[1]
Guy Bois[2]

chareslu@iro.umontreal.ca reid@iro.umontreal.ca
aboulham@iro.umontreal.ca bois@vlsi.polymtl.ca

[1] DIRO, Université de Montréal
2920 Ch. de la Tour
CP6128 Centre-Ville

[2] Ecole Polytechnique de Montréal
2500, Ch. de Polytechnique
CP6079 Centre-Ville

Abstract

[1] *SystemC is a new open source library in C++ for developing cycle-accurate or more abstract models of software algorithms, hardware architecture and system level designs. SystemC is meant to be an interoperable, modeling platform allowing seamless tool integration. Our objective is to evaluate the feasibility of linking a third party software to SystemC without modifying the SystemC source. We chose the development of a GUI (Graphical User Interface) as such an application. This application illustrates a set of applications following the observer pattern defined recently in software engineering. This class of applications can be loosely coupled to a platform designed following specific rules of software reuse.*

4.1 Introduction

In 1999, systems and embedded software companies announced the OSCI (Open SystemC Initiative) and availability of a C++ modeling platform called SystemC for free distribution [1, 2]. SystemC enables system-level intellectual property model exchange and co-design using a common C++ modeling environment containing a class library and a standard ANSI (American National Standards Institute) C++ compiler. EDA (Electronic design automation) vendors have access to the SystemC modeling platform necessary to build interoperable tools.

Our goal is to explore how easy it is to create such tools with as little as possible modification to SystemC. Since SystemC has presently no GUI, we chose such an application as a candidate. Presently, it is possible to generate a waveform trace available only after the simulation has ended and must be viewed by, usually, a tool similar to Synopsys Waveform Viewer [1]. For feedback during the course of the

it is a very powerful solution, it might not offer a very intuitive way of viewing the results. One would appreciate an interactive visualization tool with a GUI rather than an aftermath solution.

Our objectives are:

- Since SystemC is constantly evolving, it is desirable to develop a loosely coupled GUI to SystemC so that both can evolve independently but cooperate afterwards.
- Use open source libraries to develop the GUI. This will shorten the design time and allow the GUI to evolve in an open environment too.
- Seamless use of the GUI by the designer meaning that the designer can use the present SystemC models without any new syntax. For example, we could have introduced a new class of signals aware of the interface, but this will change the modeling style of the designer and the resulting model could not be exchanged easily.
- Use a Software Engineering methodology enabling other tools to be coupled to SystemC without hampering its own evolution, and providing a standard framework for such integration.

Section 4.2 describes the concept of software patterns. Section 4.3 introduces SystemC architecture. In section 4.4, implementation issues are discussed. The GUI prototype is described in section 4.5. A generalization of the use of design patterns is the topic of section 4.6. Section 4.7 concludes this paper.

4.2 Software Patterns and O.O. Paradigms

software development by providing a common design vocabulary, documentation and learning aids, support for the process of converting an analysis model into an implementation model, and encouragement for reuse of already designed code [4, 5, 6, 7].

These patterns identify, document, and catalog successful solutions to common software problems. Patterns aid the development of reusable software by expressing the structure and collaboration of components to developers at a level higher than source code or objectoriented design models that focus on individual objects and classes.

23 preliminary design patterns are catalogued in [3]. Each pattern is described by its intent, motivation, applicability, structure (in an UML-like format [8]), its implementation, a sample code and its known uses. We will concentrate on the observer pattern, which fits most our application. Its intent is to define a one-to-many dependency between objects so that when one object (subject) changes state, all its dependents (observers) are notified. The number of observers may vary. All the objects are loosely coupled by using abstraction as described later, allowing evolution of the implementation of each object without affecting the other participants.

Design pattern methodology relies heavily on two object-oriented concepts: abstraction and templates.

To implement abstraction, we usually define a base class, which contains certain methods common to all or some children. We then redefine these methods in children that need to override the common behavior given by the base class method. Such an overriding is called polymorphism.

This is done via the keyword virtual in C++. If a method is not declared virtual, a call to this method, using pointers, may lead to the base class method, even if the child class overrides it.

The purpose of templates is to allow programmers to generalize the behavior of a method or a class [9]. When using templates, the programmer does not have to implement the same kind of methods for different types. Only a general statement is given and the compiler analyses the program to know which type is needed. Only needed methods are generated by the compiler following the programmer's template.

4.3 SystemC Architecture

Understanding the internals of SystemC class library is important if we want to link it to another tool and propose changes to improve its reuse. Figure 4.1 illustrates its object structure as well as the relation between objects. This structure has been abbreviated from an UML representation of SystemC. Note that italic means that a class is abstract or a method is virtual. A member preceded by "+", "-", or "#" means the member is public, private, or protected respectively. `sc_object` is the base class of most C++ objects. It contains important basic methods and properties for identifying and classifying SystemC objects. `sc_module` is an organizational SystemC object. An instance of this class contains among other objects a list of ports, although this list is private. `sc_port` is an object, which describes the connections between signals. Ports are important structural information contained usually in modules. `sc_port_manager` is responsible for managing all the ports during simulation. `sc_signal_base` is a general abstract base class for all signals types; it cannot be instantiated but all other signal classes are derived from it. `sc_signal` is a template class which enables the definition of signals of different types. `sc_object_manager` is responsible for managing different SystemC objects and contains a private list of them (i.e. modules, signals,

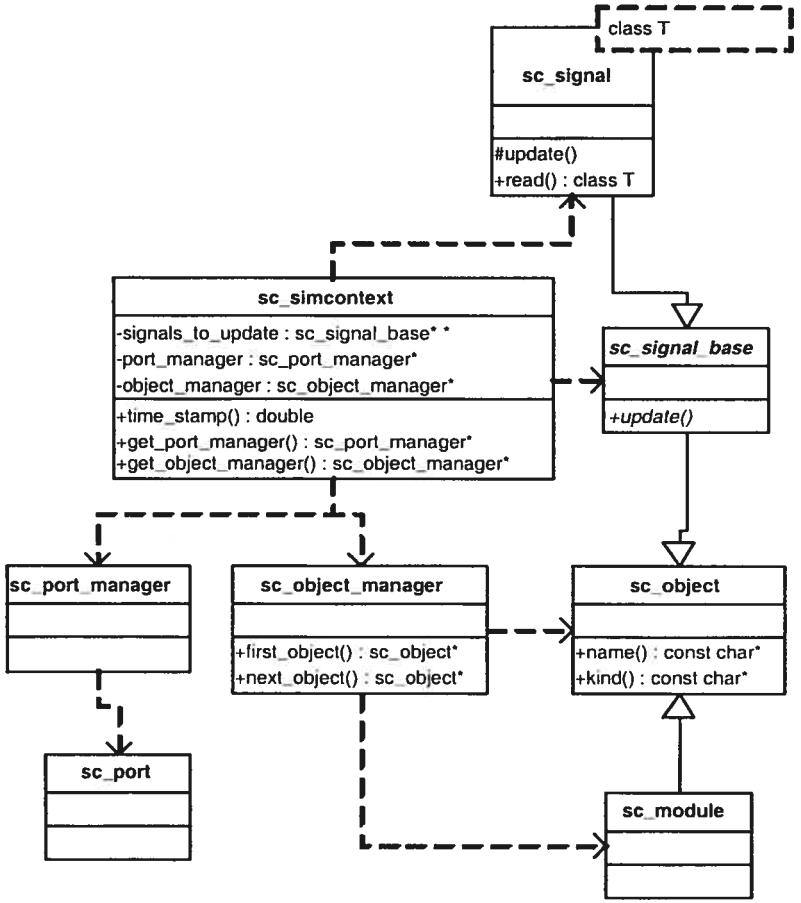


Figure 4.1: *Partial SystemC architecture*

4.4 Implementation Issues

4.4.1 Prerequisites

Since we want to build a GUI to observe simulation results and control its evolution, we need the signal list so we can insert that list into a menu allowing the user to choose the signal(s) he wants to display. We would also need the value of that signal and the time associated with every signal change. Simulation time is provided by the `sc_time_stamp()` method; however getting signal values is more difficult as it will be seen later.

Obtaining the list of modules and the list of ports would allow us to display the signals according to their associated ports making the output more readable.

We explored the existing SystemC code looking for some of the non-documented methods to query its "internal database" of signals and module structure. The search was unsuccessful since its data is well encapsulated behind protected/private properties. A balance should be stroke between flexibility and reuse on one hand, and data protection on the other. In our opinion, the implementation we are proposing reaches that balance.

Note that the changes we are proposing is a new class, which is an abstraction, it includes pure virtual methods and some static members.

4.4.2 Necessary changes to SystemC

In this section we review the changes that we must bring to SystemC in order to allow a loose coupling between SystemC and the GUI. One solution was to implement a method in the `sc_simcontex` class that could fetch the signals and return them as

class for the `sc_signal` and so they were able to create a list of "`sc_signal_base *`". When they insert a pointer toward any signal type derived from `sc_signal_base`, its type is cast automatically.

The problem we are facing is that even though we now have a pointer to an `sc_signal_base`, our GUI is not aware of what type the signal is. If we could know the type of signal, we could type cast the signal to its original form, (e.g.: `(sc_signal<bool> *) signal`) and then use any method of the original signal type.

Because SystemC needed to update all the signals using a general pointers list, they defined a virtual `update(void)` in the `sc_signal_base` class and by redefining a `update(void)` in SystemC `sc_signal` class, when this method is called from a generalized pointer, the appropriate `update()` is called according to the `sc_signal` data type and it is the signal responsibility to update itself.

The problem is that the `read()` method, which returns the actual value of the signal is not virtual and is not in the base class. The reason is that at no point, SystemC needs to read or modify the value of the signal. The signal is read and modified by the user who actually knows which type of signal he is dealing with since he created the signal. We would have liked to implement a new virtual `read()`, but this would require virtual template methods, which are forbidden in C++ [10]. We opted for the declaration of a new pure virtual method `notify_interface()` in the `sc_signal_base` class, which must be redefined in derived classes. When SystemC performs its `crunch()` cycle, we added a call to `notify_interface()` for every modified signal. So from there, it is the responsibility of the signal to notify the GUI of any change using this method. Since the signal knows its data type, it is easy to pass the value of the signal with the appropriate data type to the GUI. Figure 4.2 illustrates the proposed SystemC

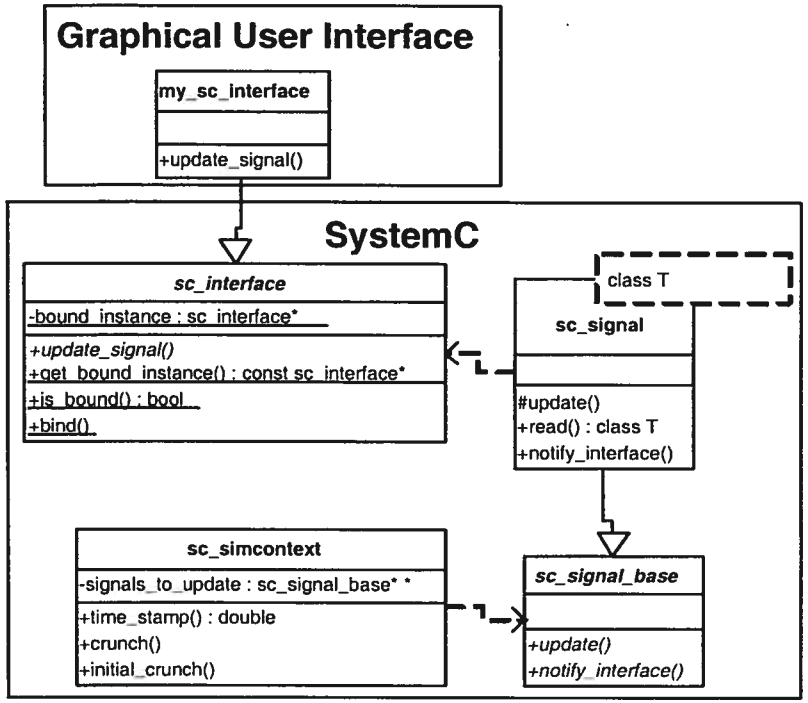


Figure 4.2: *Proposed SystemC architecture*

4.4.3 Constructing the interface

As stated previously, it is the signal responsibility to notify the GUI, but how can it notify the interface? One possibility is sending a message via a method to the GUI. Since SystemC is a standalone library and since we want our GUI to be independent from SystemC, how can we build the SystemC library without having to supply the GUI code to SystemC? Sending a message is usually done by calling a known method of the recipient class. How can we call this method if we do not supply the recipient class?

These questions are answered by following the observer pattern defined earlier

convention of SystemC. This class is abstract because one (in this case all) of its member methods is (are) pure virtual(s), so the class cannot be instantiated. We do not want the class to be instantiated because this class has only one purpose: defining a standard for implementing derived classes. The derived class must implement every method that are pure virtual before it can be instantiated.

Once the interface class is well defined, we can compile SystemC and have it call methods that will be implemented later, in the third party projects. In our case, the `notify_interface()` method of the `sc_signal` class calls the appropriate `update_signal()` of the `sc_interface` class according to the data type. Since we cannot define virtual template methods [10], we were forced to "unroll" the template by prototyping every method with all possible data types.

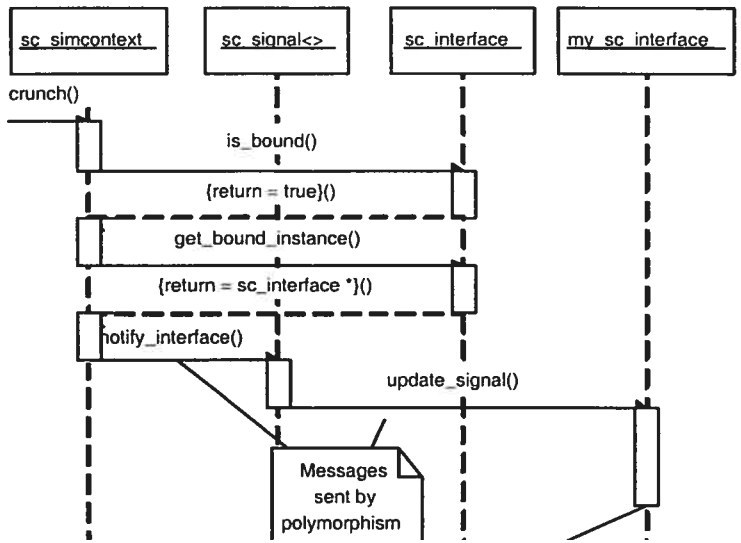
4.4.4 SystemC and the GUI interaction

We have also to define the way the `sc_interface` interacts with its derived class and SystemC. This is due to the fact that we cannot supply a reference of the derived class of `sc_interface` to SystemC since the class is not yet defined when we construct the SystemC library. Therefore we have to provide a pointer to the instance of the derived class, this is accomplished by having a static pointer in the abstract class to point to the instance of the derived class. The static member is common to all instances of the class (and derived classes) and can be used without having any instantiation of the class. We have implemented a method (namely `sc_interface::bind()`) that binds the derived interface to SystemC by setting this static member (`bound_instance`). In our case, the derived class of `sc_interface` is named `my_sc_interface`. So upon instantiation of `my_sc_interface`, the constructor automatically calls the bind method.

`terface::get_bound_instance()` is issued to get the bound interface. The method `update_signal()` of this interface is then called (via polymorphism and inheritance) with the proper data type. Figure 4.3 illustrates this interaction.

4.4.5 Cost and benefits

The cost in execution time for the modification we propose is negligible if no interface is bound. In that case the only addition to the simulation is two if tests during a call to `sc_interface::is_bound()`, which purpose is to return a boolean value. The first test is made once for each signal in the `initial_crunch()` loop. The second test is made in the `crunch()` loop, each time a signal is updated. Of course, if an interface to a third party tool is bound, the simulation time increases when the control is given to it. Cost in time will mainly be in recording or analyzing the data shipped via the notification. Among the benefits of this approach, is to get rid of the printf or cout statements, which clutter the model and slow the simulation.



4.5 GUI Prototype

In our development, we chose Qt because it is a fully object-oriented, cross-platform C++ GUI application framework providing application developers with all the functionality needed to build GUIs [11]. Qt is available on a wide range of platforms including Linux and it is free for development of free/Open Source software under Unix/X11. Qt offers objects like menus, windows, buttons, etc. We also used the STL (Standard Template Library) [12] when objects like lists, queues, and vectors were needed. This results in minimizing time for development as well as increasing code efficiency.

The GUI prototype displays Boolean signals traced and listed in windows as in Figure 4.4. Fundamentals C++ types, such as integers, can be displayed but there is still work to do for the more complex SystemC basic types.

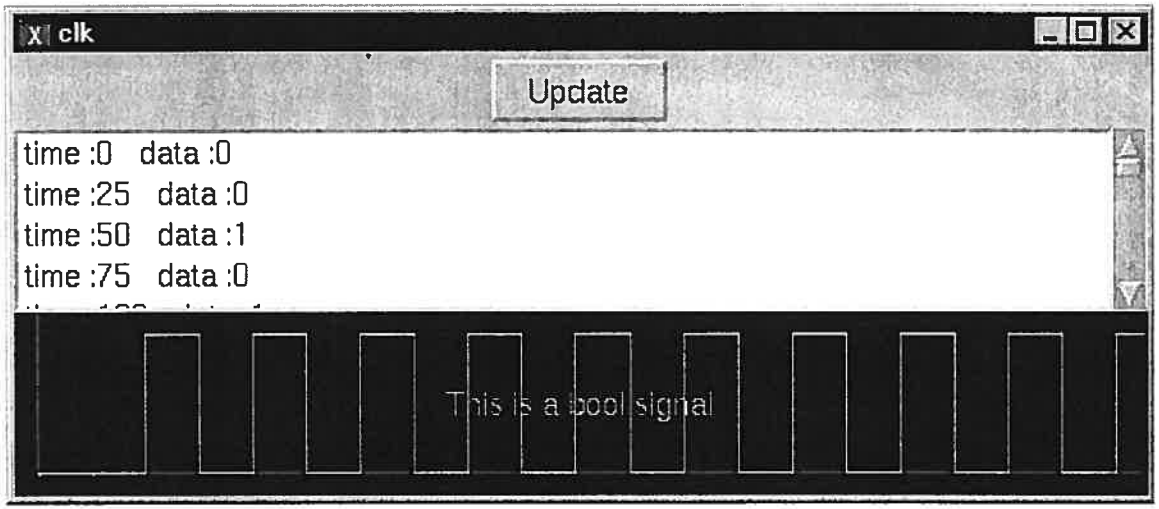


Figure 4.4: *Displaying the signal value*

The GUI system is compiled independently from the end user's project and is linked as a library. Few modifications must be made to end user's code to create the main window and then pass the execution control to it. If the user wants to be able to debug his signals with meaningful names, he must also attribute a name to each of his signal by modifying his code to use an undocumented SystemC signal constructor.

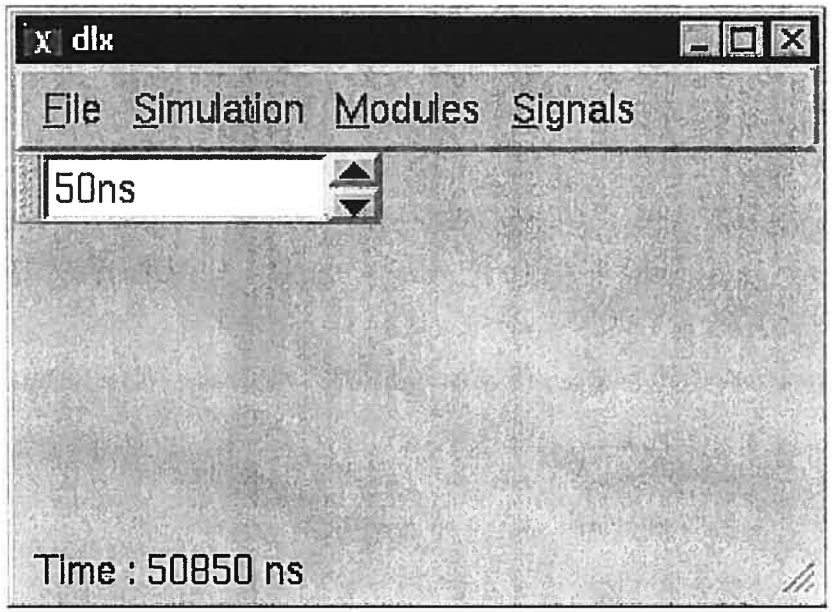


Figure 4.5: *Simulation control from the GUI*

4.6 design patterns to solve more general problems

As an extension of this work, it seems that the `sc_interface` class can be generalized to be used as a facade pattern. A facade is a pattern that provides a unified interface in a subsystem. The facade pattern defines a higher level interface that makes the

control role and an interface with a third party role. We found it more convenient to separate the responsibilities between two classes, simplifying at the same time the classes structure.

If the `sc_interface` would act as a portal between SystemC and third party applications, this class should be designed carefully to include all the necessary utility methods. This way, a programmer would have a variety of querying, and modifying tools to design his SystemC "plug-in" modules. So our recommendation is to use the `sc_interface` as a kind of two-way facade with SystemC, and provide as many methods as we can. If implementing all the interfacing methods is too overwhelming, "dummy" methods, returning default values, can be defined instead. This would provide a standard on which third party tools may be developed. Programmers of the third party tools could then easily verify that the method as been implemented by checking its returned value. As soon as SystemC implements these methods, all third party tools will have their behavior immediately changed without having to be re-engineered or recompiled.

There is another pattern, "adapter", that enables the communication between two systems, which have different protocols or structures. The `sc_interface` may serve as an adapter, if one desires to adapt an existing tool to SystemC, by deriving and implementing `sc_interface`. If SystemC wishes to modify fundamentally its core, `sc_interface` input could be manipulated to fit the old `sc_interface` standard and a new one could be defined, in parallel, in the same `sc_interface`.

4.7 Conclusion

If the SystemC community embraces this methodology and the pattern is well defined, well maintained and adequately documented, it could be a powerful framework on which third party software could be built without having to modify the SystemC core or the end user model while keeping the data encapsulated and oblivious to outside manipulation.

Bibliography

[1] Joachim Gerlach and Wolfgang Rosenstiel. System level design using the SystemC modeling platform. In *Workshop on System Design Automation (SDA'00)*, pages 185–189, Rathen, Germany, 2000.

[2] Open SystemC Initiative (OSCI). Functional specification for SystemC 2.0, 2001. `http://www.systemc.org`.

[3] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, 1994.

[4] Shriram Krishnamurthi and Matthias Felleisen. Toward a formal theory of extensible software. In *ACM SIGSOFT sixth international symposium on Foundations of software engineering*, pages 88–98, 1998.

[5] Douglas C. Schmidt. Using design patterns to develop reusable object-oriented communication software. *Commun. ACM*, 38(10):65–74, 1995.

[6] D. Schmidt. Using design patterns to guide the development of reusable object-oriented software. *ACM Comput. Surv.*, 28(4), 1996.

[9] Stanley B. Lippman and J Lajoie. *C++ Primer, 3/e.* Addison Wesley, 1998.

[10] Bjarne Stroustrup. Publications by Bjarne Stroustrup, 2000. `http://www.research.att.com/~bs/papers.html`.

[11] Trolltech AS. Qt on-line reference documentation, 2001 1996. `http://doc.trolltech.com`.

[12] Silicon Graphics Computer Systems Inc. Standard template library programmer's guide, 2001 1993. `http://www.sgi.com/tech/stl`.

Chapitre 5

A VHDL/SystemC Comparison in Handling Design Reuse

Luc Charest El Mostapha Aboulhamid

chareslu@iro.umontreal.ca aboulham@iro.umontreal.ca

DIRO, Université de Montréal

2920 Ch. de la Tour

CP6128 Centre-Ville

Montréal, Qc, Canada H3C 3J7

Abstract

This paper describes the use of the multi-paradigm aspects of SystemC to develop hardware libraries, to reuse designs, to accelerate simulation, and to allow efficient design space exploration both at the RTL and architectural levels. Concepts of commonality and variation supported by C++ are used to compare SystemC capabilities to those of VHDL.

5.1 Introduction

SystemC is a modeling platform consisting of C++ class libraries and a simulation kernel for design at the system-behavioral and register-transfer-levels. Designers create models using SystemC and standard ANSI C++ [1]. Different models of computation and design methodologies may be used in conjunction with SystemC. The design libraries and models needed to support these specific design methodologies are considered to be separate from the SystemC core language standard. This work may be considered as a design methodology for RTL and architectural models. By its nature, SystemC inherits the capabilities of C++, like the support of multiple paradigms: classes, overloaded functions, templates, modules, ordinary procedural programming, and others. The freedom afforded by these capabilities may also bring new difficulties to the understanding of the model or the ability to synthesize it. The objective of this paper is to demonstrate the benefits of this multiparadigm environment, and show that the advantages outweigh the risks. We will show that we will be able to develop more resilient specifications and models, express different behaviors and views of the same "entity", facilitating design exploration and ease specification and development

software programmable components, many designers are leaning toward a common language for hardware and software modeling. In order to make a fair comparison, we intentionally restrict ourselves to hardware modeling both at the functional and RTL level. We will mention only in passing the system design capabilities of SystemC 2.0 (like Channels and interfaces). System-level capabilities using C++ and/or SystemC are well described elsewhere [3, 4]. This may serve also as guideline for VHDL designers to understand and potentially adopt the SystemC methodology. In our comparison, we will use the concept of commonality and variation developed by Coplien [5, 6] to compare the two environments.

In this paper, Section 5.2 describes the desirable characteristics of a design environment; Section 5.3 introduces the commonality and variation concepts; Section 5.4 deals with design reuse and hardware libraries; Section 5.5 discusses configuration and variation issues; Section 5.6 shows a combination of mechanisms in an integrated example; finally, Section 5.7 concludes this work.

5.2 Characteristics of a Design Environment

In our opinion, the desirable characteristics a hardware designer is looking for are as follows:

Reuse: hierarchy is one of the mechanisms that allow reuse. Structural hierarchy is available in VHDL but not behavior hierarchy, as will be illustrated later.

Refinement and automatic path to synthesis.

Families of design: often a design house will be interested in developing a family of

Express design constraints: related to timing, testability, number of processors, etc.

During the design exploration phase it would be very helpful to break the modeling, analysis, elaboration, and simulation cycle by allowing online configuration of a model. This leads to more freedom in comparing results, and a shorter time for the analyzer to evaluate its design. The new cycle would be: model, compile, and examine the execution phase by dynamically reconfiguring the model. If the modules are fully interfaced, implemented and debugged, their construction could be controlled via a GUI at execution time.

Link to the outside world and extension of the capabilities of the environment. In VHDL, this can only be done in a limited fashion via C language interfaces. Since SystemC is an open source environment we have seen some efforts in developing other tools around the SystemC core as well as methodologies to have interoperable environments [7].

Minimize the number of environments and languages to deal with: in the case of systems containing both hardware and software, SystemC seems to have an advantage.

Have delay models that fit the need of the different levels of abstraction. Delay models are very restrictive in VHDL (inertial, delta and transport delays). These seem sufficient for describing hardware components. However they are very inefficient when developing test-benches. Testbenches and abstract architectures necessitate FIFOs, queues and stacks for example. This is very possible

resilient, if well documented. In this case, SystemC seems to have a big advantage compared to VHDL. It is open source and the models are developed using the same paradigms as those used to develop SystemC itself. GNU tools like Doxygen [8] may be used to both document SystemC kernel as well as the models developed using SystemC [9]. Graphical documentation can also be easily obtained using UML [10].

5.3 Commonality and Variation in VHDL

Hardware designers are under pressure to develop new design derivatives in less time. In the case of software systems, the benefits of explicitly identifying the common and variable aspects of the different versions of a system are presented in [5]. We will illustrate the ability of SystemC, in comparison to VHDL, to express commonality as well as variation in modeling of hardware systems. VHDL [11] allows the expression of commonality by what is called design units. A design unit is a VHDL construct that may be independently analyzed and inserted in a design library. These design units are:

Entity declaration: describes the interface view of a component (like a Data Book description). It is implementation independent.

Architecture body: describes an implementation of an entity (like a single schematic diagram). A single interface may have alternative architectures.

Package (declaration and body): contains information common to many design units. This information consists of functions, types, signals, and constants. It

Commonality can also expressed by generate statements and generic constants for regular structures like a ripple carry adder or an interconnection network. Variation is obtained by configuration, which allow the designer to choose an architecture among many others during design space exploration; by giving a specific value to a generic parameter; by overloading functions and subprograms in coordination with packages allowing the reuse of models even if we change the basic data types, like going from a bit type to a 9- valued standard logic. Commonality between processes is very limited except if we choose very complex ways like concurrent procedure calls rendering models quite cryptic. In SystemC 2.0, at the system level, commonality is expressed by abstract classes such as interfaces, then variation is encapsulated from interfaces with a possibility of multiple inheritance. Commonality can also expressed as use of design patterns [12] and other means as described later.

5.4 Design Reuse and Hardware Libraries

As seen before, the main mechanism for design reuse in VHDL is libraries of design units. Structural hierarchy is the way to reuse components declared in a library. On the other side SystemC has the same capability as VHDL and also other mechanisms, based especially on inheritance, templates and overloading.

5.4.1 Module Inheritance in SystemC

In designing a library of gates, latches, etc., we can specify all the properties related to gates in a module `Gate` from which, `And`, `Or`, `Xor`, etc. will be derived later. The purpose of the `Gate` module is to act as an interface. All the properties and methods for Flip-Flops will be captured in a similar module (Setup and hold time, delays, etc.)

```
6   };
7
8   class And : public Gate
9   {
10    public :
11      SC_HAS_PROCESS(And);
12      And(const sc_module_name& name): Gate(name)
13      { (...) }
14  };
```

5.4.2 Using Inheritance to Insert Tags or Attributes

In VHDL, we can reuse a component with a variation of behavior or semantics based on attributes. We can, for example, use attributes to specify that a process is at the RTL or behavioral level. However, it is impossible to derive processes from previous ones, or specialize their methods. In modern software-oriented methodology, recommendations are made supporting a common root for all objects in the software design. `sc_object` constitutes such a root in the core of the SystemC library. For the purpose of modeling, a library of components and IPs, `sc_module` can be the building block for the models. From this we may derive other specialized modules, as described in Figure 5.1.

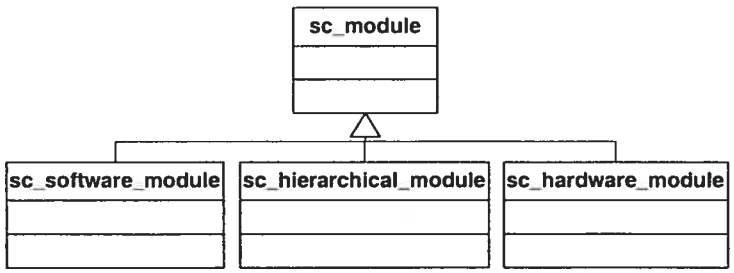


Figure 5.1: *Module specialization*

From this point, we can use these basic building blocks to construct our libraries

his design. This kind of construction could be recognized by synthesis analyzers without having to alter SystemC syntax or behavior, but is part of the static configuration and we will see later how to make this kind of choice more dynamic.

5.4.3 Hierarchical Module Construction for SystemC Hardware Libraries

Class hierarchies can be a great help when designing hardware libraries. Here are some proven advantages:

Modularity of structure: hardware properties are reused in derived classes.

Locality of code: when you have something to modify, you know it is in the class or its ancestor(s)

Reusability of code: software implementations in base classes are reused in derived classes.

Figure 5.2 represents a basic gate library:

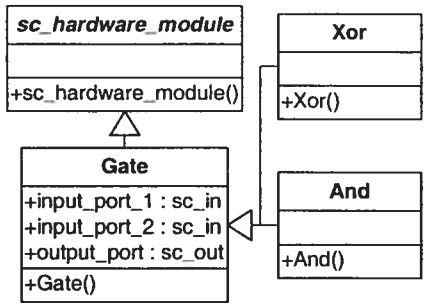


Figure 5.2: *Behavioral hierarchy*

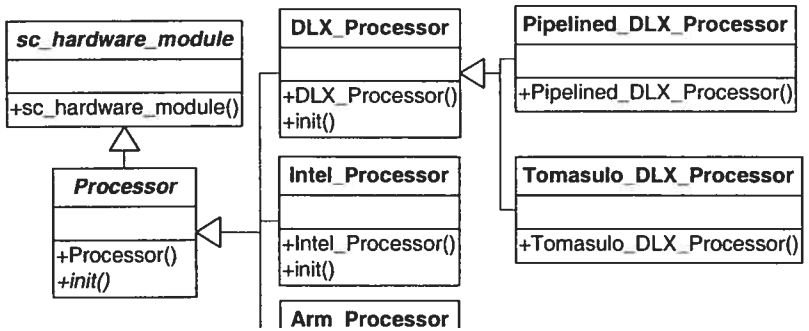
```
1  And::And(...) : public Gate(...)
2  {
3    SC_METHOD(main_process)
4    sensitive << input_port_1 << input_port_2;
5  }
```

This duplication of definition can be avoided. In Figure 5.3, if the virtual `init()` method defined in the Processor class is sensitive to some signals, then the derived init methods in classes `DLX_Processor`, etc., will inherit this sensitivity. Therefore, a library could be constructed from ground up, by adding and refining the hardware components.

5.5 Variation and Configuration

5.5.1 Multiple Architectures

Using VHDL, we can have multiple architectures related to the same entity. We can consider the entity as the abstract class and different architecture as the derived classes from the basic abstract class. Note that in VHDL, this derivation process is limited to two levels of abstractions. In SystemC, this refinement process can continue indefinitely as illustrated in Figure 5.3.



5.5.2 Regular Structures and Their Configuration

In VHDL, generate statements are used to model regular structures composed of processes or components. In SystemC, a simple declaration of an array of components is sufficient, as illustrated below. Configuration is obtained in a very natural way by instantiation.

```
1  Processor *processor_array[10];
2  processor_array[0]=new Arm_Processor();
3  processor_array[1]=new DLX_Processor();
4  processor_array[2]=new Intel_Processor();
```

The configuration could be put in a file and then read at execution time during elaboration. A switch statement could then be used to instantiate the different processors. This has no correspondence in VHDL:

```
1  int i = 0;
2  FILE input = fopen("cpu.cnf", "rt");
3    while (!eof(input))
4    {
5      char buffer[500];
6      buffer = fgets(input);
7      switch(atoi(buffer))
8      {
9        case ARM :
10         processor_array[i++] = new Arm_Processor();
11         break;
12
13       case INTEL :
14         processor_array[i++] = new Intel_Processor();
15         break;
16
17       (...)
18     }
19   }
```

5.5.3 C++ Polymorphism in SystemC Library

If we reexamine the UML behavioral refinement of Figure 5.3, we note that the abstract

the abstract class processor. Depending on the true instance of the processor the right init method will be invoked by polymorphism:

```
1  for (int i = 0; i < 10; i++)
2    //initialize all processors
3    processor[i]->init();
```

If we desire the same effect in VHDL we would need a special signal `Reset_Processor`, which should feed only processor components. This is less elegant and less readable than what we have here. It is also surely more time consuming during simulation because events on signals put a very heavy burden on the VHDL simulator.

5.5.4 Using Overloading Mechanism to Change the Behavior

In VHDL, overloading is limited to functions and procedures. In SystemC, not only methods but also constructors of classes can be overloaded allowing more dynamic configuration of threads and modules. To determine the correct method to call, the compiler only looks at the type of the parameter(s) when the method call is issued.

This is an example of overloaded constructor for a Processor object:

```
1  Processor::Processor(const sc_module_name &name, int bus_format)
2    : sc_hardware_module(name)
3  {
4    bus = new sc_in<int>();
5    (...)
6  }
7
8  Processor::Processor(const sc_module_name &name, char bus_format)
9    : sc_hardware_module(name)
10 {
11   bus = new sc_in<char>();
12   (...)
13 }
```

have a particular behavior or any optimization, the only way is to test the parameter inside the architecture related to the generic entity. In SystemC, by using overloading we can achieve this in a more efficient way as illustrated below:

```
1  void And::compute(void);
2  {
3    output_port = input_port_1 && input_port_2;
4  }
5
6  void And::compute(int n_ports);
7  {
8    if (n_ports != 0)
9      for (int i = 0; i < n_ports; i++)
10       (...)
11 }
```

5.5.6 Using Templates to Describe Regular Behavior and its Elaboration at Compilation Time

Interconnection networks or other behaviors can be easily described recursively. (However, it seems that there is no equivalent to this in the hardware world). We will show that by using templates [13], we can have very abstract descriptions translated in an iterative behavior at compile time, hiding these abstractions from the simulator, and potentially the synthesis tool.

```
1  template<int n_ports>
2  class Gate : public sc_module
3  {
4    public :
5      sc_in<bool> input_ports[n_ports];
6      sc_out<bool> output_port;
7      (...)
8  };
9
10 template<int N>
11 class Compute
12 {
```

```
21  class Compute<2>
22  {
23    public:
24      static inline void compute(bool &result, sc_in<bool> *input_ports)
25      {
26        result = input_ports[0] &
27        input_ports[1];
28      }
29  };
30
31  template<int n_ports>
32  class And : public Gate<n_ports>
33  {
34    public :
35      SC_HAS_PROCESS(And);
36      void compute_process(void)
37      {
38        bool result;
39        while(1)
40        {
41          Compute<n_ports>::compute(result, input_ports);
42          output_port = result;
43          wait();
44        }
45      }
46
47    And(const sc_module_name& name, sc_clock &clk)
48      : Gate<n_ports>(name)
49    {
50      SC_THREAD(compute_process);
51      sensitive_pos << clk;
52    }
53  };
```

Templates are resolved at compilation time. The instantiation for a specific And gate can be done either by:

```
1  #define FOO 10
2  new And<FOO>("simple_And", clk);
```

or by:

```
1  const int foo = 10;
```

```
4     output_port = result;
5     wait();
6   }
```

would result in the following iterative one:

```
1   while(1)
2   {
3     result = input_ports[0] & input_ports[1];
4     result = result & input_ports[2];
5     result = result & input_ports[3];
6     (...)
7     result = result & input_ports[9];
8     output_port = result;
9     wait();
10  }
```

This is because the compiler unrolls the template inlined method calls. This allows the compiler to perform very useful optimizations. The drawback is that this is being part of the static configuration (like in VHDL) where you must modify your code and then recompile.

5.5.7 Default Parameters

Another useful thing that might be done with overloading is fault simulation. You could want to simulate a faulty hardware to see how the system would react. This can be done with an overloaded method nor using default parameter. A default parameter will sometime act as an overloaded method and change the behavior according to our needs:

```
1   //prototype
2   void Gate::compute(int faulty_port=-1);
3   //declaration
4   void Gate::compute(int faulty_port)
5   {
6     if (faulty_port != -1)
7     {
```

5.5.8 Design Patterns

Originally, design patterns [12] are meant to identify, document, and catalog successful solutions to common software problems. Patterns aid the development of reusable software by expressing the structure and collaboration of components to developers at a level higher than source code or object-oriented design models that focus on individual objects and classes. Their use can be extended to development of models for hardware components. As a simple example, we will illustrate their use by the singleton pattern.

The singleton pattern is a software engineering solution when you need to have one and only one instance of a class (this pattern could be generalized to n instances quite easily). This pattern forbids the user to have more than one instance either by sending a warning or by returning a reference to the already created instance. The main idea of the solution is to "hide" the constructor from public access, then provide a public static method to control the instantiation of the class. One way of using this approach when dealing with configurability (Figure 5.4) is to have a base class decide which derived class is best suitable to instantiate.

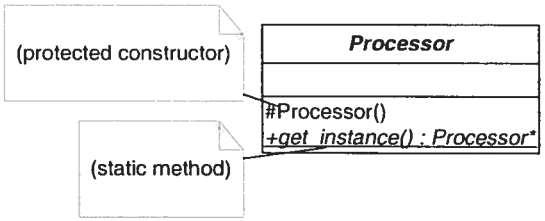


Figure 5.4: *A configurable uni-processor system*

```
1  Processor *Processor::get_instance(int type)
```

```
10          return new Intel_Processor();
11          break;
12      }
13  }
```

The problem with this kind of approach is that usually, it is the derived class that is "aware" of what is happening in the base class while the base class knows nothing about its children. However this solution is not easily extendable: when a derived class is added, the base class must be modified.

5.5.9 The Manager

A manager class can also be defined with a parameter that will allow a "dynamic" configuration of the final design:

```
1   Process *Process_Manager::get_processor(int type)
2   {
3     switch(type)
4     {
5       case ARM :
6         return new Arm_Processor();
7         break;
8
9       case INTEL :
10        return new Intel_Processor();
11        break;
12    }
13  }
```

5.6 Combining Mechanisms

The highest configuration flexibility is achieved by combining all the above mechanisms together. With an overloaded method in a base class, you can choose a constructor of the derived class that best suits your need. With method overload

`behavioral_process()` is an abstract method that simulates the module from a behavioral perspective, while `structural_process()` is the one that should represent the module by composition from other modules. Because the initialization phase might be different, `init_behavioral()` and `init_structural()` methods are provided and the general `init()` method is responsible for calling the appropriate initialization method. The constructor of the `sc_hierarchical_module` is the one that builds the module and chooses whether you are going to use the behavioral or the structural definition of the derived class. the intent of `sc_hierarchical_module` is to indicate that the module might be composed of other modules. When the hierarchical module is composed of only one module (might be changed at run time during initialization phase), we can use the method `has_subcomponent()` to determine the real nature of the module. This flag hides the Boolean value that is adjusted by the constructor at compile time. The list of sub modules is adjusted so you can then follow the hierarchy.

5.7 Conclusion

In this paper, we described the use of the multiparadigms aspects of SystemC to develop hardware libraries, to reuse designs, and to allow efficient design space exploration both at the RTL and architectural levels. Concepts of commonality and variation are used to compare SystemC capabilities to those of VHDL. The use of the presented methodology will allow the development of resilient and well documents models, it may also help in developing efficient test-benches.

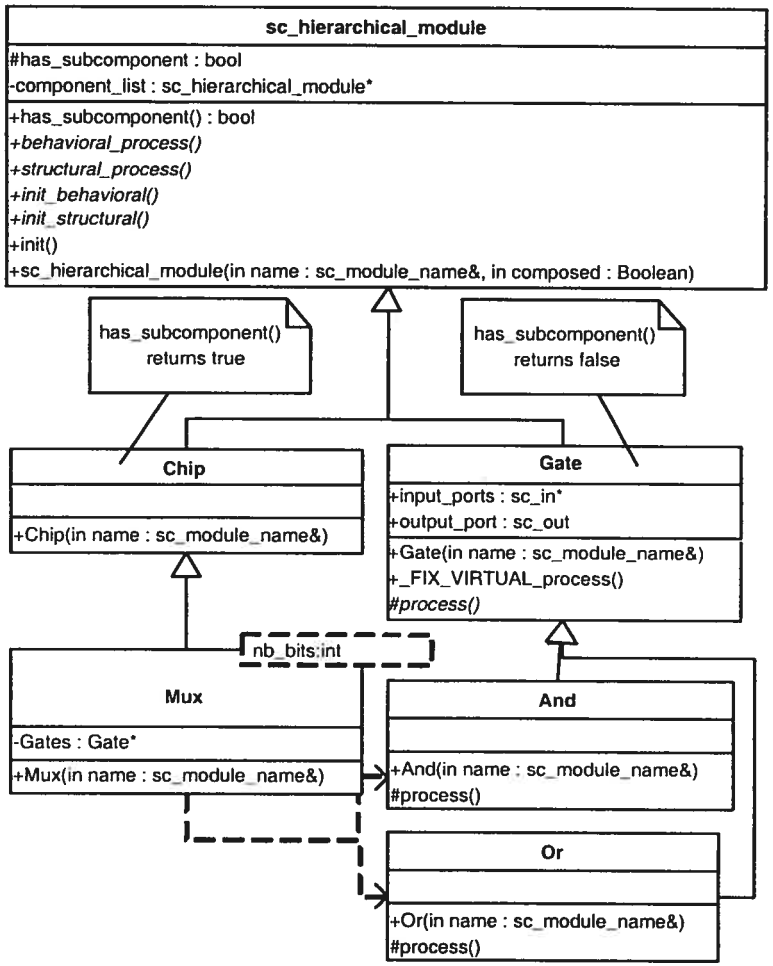


Figure 5.5: *Multi-paradigm Hierarchy*

Bibliography

[1] Open SystemC Initiative (OSCI). Functional specification for SystemC 2.0, 2001. `http://www.systemc.org`.

[2] Sowmitri Swamy, Arthur Molin, and Burt Covnot. OO-VHDL: Object-oriented extensions to VHDL. *Computer*, 28(10):18–26, 1995. 0018-9162/95/$04.00 (c) 1995 IEEE Features.

[3] D. Verkest, J. Kunkel, and F. Schirrmeister. System level design using C++. In *Design, Automation and Test in Europe*, Paris, France, 2000.

[4] Open SystemC Initiative (OSCI). SystemC, 1999-2001. `http://www.systemc.org`.

[5] James Coplien, Daniel Hoffman, and David Weiss. Commonality and variability in software engineering. *IEEE Software*, 15(6):37–45, 1998. James O. Coplien, Daniel M. Hoffman, and David M. Weiss. Commonality and Variability in Software Engineering.

[6] J. Coplien. *Multi-Paradigm Design for C++*. Addison-Wesley, Reading, MA, 1999.

[8] Dimitri van Heesch. Doxygen, 2001 1997. `http://www.stack.nl/~dimitri/doxygen/index.html`.

[9] Luc Charest. SystemC documentation, 2001 2000. `http://www.iro.umontreal.ca/~chareslu`.

[10] Grady Booch, Ivar Jacobson, and James Rumbaugh. *The Unified Modeling Language User Guide 1/e.* Addison Wesley, 1999.

[11] *IEEE Standard VHDL Language Reference Manual.* IEEE, 1076,2000 edition, 2000.

[12] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, 1994.

[13] T. Veldhuizen. Using C++ template metaprograms. *C++ Report*, 4(4):36–43, 1995.

Chapitre 6

Applying Multi-Paradigm and Design Pattern Approaches to Hardware/Software Design and Reuse

Luc Charest[1] El Mostapha Aboulhamid[1] Guy Bois[2]

chareslu@iro.umontreal.ca aboulham@iro.umontreal.ca

bois@vlsi.polymtl.ca

[1] DIRO, Université de Montréal
2920 Ch. de la Tour
CP6128 Centre-Ville

[2] Ecole Polytechnique de Montréal
2500, Ch. de Polytechnique
CP6079 Centre-Ville

6.1 Introduction

It is now feasible to manufacture chips having 100 M transistors. This is leading to the adoption of SOC (System-On-a-Chip) designs. Complexity, time-to-market pressure and evolving requirements push SOC design to reuse IP (Intellectual Property) blocks and build around programmable platforms. By opposition to the ASIC (Application Specific Integrated Circuits)s where all the desired functionality is hardwired, programmable platforms contain programmable devices (such as processors) and specific functions are hardwired to meet some performance requirements; programmability offers flexibility, easy extensibility and adaptability to new requirements. This results in new challenges in the design process. Higher levels of abstraction must be defined. Description languages must handle both hardware and software components, and must be able to capture constraints and describe communication between them. CAD (Computer-Aided-Design) tools must allow integration of IP blocks, HW/SW (HardWare/SoftWare) partitioning and communication mapping to obtain the expected performance. And finally verification must handle integrated hardware and software modules.

Co-design methodology must address the issues of SOC design where different IP blocks have different origins (not only from one organization). The following description of a typical co-design flow (Figure 6.1) is mainly borrowed from [1]. Co-design is a joint specification, design, and synthesis of mixed hardware and software components in order to obtain a complete system design. The design starts with a system specification that may be non-executable and produces an executable specification consisting of interacting modules (a multithread description). Then several design decisions and refinement steps must be undertaken, the objective being the construction

multiprocessor. In the case of a monoprocessor the main synthesis step involves code generation for the processor. A partitioning step may be needed when parts of the description have to be executed on specific hardware operators. Hardware operators are hardware devices designed specifically for a repetitive function that must be executed efficiently to meet some timing constraints; for example a software implementation of a filtering function may be unacceptably slow in a DSP (Digital Signal Processing) application. A scheduling step is needed for multithread descriptions, where the execution of several processes must be scheduled on the main processor. In multiprocessor architectures, a processor may be pure software (code executed on an existing processor), pure hardware, or a mixed model. For a single-thread input description, the main steps are partitioning and processor design. Partitioning distributes the initial description into communicating modules that are mapped on separate processors. Performance estimation must guide the whole process and if the requirements and constraints are not met then another design iteration must be undertaken. A multithread input description requires a communication refinement step, where the input description language's communication models and synchronization schemes are mapped onto communication units. These units may correspond to processors yet to be designed or existing subsystems.

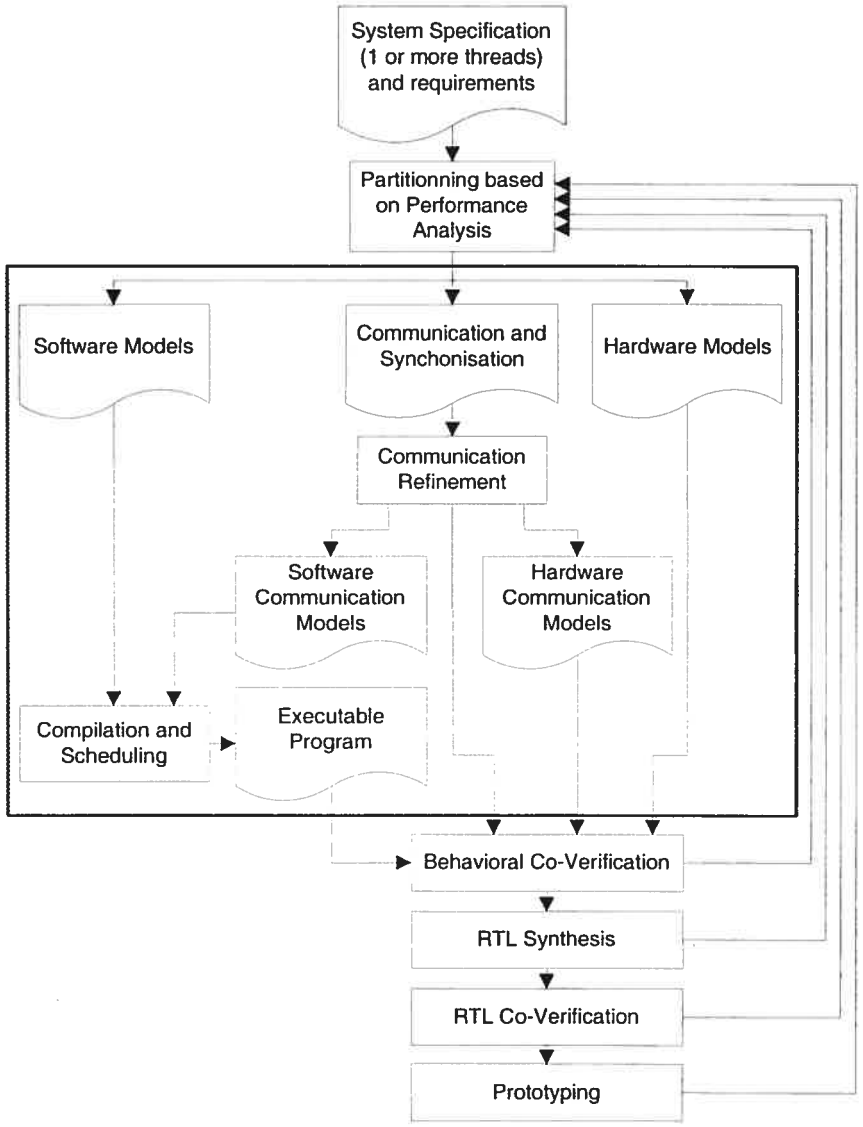
According to [2], three dilemmas face SOC designers: (1) Complexity is increasing while the time to market of consumer products is decreasing, the solution is reusing previously designed functional blocks wherever possible. These blocks are referred to as macros, cores, intellectual property (IP), and VCs (virtual components). (2) The ASIC suppliers do not usually have design expertise in all the diverse areas needed for a given SOC. (3) Reuse is difficult because SOC designers must contend with a

at the specification level. We will illustrate the similarity of problems and solutions between the design patterns methodology and IP reuse in hardware. Both design patterns and IP blocks are meant to encapsulate good-quality design experiences that are proven useful in successful projects; and both present similar challenges due to the complexity of interaction management and a wide range of provenance and platforms. This chapter can be seen as a bridge between similar methodologies in hardware and programming domains. Section 6.2 describes the desirable characteristics of a design environment; Section 6.3 describes the implementation languages for hardware and HW/SW description; Section 6.4 introduces the commonality and variation concepts; Section 6.5 deals with design reuse and hardware libraries; Section 6.6 discusses configuration and variation issues; Section 6.7 concentrates on the use of design patterns for system level modelling, and reuse; and finally, Section 6.8 concludes this work.

6.2 Characteristics of a Design Environment

In our opinion, the *desirable* characteristics a HW/SW designer is looking for are as follows:

- **Minimize the number of environments and languages to deal with.** This should not be achieved at the expense of the expressivity at different levels of abstraction. We should be able to express design requirements and constraints related to timing, testability, number of processors, protocols of communication between subsystems, etc. These environments should allow automatic refinements (synthesis) and development of methodologies, such as usage of design patterns, that go beyond the basic concepts of HDLs (hardware description lan-


System Specification (1 or more threads) and requirements
Partitionning based on Performance Analysis
Software Models
Communication and Synchonisation
Hardware Models
Communication Refinement
Software Communication Models
Hardware Communication Models
Compilation and Scheduling
Executable Program
Behavioral Co-Verification
RTL Synthesis
RTL Co-Verification
Prototyping

of smart code like STL (Standard Template Library) [3], windowing libraries [4] will help in development of test-benches. Some HDLs are very limited in the management of concurrency and scheduling of events [5]. These limited capabilities seem sufficient for describing interaction between hardware components. However they are very inefficient when developing test-benches. Test-benches and abstract architectures necessitate FIFOs, queues and stacks for example. Moreover, the full power of a multi-paradigm language can be used to develop high-level test generation and analysis tools.

- **Link to the outside world and extension of the capabilities of the environment.** HDLs support by Open source toolsets, help in developing other tools around the core language creating interoperable environments (for design pattern applications in this domain see [6]).

- **Documenting tools for the models.** The models would be more usable and more resilient, if well documented. In some Open source environnements, the models are developed using the same paradigms as those used to develop the HDL itself. GNU tools like Doxygen [7] may be used to both document the HDL as well as the models developed with that HDL [8]. Graphical documentation can also be easily obtained using UML [9].

- **Environments facilitating reuse are necessary given the complexity of the designs.** Hierarchy is one of the mechanisms that allow reuse. Families of design are another mean for obtaining reuse: often a design house will be interested in developing a family of products such as processors, controllers, and busses. All express some commonality but inside the same family, it is desirable

6.3 Implementation languages

In order to illustrate the different needs in HW/SW modelling, we will use mainly SystemC; we will also mention VHDL because it is well known to hardware designers. SystemC is a modelling platform consisting of a C++ class libraries and a simulation kernel for design at the system/behavioural and register-transfer levels, it brings the possibility of modelling and simulating parallel and collaborative entities. Designers create models using SystemC and standard ANSI C++ [10]. Different models of computation and design methodologies may be used in conjunction with SystemC. The design libraries and models needed to support these specific design methodologies are considered to be separate from the SystemC core language standard. By its nature, SystemC inherits the capabilities of C++, like the support of multiple paradigms: classes, overloaded functions, templates, modules, ordinary procedural programming, and others. The freedom afforded by these capabilities may also bring new difficulties to the understanding of the model or the ability to synthesize it. We will demonstrate the benefits of this multi-paradigm environment, and show that the advantages outweigh the risks. We will show that we are able to develop more resilient specifications and models, express different behaviours and view of the same "entity", facilitating design exploration and ease specification and development of libraries.

We will illustrate this by comparing SystemC capabilities with those of the VHDL hardware description language. The lack of some useful paradigms in VHDL has already been described as early as 1991 as summarized in [11]. Early attempts tried to augment VHDL with other constructs, but since systems have more and more software programmable components, many designers are leaning toward a common language for hardware and software modelling. We will explore hardware modelling both at

processes. At the behaviour level, the system is seen as a function with inputs and outputs, no timing or interaction is specified. At RT level, the system is seen as a collection of registers connected by arithmetic or logic units, the timing is fixed and registers are controlled by clocks.

In this study, we will use the concept of *commonality* and *variation* developed by Coplien [13, 14] to illustrate the characteristics of modelling, reuse and specification.

6.3.1 RTL and behavioural modelling

In September 1999 SystemC was announced; prior to that date two main languages, Verilog and VHDL, dominated the modelling activity of digital circuits. The three languages provide a set of similar modelling constructs at RTL and behavioural levels of abstraction. Users can construct structural designs using modules, ports, and signals. Modules can be instantiated within other modules, enabling structural design hierarchies to be built. Ports and signals enable communication of data between modules, and all ports and signals are declared by the user to have a specific data type. Commonly used data types include single bits, bit vectors, multi-valued logic, characters, integers, floating point numbers, vectors of integers, etc. SystemC offers also the fixed-point type to model fixed-point numbers in digital signal processing applications.

In VHDL and SystemC, concurrent behaviours are modelled using processes. In Verilog, concurrent behaviours are modelled using "always" blocks and continuous assignments. A process can be thought of as an independent thread of control, which resumes execution when some set of events occur or some signals change, and then suspends execution after performing some action. Since processes execute concurrently and may suspend and resume execution at user specified points, process instances gen-

formance improvements, especially when the number of process instances in a design is large.

In a model, hardware signals allow the exchange of information between concurrent processes. In different HDLs, it is desirable that they are initialised to a specific value easily recognizable as unknown state. They may have multiple drivers (multiple processes try to update their value at the same time). In this case a function is needed to compute a resolved value based on each of the driving values. This function must automatically be called when any of the driving values changes. Resolution functions are well developed in VHDL, but are very limited in SystemC and Verilog. Finally, hardware signals do not immediately change their output value when they are assigned a new value, either in simulation or in the real world. The time in a simulation environment can be seen as a two-dimensional space: the horizontal axis representing the discrete time and the vertical axis representing what is called the delta time. As long as there are causal events that change signals, time progresses on the vertical axis (Delta) without any change on the discrete time axis. When a signal assignment occurs, other processes do not see the newly assigned value until the next delta cycle. Processes that are sensitive to the signal resume execution if the signal value is changed with respect to its previous value. This delay is crucial to proper modelling of hardware, since it allows for example two registers to swap values on the same clock edge. In comparison two software variables cannot swap values without the introduction of a third temporary variable.

6.3.2 System level modelling

One of the primary goals of SystemC is to enable system level modelling, including

foundation, termed the "core language" (kernel language and data types, Figure 6.2), has been added to the language. On top of the "core language" more specific models of computation, design libraries (e.g. timers, FIFOs, signals, etc.), modelling guidelines, and other design methodologies can be added. This can be achieved using a layered approach (Figure 6.2).

The hardware signal as a mechanism for communication and synchronization is not sufficiently general for system level modelling. For example, in a system level design, a designer might want to specify that several modules communicate using queues, or that several processes execute concurrently and manage access to shared global data. New set of features for generalized modelling of communication and synchronization are introduced: channels, interfaces, and events. A channel is an object, which serves as a container for communication and synchronization. Channels implement one or more interfaces. An interface specifies a set of access methods to be implemented within a channel, but the interface itself does not provide the implementation. An event is a flexible, low-level synchronization primitive that is used to construct other forms of synchronization. Channels, interfaces, and events enable designers to model the wide range of communication and synchronization found in system designs. Examples include HW signals, queues (FIFO, LIFO, message queues, etc., semaphores, memories, and busses (both as RTL and transaction based models).

A model of computation can be defined by the following:

- The model of time employed (real-valued, integer-valued, untimed) and the event ordering constraints within the system (globally ordered, partially ordered, unordered)

- The methods of communication between concurrent processes

required functionality is already present in the simulation kernel. Even if the global model of time is fixed to an integer model, designers can construct specific channels to achieve their precise rules for communication between processes, and process activation. Note that presently, a continuous time model is not supported, limiting SystemC to discrete time systems.

<table>
<tr><td>Standard Channels
Various Models of Computation
(Kahn Process, Networks,
Static Dataflow, etc.)</td><td>Methodology-Specific Channels
(Master/Slave Library, etc.)</td></tr>
<tr><td colspan="2">Elementary Channels
(Signal, Timer, Mutex, Semaphore, FIFO, etc.)</td></tr>
<tr><td>Kernel Language
(Modules, Ports, Processes, Interfaces,
Channels, Events, etc.)</td><td>Data Types
(Intergers, Logic Type (01XZ),
Logic Vectors, Bits and Bit Vectors,
Arbitrary Precision)</td></tr>
<tr><td colspan="2">C++ Language Standard</td></tr>
</table>

Figure 6.2: *Layered approach to augment SystemC capabilities*

6.4 Commonality and Variation in VHDL

Commonality and variation can be traced back to the concept of software families [15]. Families are collections of software elements related by their commonalities, with individual family members differentiated by their variations. According to Coplien [13, 14], commonality and variability analysis are two key characterizations of most software paradigms. Commonality analysis brings three desirable characteristics:

1. it tackles complexity by supporting abstraction (inheritance hierarchy is a good

3. it reduces maintenance cost [14, 13].

Variability makes sense only in a given commonality frame of reference. It expresses different ways of variation (special, structural or temporal of a basic abstraction). Commonality and variability analysis depend closely on code implementation, this explains the presence of code in the following section. On the other hand, design patterns represent higher-level paradigms, which are independent of the implementation language. Hardware designers are under pressure to develop new design derivatives in less time. In the case of software systems, the benefits of explicitly identifying the common and variable aspects of the different versions of a system are presented in [13]. VHDL [5] allows the expression of commonality by what is called design units. A design unit is a VHDL construct that may be independently analysed and inserted in a design library. These design units are:

- **Entity declaration:** describes the interface view of a component (like a Data Book description). It is implementation independent.
- **Architecture body:** describes an implementation of an entity (like a single schematic diagram). A single interface may have alternative architectures.
- **Package (declaration and body):** contains information common to many design units. This information consists of functions, types, signals, and constants. It hides details, simplifies design, and may invoke other packages.
- **Configuration:** relates local entity and architecture references to actual units in libraries (like a parts reference list).

Commonality can also be expressed by generate statements and generic constants for

and subprograms in coordination with packages. Overloading allows the reuse of models even if the basic data types are changed, like going from a bit type to a 9-valued standard logic. Commonality between processes is very limited.

In SystemC 2.0, at the system level, abstract classes such as interfaces express commonality; variation is obtained from interfaces by possibly using single or multiple inheritances.

6.5 Design Reuse and Hardware Libraries

As seen before, the main mechanism for design reuse in VHDL is libraries of design units. Structural hierarchy is the way to reuse components declared in a library. SystemC has the same capability as VHDL and also other mechanisms, based especially on inheritance, templates and overloading.

6.5.1 Module Inheritance in SystemC

In designing a library of gates, latches,..., we can specify all the properties related to gates in a module `Gate` from which, `And`, `Or`, `Xor`, ..., will be derived later. The purpose of the `Gate` module is to act as an interface. All the properties and methods for Flip-Flops will be captured in a similar module (Setup and hold time, delays, etc.). This is an example of simple inheritance of a module:

```
1  class Gate : public sc_module
2  {
3    public :
4    Gate(const sc_module_name& name): sc_module(name)
5      { (...) }
6  };
7
8  class And : public Gate
```

6.5.2 Using Inheritance to Insert Tags or Attributes

In VHDL, we can reuse a component with a variation of behaviour or semantics based on attributes. We can, for example, use attributes to specify that a process is at the RTL or behavioural level. However, it is impossible to derive processes from previous ones, or specialize their methods. If we look at modern developement tools such as Java, some major benefits come from the usage of a common root class. Even if there is barely nothing in this class, it ensures that all objects are treated in an uniform way. Unique root is unnecessary when commonality between a set of specific objects is absent (see the FSM (Finite State Machine) pattern implementation example, Figure 6.8 and Figure 6.9 in Section 6.7.3). In SystemC, `sc_object` constitutes such a root for the core of the library. For the purpose of modelling a library of components and IPs, `sc_module` can be the building block for the models. From this we may derive other specialized modules, as described in Figure 6.3. We can then use these basic

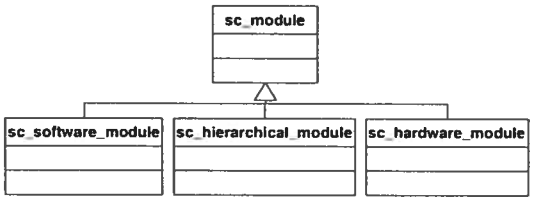


Figure 6.3: *Module specialization*

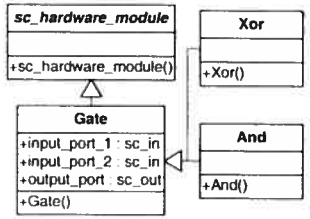


Figure 6.4: *Behavioural hierarchy*

building blocks to construct our libraries; Figure 6.3 will be enhanced in Section 6.7.2 describing the use of the compositional pattern. With this kind of construct, we can specify how the module should be instantiated. Because all three modules inherit from the same `sc_module`, they have the same behaviour (unless we choose otherwise). A designer could decide that a particular software module should be implemented in hardware, by simply changing the ancestor of his design. This kind of construction could be recognized by synthesis analysers without having to alter SystemC syntax or behaviour, but is part of the static configuration. Changing inheritance is simple

could also implement a variable in a base class that would choose between software or hardware implementation but then the following structure would then appear repetitively in the design objects :

```
1  #define STRUCTURAL 0   //(or better, a class enum...)
2  #define SOFTWARE   1
3  #define HARDWARE   2
4
5  some_class::some_method(...)
6  {
7    switch(module_type)
8    {
9      case STRUCTURAL :
10       (...)
11       break;
12
13     case SOFTWARE :
14       (...)
15       break;
16
17     case HARDWARE :
18       (...)
19       break;
20   (...)
```

We will see later (Section 6.7.2) how to make this kind of choices more dynamic.

6.5.3 Hierarchical Module Construction Hardware Libraries

Class hierarchies can be of great help when designing hardware libraries. Here are some proven advantages:

- **Modularity of structure:** construction of hardware properties are reused in derived classes.
- **Locality of code:** future possible modifications are often limited to a class or its ancestor(s).

Figure 6.4 represents a basic gate library. There is no need in the "child" class to declare again the port information (this is impossible in VHDL), all we need to do is to implement a sensitive process to these ports and have these methods set sensitive in each of the constructors.

```
1  And::And(...) : public Gate(...)
2  {
3    SC_METHOD(main_process)
4    sensitive << input_port_1 << input_port_2;
5  }
```

This duplication of definition can be avoided. As an illustration, in Figure 6.5, if the virtual `init()` method defined in the Processor class is sensitive to some signals, then the derived init methods in classes such as `DLX_Processor` will inherit this sensitivity. Therefore, adding and refining the hardware components could result in the construction of a library from ground up.

6.6 Variation and Configuration

6.6.1 Multiple Architectures

Using VHDL, we can have multiple architectures related to the same entity. We can consider the entity as the abstract class and different architecture as the derived classes from the basic abstract class. Note that in VHDL, this derivation process is limited to two levels of abstractions. In SystemC, this refinement process can continue indefinitely as illustrated in Figure 6.5.

6.6.2 Generation and Configuration of Regular Structures

VHDL provides a construct called

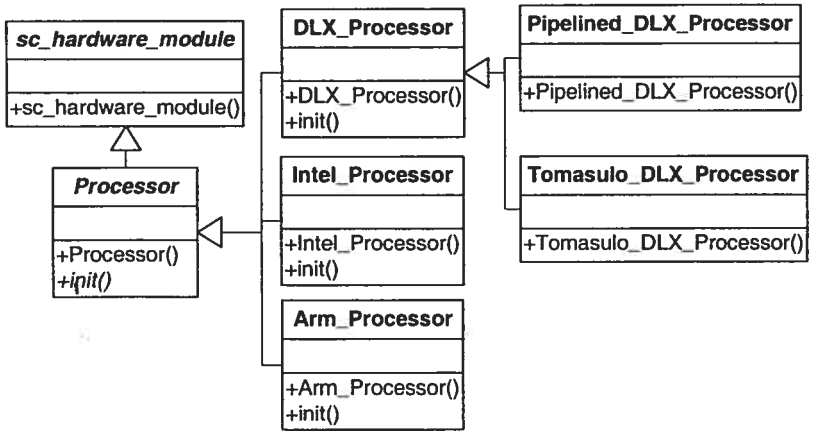


Figure 6.5: *Behavioural refinement*

```
2   processor_array[0]=new Arm_Processor();
3   processor_array[1]=new DLX_Processor();
4   processor_array[2]=new Intel_Processor();
```

The configuration could be put in a file and then read at execution time during elaboration. A switch statement could then be used to instantiate the different processors. This has no correspondence in VHDL:

```
1   int i = 0;
2   FILE input = fopen("cpu.cnf", "rt");
3   while (!eof(input))
4   {
5     char buffer[500];
6     buffer = fgets(input);
7     switch(atoi(buffer))
8     {
9       case ARM :
10        processor_array[i++] = new Arm_Processor();
11        break;
12      case INTEL :
13        processor_array[i++] = new Intel_Processor();
14        break;
15      (...)
```

and `Arm_processor`. If general enough, these methods are reused in the third level of refinement without having to redefine them. Modularity of code is very important to achieve reusability. A module, which uses these processors, can be written using only the abstract class processor. Depending on the true instance of the processor the right init method will be invoked by polymorphism:

```
1  for (int i = 0; i < 10; i++)
2    //initialize all processors
3    processor[i]->init();
```

If we desire the same effect in VHDL we would need a special signal `Reset_Processor`, which should feed only processor components. This is less elegant and less readable than what we have here. It is also surely more time consuming during simulation because events on signals put a very heavy burden on the VHDL simulator.

6.6.4 Using Overloading Mechanism to Change a Behaviour

In VHDL, overloading is limited to functions and procedures. In SystemC, not only methods but also constructors of classes can be overloaded allowing more dynamic configuration of threads and modules. To determine the correct method to call, the compiler only looks at the type of the parameter(s) when the method call is issued. Here is an example of overloaded constructor for a Processor object:

```
1  Processor::Processor(const sc_module_name &name, int bus_format)
2   : sc_hardware_module(name)
3  {
4    bus = new sc_in<int>();
5    (...)
6  }
7
8  Processor::Processor(const sc_module_name &name, char bus_format)
9  : sc_hardware_module(name)
10 {
11   bus = new sc_in<char>();
12   (...)
```

6.6.5 Using Overloading to Speedup Simulation or Specify a Particular Behavior

In VHDL, if we had to specify a generic entity with n ports, we first define a generic and then declare an array of n ports. However, if for a specific number of ports we have a particular behaviour or any optimisation, the only way is to test the parameter inside the architecture related to the generic entity. In SystemC, by using overloading we can achieve this in a more efficient way as illustrated below:

```
1  void And::compute(void);
2  {
3    output_port = input_port_1 && input_port_2;
4  }
5
6  void And::compute(int n_ports);
7  {
8    if (n_ports != 0)
9      for (int i = 0; i < n_ports; i++)
10       (...)
11 }
```

6.6.6 Using Templates to Describe Regular Behaviour and its Elaboration at Compilation Time

Interconnection networks or other behaviours can be easily described recursively. We will show that by using templates [16], we can have very abstract descriptions translated in an iterative behaviour at compile time, hiding these abstractions from the simulator, and potentially the synthesis tool. This can be described as a new design pattern: *meta-template pattern.*

```
1  template<int n_ports>
2  class Gate : public sc_module
3  {
4    public :
5    sc_in<bool> input_ports[n_ports];
6    sc_out<bool> output_port;
```

```
14    static inline void compute(bool &result, sc_in<bool> *input_ports)
15    {
16      Compute<N-1>::compute(result, input_ports);
17      result = result & input_ports[N-1];
18    }
19  };
20
21  class Compute<2>
22  {
23    public:
24    static inline void compute(bool &result, sc_in<bool> *input_ports)
25    {
26      result = input_ports[0] & input_ports[1];
27    }
28  };
29
30  template<int n_ports>
31  class And : public Gate<n_ports>
32  {
33    public :
34    SC_HAS_PROCESS(And);
35    void compute_process(void)
36    {
37      bool result;
38
39      while(1)
40      {
41        Compute<n_ports>::compute(result, input_ports);
42        output_port = result;
43        wait();
44      }
45    }
46    And(const sc_module_name& name)
47      : Gate<n_ports>(name)
48    {
49      SC_THREAD(compute_process);
50      for (int i = 0; i != n_ports; i++)
51      sensitive << input_ports[i];
52    }
53  };
```

Templates are resolved at compilation time. The instantiation of a specific And gate can be done either by:

The compilation of the following code

```
1  while(1)
2  {
3    Compute<n_ports>::compute(result, input_ports);
4    output_port = result;
5    wait();
6  }
```

would result in the following iterative hardware structure:

```
1  while(1)
2  {
3    result = input_ports[0] & input_ports[1];
4    result = result & input_ports[2];
5    result = result & input_ports[3];
6    (...)
7    result = result & input_ports[9];
8    output_port = result;
9    wait();
10 }
```

This is because the compiler unrolls the template inlined method calls. This allows the compiler to perform very useful optimisations. The drawback is that this is being part of the static configuration (like in VHDL) where a configuration modification implies that you must change your code and then recompile.

6.7 Use of Design Patterns

Coplien [14] states that:

> "Patterns often express what programming languages can not directly express. Whereas multi-paradigm design builds solutions only at the level of language features, patterns are much

He sees them as complementary to a multi-paradigm approach and they should be introduced where the multi-paradigm approach is of less use (negative variability). Originally, design patterns [17] are meant to identify, document, and catalogue successful solutions to common software problems. Patterns aid the development of reusable software by expressing the structure and collaboration of components to developers at a level higher than source code or object-oriented design models that focus on individual objects and classes. Their use can be extended to development of models for hardware components. It has been shown [18] that several design patterns from the software pattern catalogue [17] can be easily applied to the design of a hardware data path library in C++. Four patterns have been applied in that study: Composite, Object Adaptor, Abstract Factory and Decorator. The usability of those patterns is demonstrated by the design of a synchronizer for a mobile radio system. The design is compared to an equivalent VHDL design. The results show that the C++ design has half the code size compared to the VHDL design. In this chapter, we illustrate the use of two existing patterns: singleton and composite ones. We, then, introduce two new ones: the FSM and the meta-template patterns. A traffic light controller will serve as an implementation of the FSM pattern; while Section 6.6.6 is a possible implementation of the meta-template pattern. The documentation of these two new patterns follows the format suggested in [17]. The list of the pattern that can be used or created in HW/SW modelling is much larger that what is presented here. The four templates are used only as an illustration of the possibilities of the design pattern methodology.

6.7.1 The Singleton Pattern

The main idea of the solution is to "hide" the constructor from public access, then provide a public static method to control the instantiation of the class.

One way of using this approach when dealing with configurability (Figure 6.6), is to have a base class decide which derived class is best suitable to instantiate.

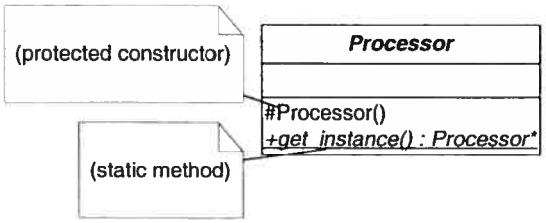


Figure 6.6: *A configurable uni-processor system*

```
1   #define UNKOWN  0  //(or better, a class enum...)
2   #define ARM     1
3   #define INTEL   2
4
5   Processor *Processor::get_instance(int type)
6   {
7     Processor *new_processor_instance;
8     switch(type)
9     {
10      case ARM :
11        new_processor_instance = new Arm_Processor();
12        break;
13      case INTEL :
14        new_processor_instance = new Intel_Processor();
15        break;
16      (...)
17    }
18    return new_processor_instance;
19  }
```

The problem with this kind of approach is that usually, it is the derived class that is "aware" of what is happening in the base class while the base class knows nothing

6.7.2 Composition Pattern

This pattern naturally describes hierarchical designs in hardware (see Figure 6.7). This kind of construct allows a certain kind of object and its children to be treated the same uniform way (`mixed_module` in our example).

`structural_module` has inside a list of `mixed_modules` and implements the addition and removal of those modules (or child modules).

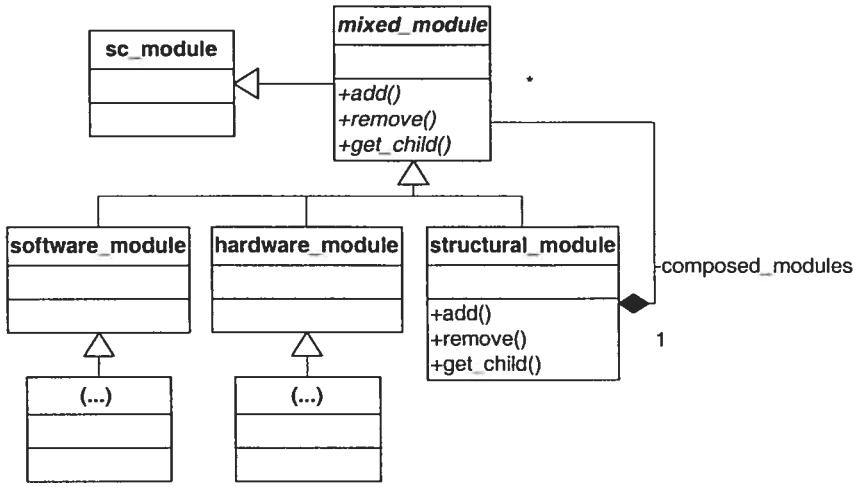


Figure 6.7: *Composition Pattern To Describe Structural Hierarchy*

6.7.3 Abstract Factory and FSM Patterns

FSMs are ubiquitous in describing protocols, processors and controllers. Generally processors are seen as control FSMs and data-paths. Abstract Factory Pattern allows the reuse and specialization of methods to generate such FSMs.

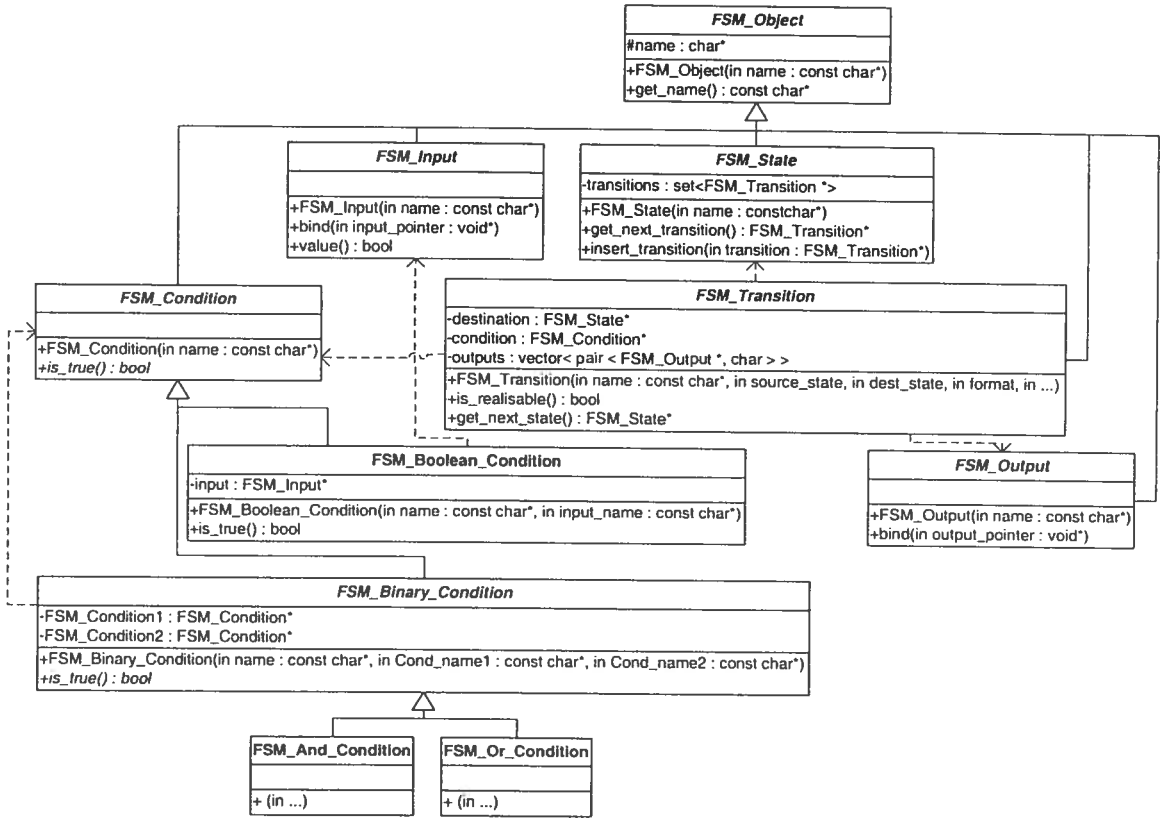


Figure 6.8: *FSM base objects interface needed to describe an FSM machine*

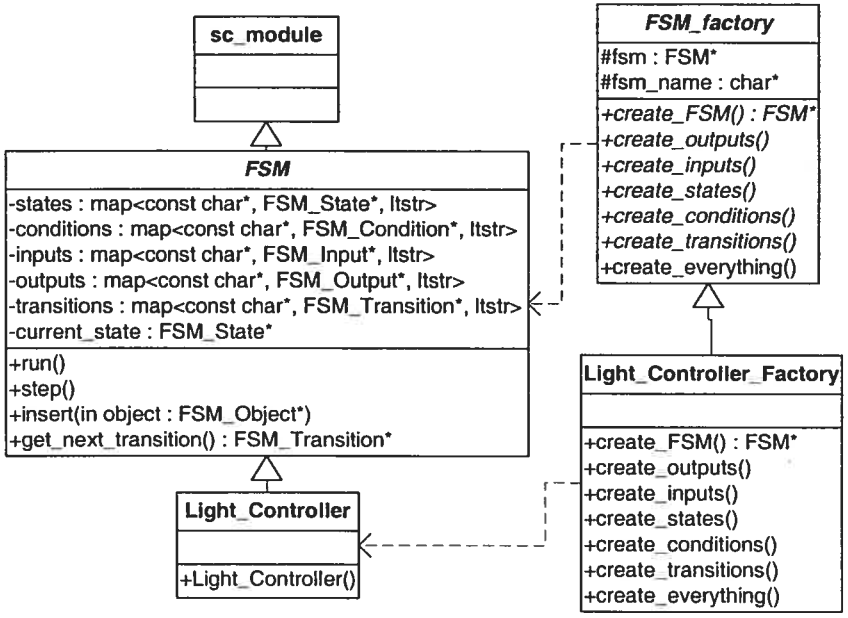


Figure 6.9: *FSM factory and the FSM machine using the abstract factory and builder pattern*

Also known as

Protocol describer, control describer.

Motivation

The purpose of this pattern is to have a flexible way of describing a finite state machine while keeping this methodology as standard as possible. This pattern could also be used as an illustration of the abstracted factory and builder patterns for it is based on both of them. While it sets a standard way for describing FSM rules, the methodology for using these structures (Figure 6.8) are then intuitive. A standard resolution engine (i.e.: which transition to take at each step...) can then be supplied in the base classes

- There is a need to implement an FSM (protocol or simple machine) and also a need to change its structure or behaviour dynamically.
- More than two similar FSMs (protocols) have to be implemented and reuse of the same methodology is desirable.
- Different versions of the same FSM (protocol) have to be explored while minimizing the change to the code.

Structure

Since the FSM example we used to describe this pattern (Figure 6.8 and Figure 6.9) is general enough and since we used an object oriented approach to describe the communality of the solution, we can use these figures, practically inchanged as the generic class structure scheme. The common classes at the root of this scheme can be specialized as needed. In our example, a `FSM_boolean_condition` class was created because we needed to check the combined predicate value of two conditions with a logical *AND* operation. This should be considered as an example to show that any FSM (based on `FSM_objects`), as particular as can be imagined, can be conceived without altering the basic interface scheme already available. Also, if necessary, one could decide to create template specialized classes, but one should be careful not to bring the templates too early in the root for it is hard then to type-cast back into basic types.

Participants

`FSM_Object`: This is the base class of all other FSM components. It is not mandatory

`FSM_State`: It is the basis component of the state machine. It permit to declare every possible state in your system by establishing a relation between the state instance, its name, and the relation with other FSM objects.

`FSM_Input`: This class plays the role of an adapter, it is an interface with the external world of the FSM machine. If well modelled, it should accept any kind of inputs (and maybe transform or buffer them) into data the FSM objects could deal with.

`FSM_Output`: It has basically the same kind of functionality as the `FSM_Input`. The data which has been generated by the FSM must then be analysed with this class to create (transform or buffer) to the external world according to the given output format.

`FSM_Condition`: This class models facts. Facts can either be true or false according to the inputs. An `FSM_Condition` is associated to a transition; the transition can be taken only if the condition evaluates to TRUE.

`FSM_Transition`: Transitions between states are the backbones of the FSM machine and must be well defined. This class defines transition in term of origin state, destination state, condition needed to be realized and finally outputs to be generated once the transition is taken.

`FSM`: An abstract class that is in fact a builder for the FSM machine. It is intended to be used in conjunction with `FSM_Factory` following the factory methodology. This class is the interface to the class that should act as the container of the actual implementation of the `FSM` or as a "wrapper".

Collaboration

As specified before, the abstract factory is used to create a specialized version of the `FSM` machine while maintaining sufficient generality to be flexible and reusable. By inheritance and polymorphism, all the creation is delegated to the specialized child (in Figure 6.10, it is `Light_Controller_Factory` who is responsible for the specialized creation) while the `FSM_factory` at the root establishes a standard way of creating objects.

This new specialized class can then choose its implementation of the appropriate `FSM` instance, input, output, condition, states, etc. For the communication part (see

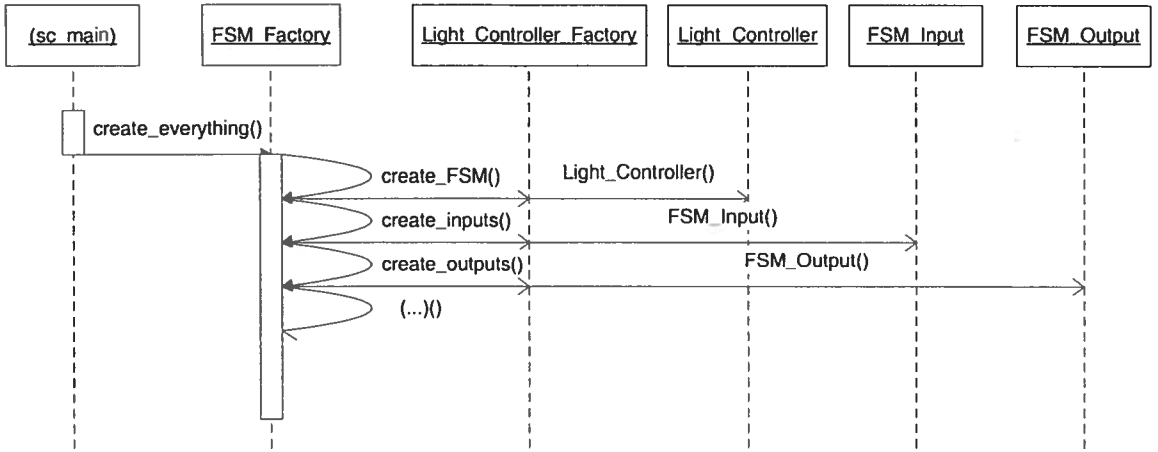


Figure 6.10: *FSM factory creation sequence*

Figure 6.11), our `sc_main` (which is SystemC's main) can call a run method into the `Light_Controller` FSM. This method is defined in the base class but can be modified if necessary in specialized classes. At each clock cycle, the `step()` method is called asking for a new state. The method `get_next_transition()` of each transition of

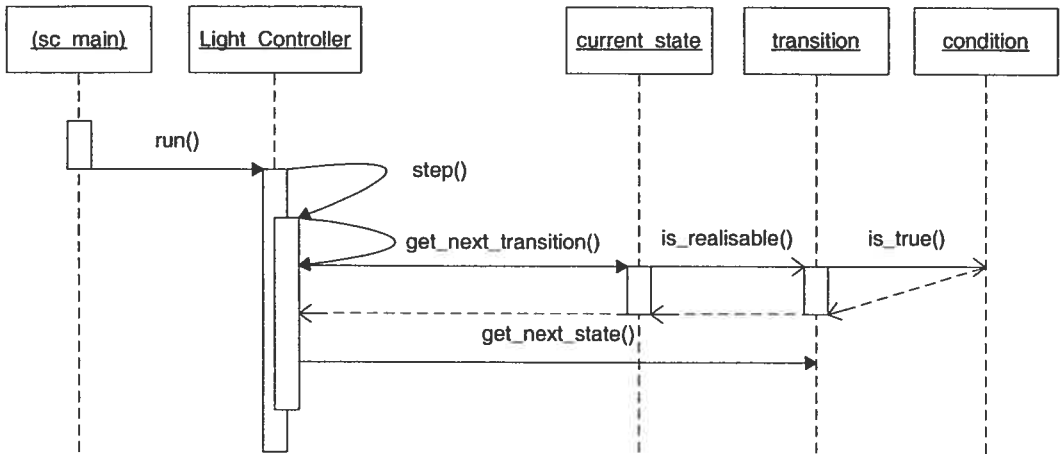


Figure 6.11: *FSM single step sequence*

Consequence

1. Configuration is dynamic : During run time, one could decide that a particular transition is inappropriate, pause the simulation, disable the transition and then resume execution. Only the first construction of the whole FSM is static in our example, but nothing prohibits from having a FSM created from database information or from user's GUI inputs.

2. Uniformity : Every object in the FSM is standard and can be treated the same uniform way.

3. User friendly : FSM can be created in a human readable way. In our example, we have used character strings to increase readability and because these strings can then be converted to pointers, they will not slowdown the FSM execution.

4. Extendable : Because it is based on abstract factory pattern, the creation of

6. May be slower than hand coded FSM : there are ways to write such FSM more directly using straight conventional C code, however reuse will be diminished.

Implementation

In our implementation, we chose to use text strings as common linking device between FSM objects. This choice is quite logical since it is more understandable to the designer than pointers or obscure index numbers. After all the links have been well established, these strings are no longer required so they are discarded to be replaced by pointers to gain more speed during simulation.

We have also used SystemC as a driver for the input, output signals and for clocking the FSM's change. It was then very useful for us to derive our FSM abstract object from `sc_module` but we could have done the inverse without affecting the result. In other words, this means that if you take the `FSM` as the base class and specialize it in a `SC_FSM`, at the end, you get the same result. This means our `FSM` could be implemented independently from SystemC and could serve as another computational model.

These are the default run methods that handle the main process. Notice they can be redefined in later derived `FSM` class. As mentioned previously, these have some SystemC syntax but it could be implemented otherwise.

```
1  void FSM::run(void)
2  {
3    while(1)
4    {
5      wait();
6      step();
7    }
8  }
9
10 void FSM::step(void)
11 {
12   current_state = current_state->get_next_transition()->get_next_state();
```

```
2   {
3     for (set<FSM_Transition *>::iterator i = transitions.begin();
4          i != transitions.end(); i++)
5     {
6       if ((*i)->is_realisable())
7         return (*i);
8     }
9     return NULL;
10  }
```

By default, the "possibility of taking" of the transition is defined as follows

```
1   bool FSM_Transition::is_realisable (void)
2   {
3     return condition->is_true();
4   }
```

Known uses

FSMs are used in to describe telecommunications protocols, small devices and embedded systems, automatic vending machines, etc.

Related Patterns

This pattern uses the Abstract Factory to create the concrete FSM and the builder pattern to break the tedious process of FSM creation in small parts. Adapter patterns can also be used in the creation of output and input class.

6.7.5 The Meta Template Pattern Documentation

Intent

Transforms a complex parametric procedure into a more efficient one by fixing some parameters at compilation time and using static configuration.

Motivation

This pattern is not really based on object interrelations but rather on a parametrized class acting as a container for some method that can be converted directly by the compiler to allow a faster execution time; however the constructs output by the compiler may not be very flexible for dynamic changes.

Applicability

Use this pattern when:

- There is a procedure and a predefined number of iterations or constant parameters and the procedure must be accelerated at execution time.
- There is a complex computation that can be reduced to constant value during compilation time.
- There is a loop that needs to be unrolled and the number of iterations is known at compilation time.
- There is a complex structure to build (with nodes and links) but it needs to be static, not dynamic, for a speed or implementation purposes.

Structure

Figure 6.12 shows the structure of the Meta Template pattern. In this kind of pattern, the user must create two different classes. A generic class which holds the generic method and an overloaded class.

It is very important to see that the "generic class" and the "overloaded class" are all

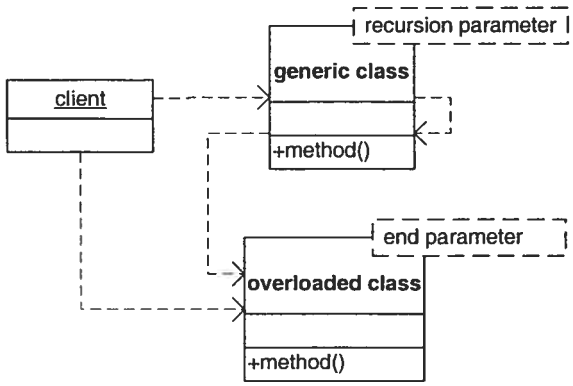


Figure 6.12: *Generic Structure of the Meta Template*

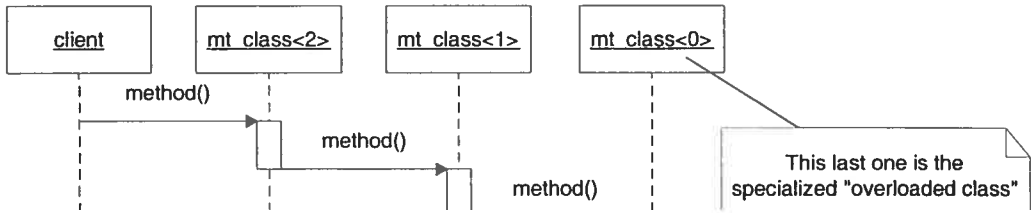
Participants

generic class: instance of the template class holding the generic form of the recursive or iterative method.

overloaded class: overloaded instance of the template class, holding the method containing specific code for the base of the recursion.

Collaboration

Basically, there is no dynamic collaboration. The following sequence diagram represents the compiler transformations prior to execution time. Everything should be resolved statically.



Consequence

The Meta Template pattern has the following benefits and liabilities:

1. Configuration is static: it is the compiler that completes the computation associated to the constant value, this value will be used during execution.
2. Compilation time may increase due to the static pre-computations.
3. Execution time is faster: Since the C++ compiler resolves the problem, often the results of a long mathematical expression is replaced by a single constant during execution or hardware implementation.
4. Recompilation may be needed more often: since the configuration is static, it means that any change in the constant part of the procedure or the (one of the) constant parameter(s) will result in a re-compilation.
5. Might not be portable on every language: templates are mainly a C++ paradigm.

Implementation

(see Section 6.6.6)

Known uses

The pattern itself is new so there is no known uses at the time of the writing of this document, but the methodology was first described in [16] where conversion of sine calculations into constants and other mathematical application were depicted.

Related Patterns

6.8 Conclusion

In this chapter we addressed System On Chip modelling in order to minimize the impact of the increasing complexity and the shrinking of the time to market of consumer products. We showed how to increase reuse at the modelling level by using multi-paradigms existing in languages such as C++ as well as higher paradigms such as design patterns. These modelling techniques are in their infancy when considering HDLs. We illustrated the similarity of problems and solutions between the design patterns methodology and IP reuse in hardware.

Bibliography

[1] Tarek Ben Ismail and Ahmed Amine Jerraya. Synthesis steps and design models for codesign. *Computer*, 28(2):44–52, 1995.

[2] Mark Birnbaum and Howard Sachs. How VSIA answers the SOC dilemma. *Computer*, 32(6):42–50, 1999.

[3] Silicon Graphics Computer Systems Inc. Standard template library programmer's guide, 2001 1993. `http://www.sgi.com/tech/stl`.

[4] Trolltech AS. Qt on-line reference documentation, 2001 1996. `http://doc.trolltech.com`.

[5] *IEEE Standard VHDL Language Reference Manual.* IEEE, 1076,2000 edition, 2000.

[6] Luc Charest, Michel Reid, E.Mostapha Aboulhamid, and Guy Bois. A methodology for interfacing open source SystemC with a third party software. In *Design Automation and Test in Europe Conference & Exhibition*, pages 16–20, Munich, Germany, 2001. IEEE Computer Society.

[9] Grady Booch, Ivar Jacobson, and James Rumbaugh. *The Unified Modeling Language User Guide 1/e.* Addison Wesley, 1999.

[10] Open SystemC Initiative (OSCI). Functional specification for SystemC 2.0, 2001. `http://www.systemc.org`.

[11] Sowmitri Swamy, Arthur Molin, and Burt Covnot. OO-VHDL: Object-oriented extensions to VHDL. *Computer*, 28(10):18–26, 1995. 0018-9162/95/$04.00 (c) 1995 IEEE Features.

[12] D. Verkest, J. Kunkel, and F. Schirrmeister. System level design using C++. In *Design, Automation and Test in Europe*, Paris, France, 2000.

[13] James Coplien, Daniel Hoffman, and David Weiss. Commonality and variability in software engineering. *IEEE Software*, 15(6):37–45, 1998. James O. Coplien, Daniel M. Hoffman, and David M. Weiss. Commonality and Variability in Software Engineering.

[14] J. Coplien. *Multi-Paradigm Design for C++.* Addison-Wesley, Reading, MA, 1999.

[15] D. L. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, SE-2:1–9, 1976.

[16] T. Veldhuizen. Using C++ template metaprograms. *C++ Report*, 4(4):36–43, 1995.

[17] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, 1994.

Chapitre 7

Using Design Pattern for Type Unification, Structural Unification, Semantic Clarification and Introspection in SystemC

Luc Charest[1] El Mostapha Aboulhamid[1] Guy Bois[2]

chareslu@iro.umontreal.ca aboulham@iro.umontreal.ca

bois@vlsi.polymtl.ca

[1] DIRO, Université de Montréal
2920 Ch. de la Tour
CP6128 Centre-Ville
Montréal, Qc, Canada H3C 3J7

[2] Ecole Polytechnique de Montréal
2500, Ch. de Polytechnique
CP6079 Centre-Ville
Montréal, Qc, Canada H3C 3A7

Abstract

Reflective environments such as Java and .NET have provided programmers with the ability to gain access to a program structural information with ease. Reflectivity allows program metadata to be accessible at runtime.

C++ is perceived by many to be a well-balanced language; it combines elegant software constructs and raw execution speed. Due to C++ success, many hardware engineers are moving away for traditional solution such as VHDL to new ones such as SystemC. Since SystemC is based on C++, it is lacking some of the advanced features and concepts available in modern languages. For this reason, we built a system-level modeling environment called ESys.Net that is based on .Net and C# and our research is now focusing on introspection and interoperability avenues of SystemC.

While avoiding the RTTI (Run Time Type Information) library and those TCP/IP solutions, we propose in this paper some better and faster introspection and interoperability solutions that can be integrated, with minor modifications, to SystemC core.

7.1 Introduction

Having built ESys.Net [1], we are now exploring interoperability solutions with SystemC [2] – a well established system-level modeling solution. SystemC relies on the constructs of macros and templates to implement a simulation engine. Models written in SystemC which respect a well defined set of restrictions with regards to the use of object-oriented features may now be synthesized into hardware [3]. A "SystemC Verification Library" has been recently proposed by the OSCI consortium [4]. SCV

however, the end user must still wrap his types with a side declaration using templates and macros.

We propose a new approach based on type unification. This solution unifies all the datatypes used in a model (all user types will have a common ancestor class) and will forbid the misuse of these data types by enforcing SystemC and the user model to "look" for the common inherited base class before considering that it is a datatype. Our solution is more elegant in term of design for it makes minimal use of macros and templates, and it is based on an approach supported by the software community since

1. it is based on a Common Rooted Object, which is how Java implemented its type system,

2. it is similar to .Net type system and,

3. it is actually a mere implementation of the Composite design pattern [5].

A Design Pattern is a sketch for an Object-Oriented design solution. It shows, at a high abstraction level, how to resolve common OO problems. They were discussed in one of our previous paper [6] and Astrom [7] has used the Composite pattern for HDL models, but it was for bus implementation, our utilization of the pattern is much broader and general.

Another approach to introspection can be found in [8]. Doucet et al. are working on Balboa [9], an environment that brings introspection with the insertion of a higher level abstraction language along with its compilation phase. They have listed different strategies for realizing introspection but the solution we propose, which is based on inheritance, fits none of their list. The closest match could be a mix of what they define as "Sub-typing" and "Composition Replication". Although we do not intend to

that facilitate the design of models at transaction level by making signal and port declarations simpler and by making `sc_module` more generic than before. For a multiprocessor application, where the localization and the structure of the data are of utmost importance, introspection may be vital. The method we propose not only permits us to link more easily to an external program (such as a GUI tool) and to have the ability to inspect the structure of a given data without knowing statically the layout of its structure, but it also draws a clear line between structural elements and data elements of a SystemC user model.

7.1.1 Introspection & interoperability

The terms "reflection" and "introspection" are the ability that a language (a program) has to query its nature and structure during execution time (dynamically). Interoperability can be seen as a metric quantifying the ease of concurrent execution, communication and cooperation of at least two distinct processes.

We can classify information we would like to recover from the model and the simulator into different categories:

Structural information :

- Modules:
  - Name
  - Position in model topology
  - Position in model hierarchy
  - Incoming and outgoing communications (ports, interfaces and chan-

  - State

Data and type information :

- Signals
- Ports
- Channels (e.g. FIFOs)
- Custom structs
- Events

Miscellaneous :

- Methods (process):
  - Name
  - Parameters
  - Content (procedural description of the behavior)
- Higher semantic:
  - Hardware: Is it a bus? A processor? A register?
  - Software: Does it have an interface? Is it recursive? Is it an RTOS (Real-Time OS)?

For the interoperability part, there might be the need for some functionalities, for example:

Simulation control :

- Pause
- Reset (Restart)

Debugging control :

- Trace X
- Start trace
- Stop trace

Introspection is needed mostly when cooperation is needed through interoperability, hence their relation. As we will see later, a lot of solutions include parsing the source code and generating a wrapper which will then mimic operations, bringing interoperability. In this paper, we seek for the moment a low level and fast interoperability. A lot of applications can justifiably qualify for interoperability and introspection. For example, it can be a GUI needing to query the structure of the model (module hierarchy and signals connections) and its current simulation state (data values of signals, current process executing, simulation time, delta count, ...). A co-simulations engine is another application of introspection and interoperability, where a model could want to instantiate, then "look" and "control" another model. The example might be far fetched, but imagine a generic multiprocessor model that could adapt itself automatically with the processor models by querying dynamically their bus and protocol requirements. It could then select the appropriate interfaces, instantiate them, connect the wires and then control the processors.

Of course to realize such a concept, we need then to bring the introspection to a higher layer of information and to tag semantically our models. Although we do not

Even if hardware a synthesis tools programmer could be interested in getting semantics and structural information out of the models, we think hardware synthesis via introspection is more or less an inappropriate choice. The difficulty lies in the fact that C++ has no built-in introspection mechanisms (except RTTI for basic datatypes) and this hardens the task of getting the behavioral information out of the methods. Naturally, we could compile for, and use some debugging information to get the link with the source code line number, but then there is parsing to be realized. Even if we get the machine codes of the host machine of the simulation, these opcodes will have to be translated and analyzed by the synthesis tool, loosing at the same time the portability (because SystemC can execute at least on Intel and Sun, opcode will differ). A better and simpler solution for synthesis is through parsing, but even then, there is a problem and introspection can "team" with code parsing to resolve this problem.

The dynamic nature of C++ (hence SystemC) is of a big advantage over VHDL. Just imagine the nightmare of creating a flexible recursive network interconnection scheme with VHDL. With SystemC it is quite easy because of its dynamic nature, the realization of the model is deferred up to the execution time. There is then no guaranty beforehand for the tool that a module is constructed or not, even if the module is present in the source code. Parsing alone, is not enough; code analysis (or execution) is unfortunately required. The lines of code below should prove our point:

```
1  if (rand() > (RAND\_MAX / 2))
2    module = new module_a();
3  else
4    module = new module_b();
```

Until the time of execution, there is absolutely no way of determining if `module_a`

as random values can be justified in the case we are doing architectural exploration. Imagine constructing the model while relying on the calculations of some statistics analysis or creating modules from an hypothetical genetic code generated in a loop of a design exploration using genetic AI algorithms.

One should not mix this dynamic nature of SystemC with conditional compilation, where we usually use `#define` and `#ifdef` C keywords. This code inclusion/exclusion mechanism is resolve statically by the cpp (C Preprocessor) and not during the course of the execution of the compiled program.

Unless we enforce a subset of SystemC and strict rules of module instantiations (something as forbidding module declarations inside conditions and loops), another (yet maybe much better approach) would be not enforcing any instantiation rules (reconciling at the same time the simulation and the synthesis semantics) by using introspection. Since the model has to be instantiated in whole and its signals linked correctly just before calling the `sc_start()`, we could take advantage of that precise moment to help synthesis tools by taking a "snapshot" of the model using introspection on the model structure. A small algorithm could query the list of instantiated modules, signals, connections (...) and then dump the information on disk, in a structured (text, XML, ...) file that would be recovered by the synthesis tool that would then be able to do static analysis of the modules using the instantiation information.

There are a lot of different needs for introspection and a lot of ways to realize it (see appendix 7.A) and we did not intend to list all the variants. Considering that each time we add a layer it makes the things slower, the needs of the end users are an important thing to consider, where someone doing SystemC-with-SystemC communications inside the same execution context (intraprocess interoperability) does

- Used within the same simulation context *versus* interprocess (over some interprocess communication channel)
- Used with the same type of host (same platforms) *versus* distinct hosts (cross-platforms)
- Used with the same language (homogeneous language) *versus* distinct languages (heterogeneous language)
- Implemented by the language itself (usually better for standard) *versus* by a library on top of the language
- In a one-way mode (can be queried but can not query) *versus* two-way mode (complete communication/synchronization layer)
- Made statically (through parsing, usually faster execution because of the compiler optimizations) *versus* dynamically (usually more flexible, more interactive)

The approach we present in this article targets mainly dynamically intraprocess introspection and interoperability, while being aware we are helping by doing so, a potential interprocess introspection and interoperability. We prefer to see the SystemC library itself modified for this purpose (Figure 7.26 but replacing SystemC by an enhanced version) rather than implemented in a library over it (as in Figure 7.27). It is better for the standards, where any SystemC model would be more interoperable (not just the ones respecting the introspection/interoperability library). It is not intended to be used across platform and across language, but it could be the base of some mechanism having these objectives.

7.1.2 Current introspection approaches

Verilog PLI (Programming Interface Language)

The Verilog world has a very strong interface language which is used to realize interoperability along with introspection. The two-way interface language supports many applications [10] and is exactly the kind of complete interface we are looking for on the SystemC side:

- Foreign language interface (C language in this case), can execute C model in part or in whole concurrently with Verilog
- Can access to C libraries.
- Read test vectors in C
- Delay calculation
- Custom output displays
- Co-Simulation
- Design debug utilities
- Simulation Analysis

Of course, since C is the basis of C++, we could use C++ code also and we can construct a custom interface "translator" to every other language that has an interface to C. The PLI standard does define a lot of functions which the designer can use but:

- "The PLI standard does not provide any guidelines on how PLI applications

Even if the "plug-in" is linked dynamically, there is no interprocess "communication channel" to use, we can both, pass values and call functions only directly.

Of course, the PLI interface is excellent for Verilog but it is dedicated and tightly coupled to Verilog; it cannot be implemented directly for SystemC. It has to be re-engineered for SystemC and that, in fact, goes in the direction of the proposed solutions of this paper. Currently, with some work, we could interconnect SystemC to Verilog using the present PLI interface, but nothing more.

Type "tagging" and RTTI (Real Time Type Information)

This is a mechanism which "tags" the types and permits to query dynamically (during run time) the type identification of any C++ object. By including the "typeinfo" header, the compiler uses the `type_info` utility class (Figure 7.1) which contains information about the type.

```
1   class type_info {
2   public:
3     virtual ~type_info();
4     int operator==(const type_info& rhs) const;
5     int operator!=(const type_info& rhs) const;
6      int before(const type_info& rhs) const;
7      const char* name() const;
8      const char* raw_name() const;
9    private:
10     ...
11   };
```

Figure 7.1: *type_info interface definition code*

The constructor is private forbidding the end user to instantiate the class for an inappropriate use. The class main utility is to check the type via a series of if-then-else clauses using the overloaded == operator (line 4 of Figure 7.1) on the return of the

```
1   func(void *p)
2   {
3     if (typeid(p) == typeid(someclass))
4       someclass_func();
5     else
6       if(typeid(p) == typeid(someotherclass))
7         someotherclass_func();
8       else
9          (...)
10  }
```

Figure 7.2: *Typical example of the undesirable "if-then-else" scheme*

flexibility; when a new class is created, this `func()` will likely have to be modified. Tweaking this "tagging" scheme can be done but the result is more or less the same, the need of the tag comparing function will always be present, no matter the form, with its same lack of flexibility and evolution. For example, an STL (Standard Template Library) [11] `map<>` dictionary can be used, where the `name()` is the key and a function pointer is the return value. The algorithmic complexity resides in fetching an element from the map but also in the string comparison algorithm.

RTTI does not bring a solution for the rest of our goals; although it can determine if an instance X is a module, it will not help to establish it place in the model hierarchy or the link with other signals. This has to be realized with methods properly implemented within the SystemC's kernel.

Also, RTTI does not resolve the `struct` declaration problem. Refer to Figure 7.26, if the first model is declared as:

```
1   struct
2   {
3     char version;
4     int count;
5     (...)
6   } foo;
```

the fact that the structure contains a version and a count field and that they differ in length.

Another useful part of the RTTI mechanism is the casting operators (`static_cast<>`, `dynamic_cast<>`, ...). The distinction between the two casts is that the static one is done during the compilation time, without any check while the other (dynamic) is done during execution time and should it fail, it will return a `NULL` value indicating the cast request was not legit.

There is the possibility to implement another version of the if-then-else with the dynamic casting operator; you try each type until you hit the right one. As stated before, this is usually not a good solution because it costs a lot to guess each time while other solutions usually exists (such as polymorphism, see section 7.1.2).

Type tagging via custom implementation and Common Rooted Objects

Most of the time, a custom implementation of the type "tagging" mechanism is done by assigning a unique integer value to each different type of the system and have a virtual function as "`get_ID()`" to return that number. With the help of object-oriented methodology, it is realized fairly easily with a common root except when the end user omits to implement the required functions for its custom type. It is still not that flexible and still does not bring a direct solution to the "`struct`" introspection problem but does have a few advantages over RTTI:

- The unique ID is "controllable" and can set a standard.
- It is easier to manipulate than a string (in terms of storage and speed)
- It brings the possibility of using the switch-case control flow which is quite faster

which is immediately initialized with the static string labeling the class (line 103 of Figure 7.4). As a reminder, a static variable exists as a global variable (only one copy for all the instance of the class). Any `sc_object` calling the `kind()` method (line 106 of Figure 7.4) will be served with the same returned value.

```
72   static const char* kind_string;
73   virtual const char* kind() const;
```

Figure 7.3: *Simple introspection mechanism defined in `sc_object.h` inside SystemC's Kernel*

```
103  const char* sc_object::kind_string = ''sc_object'';
104
105  const char*
106  sc_object::kind() const
107  {
108      return kind_string;
109  }
```

Figure 7.4: *Implementation of the simple introspection mechanism*

In the case of SystemC, it is not clear whether any object deriving from `sc_object` should or not implements this function. Obviously and logically, `sc_module` which inherits from it, implement this function by defining its proper `static kind_string` which is global to all `sc_module` instances. If an end-user re-implements this function in his hypothetical `my_sc_module`, it might break the simulator looking for "`sc_module`" and getting "`my_sc_module`". Maybe two distinct functions "`sc_kind()`" and "`kind()`" are needed. Currently, we can get the list of the modules of the system in another manner (see section 7.2.4), but as we will see later, this list is not organized and not suitable for fast operations. Note there is always the possibility to test with the `dynamic_cast<>()` but then we enter in the "try-fail-until_success" loop instead of a direct identification

Virtual functions and polymorphism

This is not a introspection mechanism but something that can avoid it and is usually a better road than the "if-then-else" one (Figure 7.2). The end user often thinks he needs introspection while he actually do not need it at all. It seems as a natural solution for a procedural programmer, when he has two different objects of distinct datatype, to test the type, then call the appropriate function according to the datatype.

When choosing this "polymorphic" way, both functions `some_func()` and `some_other_func()` would be renamed into a `func()` method and put into different classes. These new methods would be specialized in, for example, `some_class` and `some_other_class` which would both be derived from a new `some_base_class`. The base class would supply a `virtual func()` definition which would allow the polymorphic mechanism to know there might be some specialization in the derived class. Upon the `func()` call, the polymorphic mechanism dynamically finds the appropriate version of the method according the underlying datatype, using a virtual table the compiler constructed.

Unfortunately, this solution is not always viable because it is not possible to foresee all the needs that an end user could have. The base classes of SystemC are defined by OSCI and any significant change to the code will make it non-standard. We need introspection because we must inspect the datatype from an external tool/model which often will have be compiled separately, having its own "knowledge scope" which does not include the datatype definition.

The Visitor Pattern

As stated earlier, the custom tagging suffers from poor extensibility and virtual func-

objects of the hierarchy implements a specialized version of the `accept(Visitor v)` method. When the visitor calls the `accept` method it is signifying to the object that it would like to do a certain operation on it. The object responds with a call back, identifying itself to the visitor by the same occasion. The visitor has to implement different actions according to the kind of the visited object.

The biggest reproach made to the Visitor Pattern is that it breaks a little bit encapsulation by deferring some responsibilities to another class.

SCV (SystemC Verification Library)

SCV is made mostly for verification through the randomization service they provide, and offer some kind of introspection. SCV enable to build a wrapper over the custom datatype by creating (on the side) some other structured information about that datatype. The end user that wants to gain knowledge about that datatype is required to construct that wrapper. In a C++ coding style spirit, they both use heavily macro and template declaration. Figure 7.5 show a structure to be wrapped by SCV. Figure 7.6 show the extension which the creator of the model must declare in order to render its structure introspectable. As we mention earlier and shown in Figure 7.27 (where a library is built over SystemC), this kind of approach makes the introspection part of the library more or less useless because it is not nested into the SystemC's

```
1  struct packet_t {
2    sc_uint<8> addr;
3    sc_uint<12> data;
4  };
```

Figure 7.5: *Original structure wrapped by the SCV mechanism (example taken from SCV 1.0e specification)*

```
1  SCV_EXTENSIONS(packet_t) {
2    public:
3      scv_extensions< sc_uint<8> > addr;
4      scv_extensions< sc_uint<12> > data;
5      SCV_EXTENSIONS_CTOR(packet_t) {
6        SCV_FIELD(addr);
7        SCV_FIELD(data);
8      }
9  };
```

Figure 7.6: *Wrapping extension the end user must declare for the* `struct` *of Figure 7.5 (example taken from CV 1.0e specification)*

knowledge about the original structure. Looking at their code example (Figure 7.7) the `tool_B()` function (lines 20-22) sure may be compiled alone but needs to be called from another function (lines 8-14) which needs to instantiate the template SCV extension, and this one *needs* to be compiled with the knowledge of the structure. This is because the compiler, upon precise moment of the static evaluation of the template parameter `T`, needs to know exactly what types to generate. Then, once created, the extension information must be passed to the introspecting tool/model.

So if static extensions are to be used for introspection independently compiled, there is a serious call-back mechanism to implement and this would not be standard if it is not implemented by OSCI.

The SCV solution of getting the interface of wrappers is interesting, but there is basically no standard common access point where the end user could get what they call the SCV extension for the custom structure.

SCV defines two kinds of extensions, static (Figure 7.7) and dynamic. The SCV documentation lacks a concrete dynamic extension example and could be augmented with one. An example (Figure 7.8) packaged with the SCV brings some hints without proving it can solve the introspection problem.

```
1  // The function my_code needs the C++ header for packet_t to instantiate p.
2  void my_code() { packet_t p; tool_A(p); }
3
4  // The utility tool_A can be designed for a generic type instead of
5  // the specific type packet_t
6  // This template needs the header for packet_t to compile.
7
8  template <typename T> void tool_A(const T& p) {
9    scv_extensions<T> ext = scv_get_extensions(p);
10   cout << "Argument p has " << ext.get_num_fields() << " fields." << endl;
11
12   if (ext.is_integer()) ext.assign( rand() ); // basic assignment of values
13   tool_B(ext);
14 };
15
16 // The implementation of a proprietary function can be hidden with the
17 // use of scv_extensions_if
18 // This function does not need the header for packet_t to compile.
19
20 void tool_B(scv_extensions_if& p) {
21   if (p.is_integer()) cout << p.get_integer() << endl;
22 };
```

Figure 7.7: *SCV static extension usage, (code example from the SCV 1.0e specification)*

point to an extension instance. With the extension, introspection can be done. From here, is it rather unclear what SCV expect the designer to do. Whether he modifies his code to make such thing as `signal<>` and `port<>` of type `scv_extensions_if *` so that any external tool/model inspecting the internal structure of SystemC could get a hold on the signal type, whether the smart pointer is to be used as a methodology for wrapping legacy code "as is".

So for the moment, we could say SVC introspection is to be used when the other tool/model is compiled with the SystemC model in a more or less tightly coupled way.

Java and .Net

```
268   // ----------------------------------------
269   // dynamic extension (allow attachment of callbacks,
270   // uninitialized values, constraints, etc.)
271   //
272   // - require the use of scv_smart_ptr template.
273   // ----------------------------------------
274
275   // randomization (direct) (VWG R3d, Rb1)
276   scv_smart_ptr<packet_t> pp;
277   scv_shared_ptr<scv_random> g(new scv_random("gen", RND_SEED));
278   pp->set_random(g);
  (...)
288   pp->packet_type = fieldValues[0];
289   pp->src = fieldValues[1];
290   pp->dest = fieldValues[2];
291   pp->payload = fieldValues[3];
292   scv_extensions_if *ppExt = pp.get_extensions_ptr();
293   size = ppExt->get_num_fields();
```

Figure 7.8: *Source code of* ***examples/extensions/introspection1/test.cpp*** *example packaged with SCV*

them, they do not apply to SystemC usage. Their only place in the SystemC world is in the elaboration of a communication interface to realize the interoperability between SystemC and [Java/.Net].

We used in the elaboration of our ESys [1] tool the attributes of .Net which are tightly related with the introspection mechanism. The reflective mechanism of .Net is so deep and rich that one can recover the information about the parameters of a class method.

These mechanisms are introspecting classes and methods, they are not related to the upper level of semantics. This is where attributes can take the relay and augment the model with semantics information. The reflectivity mechanism does not allow tracking instance information. For example, one can get the structure of a class, but it can not get a list of instances of this class in the system. This kind of information

7.1.3 Current interoperability approaches

Here we list some interoperability tools we could use to enhance SystemC interoperability aspect.

SWIG (Simplified Wrapper and Interface Generator)

This program is a candidate tool to ease interoperability of SystemC. It is used to wrap mainly C [14] (C++ supported but not recommended) into a lot of languages (Guile, Java, Mzscheme, OCAML, Perl, PHP, Python, Ruby, Tcl, Chicken, C#, ...). We have looked closely at the tool when we considered at ways of connecting our ESys.Net [1] (written in C#) with SystemC. Although SWIG seems to be strong and stable, we ran into a bug where SWIG attempted to access a private assignment operator which then makes the interface uncompilable. If we restrict the assignment operator for a read only port, that is totally acceptable in a situation where an end user could inadvertently write to the port but it is not the case when a tool must make a copy of that port. Although we could have modified SystemC to include some kind of friend declarations, we submitted a bug report to SWIG.

CORBA, DCOM and StepNP

CORBA (Common Object Request Brokerage Architecture) approach [15] which comes from the OMG (Object Management Group) defining a way of interfacing interprocess communication using the TCP/IP stack. By writing IDL (Interface Definition Language) code, one can describe the entry points of its piece of code written in any of the selected language, rendering the component "CORBA enabled". An automatic converter exists that parses the source and generates the IDL. Once described, the

SystemC with SystemC co-simulation. Therefore, we are looking at lower level interoperability solutions. A possibility is DCOM (Distributed Component Object Model) is the pendant of CORBA on the Microsoft side. It mostly targets Microsoft platforms. Another option is StepNP which is a research platform developed by research engineers from STMicroelectronics. Derived from the standard IDL, StepNP propose the SIDL (SystemC IDL) [8] which wraps SystemC code in an IDL way.

7.1.4 Our approach

Objectives

- Unify the data types and development methodology.
- Bring better Object Oriented design.
- Ease the development, code reuse, portability, introspection and interoperability of user models.

The remainder of this paper is organized as follows: in section 7.2 we discuss the current problems of SystemsC we would like to solve, section 7.3 is a first part that will introduce a solution using the Composite pattern to ease custom datatype introspection while enhancing the design of the end user model. Section 7.4 will present a second part that introduces a solution which is independent but derived from the first part with the distinction that it is applied to the structure instead of the datatype. Section 7.5 will shed light on how simple modifications to SystemC can help the end user to have a better control and understanding on the SystemC's core. Section 7.6 will wrap the subject by bringing a better idea of the overall solution and section 7.7

7.2 Introduction to SystemC and its problems

SystemC is a C++ library built over C++. It is designed for general purpose hardware focused parallel simulation. The way the library is referenced, the way the models are constructed, the usage of defined macros (`SC_MODULE`, `SC_METHOD`, ...) may seem to make SystemC a new language of its own. The fact that it was meant to be built on top of C++ and directly compilable by existing C++ compilers is a major advantage, but it makes SystemC nothing more than a C++ library.

The modules are there to reflect the `entity-architecture` declaration of legacy VHDL, but processes are of most importance when speaking of hardware simulation. They are the ones, along with the signals construct, that are scheduled for execution in a very specific manner to emulate the hardware parallel behavior.

7.2.1 Multiple simulations

If one looks at the way SystemC is currently implemented and used, he will find that some constructions are not really following the Object Oriented philosophy. A good example is by looking at how references to `sc_simcontext` methods are defined. Inside the alpha version of the LRM (Library Reference Manual) [16] we can find the following explanation about `sc_simcontext`:

"`sc_simcontext` is a class that is used by the simulation kernel to keep track of the current state of simulation. It can provide information to modelers such as the current delta-cycle count, and provides access to any structural element in the design."

And if we look in the actual source code of `sc_simcontext`, we can see that multiple simulations was envisioned:

reasons: it is implemented as a global function and not as a class method, we could pass a pointer to a `sc_simcontext` as a parameter but then why not use the simpler object oriented construction such as `my_simcontext.start()`?

```
875  void
876  sc_start( const sc_time& duration )
877  {
878      sc_get_curr_simcontext()->simulate( duration );
879  }
```

Figure 7.9: *sc_start() function definition*

More precisely, we can look at Figure 7.10. Line 831 of `sc_simcontext.cpp` contains a pointer to the sole instance of `sc_simcontext` that should exist in the system. This instance is created by the first call to the global function `sc_get_curr_simcontext()` (line 842-843). The line 830 indicates with reason that this is not multi-thread safe.

```
830  // Not MT-safe!
831  static sc_simcontext* sc_curr_simcontext = 0;
832
833
834  sc_simcontext*
835  sc_get_curr_simcontext()
836  {
837      if( sc_curr_simcontext == 0 ) {
  (...)
842          static sc_simcontext* sc_default_global_context = new sc_simcontext;
843          sc_curr_simcontext = sc_default_global_context;
  (...)
845      }
846      return sc_curr_simcontext;
847  }
```

Figure 7.10: *Creation of the global sim_simcontext*

Why have they done it this way? Maybe SystemC kernel is not ready yet for

the usefulness of the simulation context to the user and prepare him to migrate to a multiple simulation environment. Currently, the link between a user module and the simulation context is made implicitly and automatically upon the creation of the module. Moreover, they should have used the singleton pattern instead of the whole code of Figure 7.10 which does not prevent the end user from modifying directly the simulation context anytime.

7.2.2 Semantic separation between the simulator, the context of the simulation and the user model

As one might begin to realize with section 7.2.1, the model, the simulation context and the simulator itself forms a whole. For example, if there is a need to merge two distinct models into a single simulation, a lot of work may be required if the end users had not foresaw this situation by encapsulating their models into a single "easy to instantiate" module. The same thing apply to multiple distinct simulators; if we would like to have distinct "models of computation" as the researchers from Berkeley [17] define them, it would be a very tedious task to try to instantiate multiple distinct simulators because the simulation context is implicit with SystemC. Even if the task of synchronizing "heterogeneous" simulators inside a single simulation, it would be feasible to spawn two "homogeneous" (event driven in our SystemC case) simulators for a emulation of a distributed environment for example. Even though this is still non sense, we think a clear semantic separation between these concepts is primordial and we will show what we mean in section 7.5.

7.2.3 Semantic separation between data information and struc-

temC structure.

Hierarchical channels : is described as a channel that is more complex and that can support ports, signals and other nested modules. But in fact, looking at line 558 of `sc_module.h`, we can find this construct:

```
558   typedef sc_module sc_channel;
```

This means that a hierarchical channel is not "as" a module, it is a module. When the user declares a `sc_channel`, he will really have a hard time finding its channel inside the module list of object as we will see in the section 7.2.4.

7.2.4 Recovering structural information

After having created and bound the modules and signals together, one can, following the `sc_simcontext::sc_first_object()` description, get all the information about the model hierarchy. We use the code of Figure 7.11 to get the list of SystemC objects that were instantiated and linked with the simulator.

```
(...)
1   sc_object *obj = sc_get_curr_simcontext()->first_object();
2   int count = 0;
3
4   while (obj != NULL)
5   {
6      cout << count++ << ": (" << obj->kind() << ") " << obj->name() << endl;
7      obj = sc_get_curr_simcontext()->next_object();
8   }
(...)
```

Figure 7.11: *Code to get information about the objects of the simulation*

Now what would be interesting is if we could get information about how the user

```
1   0: (sc_module) Emitter
2   1: (sc_method_process) Emitter.emit
3   2: (sc_in) Emitter.port_0
4   3: (sc_out) Emitter.port_1
5   4: (sc_module) Receptor
6   5: (sc_in) Receptor.port_0
7   6: (sc_in) Receptor.port_1
8   7: (sc_method_process) Receptor.receive
9   8: (sc_clock) clock_0
10  9: (sc_method_process) clock_0.negedge_action
11  10: (sc_method_process) clock_0.posedge_action
12  11: (sc_signal) signal_0
```

Figure 7.12: *Result of the execution of the code of Figure 7.11*

for interprocess, cross-platform or cross-language introspection but for intraprocess (and homogeneous language), it is quite an overhead, where before doing anything serious with this kind of information, a 3^{rd} party program would likely transform this information into a nice structured tree. If we are dealing with the same language, a structure as will be shown in section 7.4 could be used directly to represent the model element hierarchy.

With current SystemC basic introspection, problems arise when trying to get connection information between ports (module) and signals. When the user creates a signal, the constructor of `sc_prim_channel` insert the signal into an undocumented `sc_prim_channel_registry` instance in the `sc_sim_context` class. This is a place where one could get information about the signal interface but there is no facility (as a method publicly available) to fetch this information. When a new signal is bound to a port, it is bound using its `sc_interface` nature (via implicit conversion to pointer of this base class). This way, the port only stores the information about the `sc_interface` of the `sc_signal<>` which then makes it very difficult to get back the information.

7.2.5 Recovering datatype information

Even if one can get a `sc_object` pointer to a signal, we can not upcast to a `sc_signal<T>` unless we know a priori the type `T`, so an introspecting program cannot get the value of the signal unless he was built for that type, unless he "knows" that signal X is of type Y.

As described in section 7.1.2, there could be a way to tag each type with a different ID code, but then all the ID codes must be known in advance by the 3^{rd} party tool. If this is part of the standard definition, it can be more than acceptable this way. A `virtual int sc_prim_channel::get_type_ID()` can be programmed to return such code that would have to be reimplemented and distinct in each signal type instance.

Problems arise with user defined `struct` construct. If the end user defines (as example) a `sc_signal< struct tcp_header >`, the "knowledge scope" (as defined in section 7.A) of the `sc_signal` part is not so limited (SystemC and all of its models) but the variable part (`struct tcp_header`) has a "knowledge scope" limited to the model defining the structure. Naturally, the `struct` keyword comes from C++ but the language does not offer intrinsic reflectivity on this construction. Even though another model could get a hold on a generic pointer (`void *`) it has no knowledge on how this structure is built, hence, it cannot use that structure. There is basically no way (unless the end user code gets directly parsed by some other tools) to get information about that tcp_header since no ID has been associated with that type. Even if we let the user create a new type identifier, the introspecting tool then has to be modified to "gain knowledge" of this structure along with its type ID.

7.2.6 Introspection problem of interfaces (channels)

interface any time, making a structural check impossible.

7.3 First part: datatype introspection solution – the Composite pattern for type unification

Using the type ID solution described in section 7.2.5 combined with the use of the Composite pattern [5] which permits to express hierarchical structural imbrications of elements, we will now be able to introspect SystemC user defined types. We first create an abstract base class (`sc_datatype` in our case) upon which we derive all the possible implementations of SystemC types.

Among these implementations, we supply a class which acts as a container of a set of multiple instantiations of the type of the base class (consequently, its inheritors). By polymorphism, the container will be able to hold only objects from the derived classes and/or other instances of the "container" class itself or inheritors.

7.3.1 Description of a new set of classes

To implement the Composite pattern for a SystemC model requires some new classes (Figure 7.13). We used the SystemC's prefix "`sc_`" on classes we consider fitting more into the SystemC library than on the end user side:

`sc_datatype`: this class is implemented to be the abstract base class of any data that should exist within the world of SystemC. This class brings the assurance that if a data exists in the system, it behaves as a system data. If a given property or method were to be considered as being vital for all the data of the system, it

derived classes that have an impact on the base method, so each inheritor has the responsibility of its proper duplication.

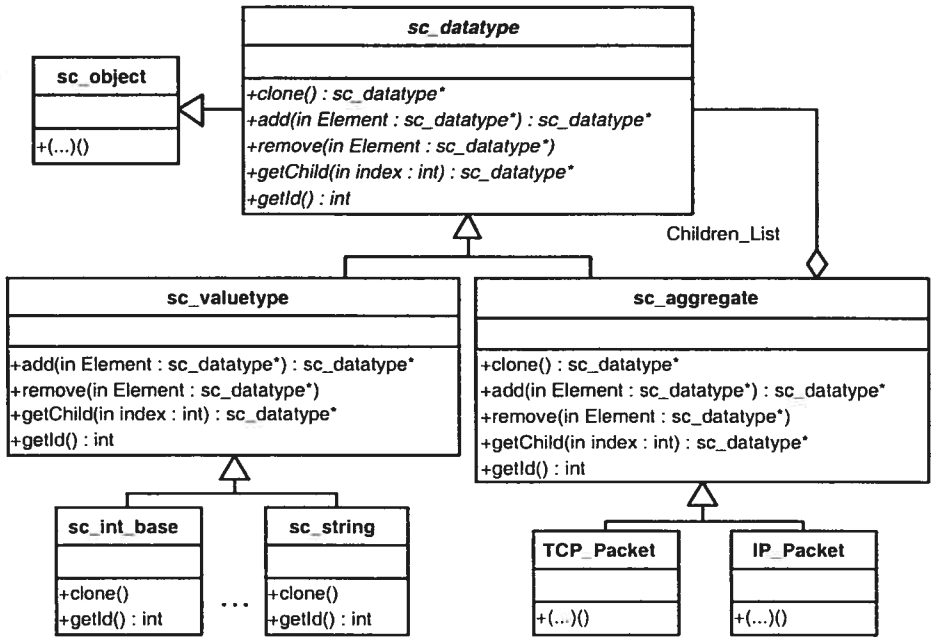


Figure 7.13: *Composite pattern applied to SystemC (partial class diagram)*

`sc_valuetype`: this class is an abstract class that permits to establish a common root of concrete types. As for `sc_datatype`, any useful feature of concrete types should be implemented in here if it has no relation with the next class: sc_aggregate.

`sc_aggregate`: the name of this class has been carefully chosen; we did not want something that would have brought confusion with the `struct` C++ keyword. This name emphases the fact that we put some atomic units to form a whole.

respected, the introspection is achieved by looking at the given `sc_datatype` pointer. Because the base class is abstracted, it cannot be directly a `sc_datatype` but one of its derivatives. We first call the `is_aggregation()` predicate (not shown in diagram). A false value indicates that we automatically have to deal with a derivation of a sc_valuetype.

In the case it is a `sc_aggregate` or derived, we must first assure Children_List is not empty. If not, we have to deal with sub types and we would have to repeat the process recursively, or at least repeatedly to gather all the information about the "children".

A method which is implemented in the base class type does not have to be necessarily reimplemented in the derived classes. Indeed, the `sc_aggregate::clone()` method must call the `clone()` of all the subtypes inside its `Children_List` (Figure 7.14), but if the base class implementation is sufficient, `clone()` does not necessarily need to be reimplemented by a class which inherits from `sc_aggregate`. This way, our types implement the Prototype Pattern which permits to create a first instance of an object and then replicate it at will (Figure 7.14), based on the first one (a prototype). This way, `sc_aggregate` can clone itself recursively (while preserving the values of all the fields) if and only if all its "children" field implement `clone()`.

The current implementation for the internal container (`Children_List`) is the SystemC's custom vector implementation which mimics the one supplied by STL [11]. It could be even a better idea if we use a dictionary, or in STL terminology, a map that would link the subtype name with the field itself.

```
1  sc_datatype *sc_aggregate::clone(void)
2  {
3    sc_aggregate *ret = new sc_aggregate();
4    //for each field in Children_List
```

7.3.2 Modifying SystemC

The base classes of the Composite pattern were implemented as described in the previous section, but now we need the atomic elements to realize the link with the conventional C++ and SystemC datatypes. To show that our solution can be implemented without a lot of modifications to SystemC we chose some SystemC datatypes (`sc_int_base`, `sc_int<>`, `sc_string`) and we modified them. All we had to do was typically to derive these classes from the new `sc_valuetype` and fix the constructors accordingly. There were some minor problems such as mutual inclusion conflicts and forward references to add but they where quickly overcome.

7.3.3 Static types versus dynamic types

We can consider that the Composite pattern solution brings a new kind of "dynamic types" where we put each part of the "type" inside a list; this renders the "type" more dynamic than before. Since everything in the list is considered as being a subcomponent of the whole "type", and since the list is modifiable after the executable has been launched, the "type" is hence modifiable after the compilation and after execution is launched. By opposition, even if you have such generic concept as a template, it must be known at compilation time because templates are being resolved and replaced during this stage of the compilation process (so if it was fixed as being `int`, once it is compiled, it will stay as an `int`).

Because the Composite pattern is based on common rooted objects principle, some interesting things can be done more easily; one could use our solution to start a simulation and then prompt the user for "which kind of data you want to pass on the bus?". The user would choose on the fly how the datatype is constructed without

structure.

The applications of this Prototype pattern is, for example, when a complex objects must be created with distinct default values (where one do not want to hard code these values in the constructor as default values). Multiple object can be instantiated with different values and be cloned to have those default values modified afterward. Another usage would be in a genetic algorithm, where the need to clone the gene sequence multiple times can be interesting, using the `clone()` method. Even the modification of our new "datatype" on the fly could make sens in a genetic algorithm application.

7.3.4 Linking to eliminate the type cast

If the programmer must always reference the `Children_List` vector to fetch the information, it can be quite tedious and time consuming, especially if the extent of the subtype is quite deep. It might be preferable to access the data more directly instead of having to query the `Children_List` vector for the appropriate information. We suggest pointers class properties, initialized inside the constructor:

```
1   sc_int<4> *version; // inside class declaration
2   // inside constructor...
3   version = static_cast<sc_int<4> *> (add(new sc_int<4>(init_val)));
```

Unless the type is dedicated to a "normal static usage" the problem with this first approach is that the link between the reference and the data itself can be broken. If we think the other way around, creating the object as usual then insert it into the list might be seen as better:

```
1   sc_int<4> version; // inside class declaration
```

is no way to delete this field and recover the used memory space unless we delete the whole object.

Avoiding the `static_cast<>` in the constructor if there is no need of creating the type "dynamically" (on the fly, after execution) is done fairly easy by moving the assignment inside the `add()` method call:

```
1  sc_int<4> *version; // inside class declaration
2  add(version = new sc_int<4> (init_val)); // inside constructor
```

This is yet another way to do it but even though it might be confusing at first sight it should be kept together since the call to the `add()` method could be more easily forgotten by the end user if they were separated.

7.3.5 TCP/IP packet: an illustrative datatype introspection example

We first started with our modification integrated into SystemC. Here are descriptions of classes that could have been created by a SystemC user using the new datatype methodology:

`TCP_packet`: Transfer Control Protocol Packet class, is derived from `sc_aggregate` because it constitute an aggregation of basic types (`sc_int<>`, `sc_string`, ...).

`IP_packet`: Internet Protocol Packet class (Figure 7.15 and Figure 7.16), is also derived from `sc_aggregate` for the same reasons as `TCP_packet`. The code for the class definition is not much complicated than a plain `struct` declaration except the fields are now replaced by pointer properties. It contains a user type;

```
1  class IP_packet : public sc_aggregate
2  {
3    public :
4
5    // "quick reference" pointers
6    sc_int<4> *version;
7    sc_int<4> *hdr_length;
8    sc_int<8> *service_type;
9    sc_datatype *data;
10
11   // constructor
12   (...)
13 };
```

Figure 7.15: *Partial IP_packet class definition*

we have created these specialized versions of the aggregation to show that it is more than justifiable for sc_aggregate to be extended.

```
1  IP_packet::IP_packet(int version, int hdr_length, int service_type,
2   sc_datatype *data) : sc_aggregate()
3  {
4    this->version = static_cast<sc_int<4> *>(add(new sc_int<4>(version)));
5    this->hdr_length = static_cast<sc_int<4> *>(add(new sc_int<4>(hdr_length)));
6    this->service_type = static_cast<sc_int<8> *>(add(new sc_int<8>(service_type)));
7    this->data = (add(data));
8    (...)
9    //a macro can be used here...
10   //  #define DATA_INSERT(NAME,TYPE,INIT) \
11   //  this->NAME = static_cast< TYPE * >(add(new TYPE(INIT)))
12 }
```

Figure 7.16: *IP_packet constructor*

As for example, such things as the checksum field can be calculated whether by the field class itself or by another class using the Visitor pattern [5]. Although we did not implement any, it is possible to extend the described datatypes system with new methods to help the Visitor pattern and calculated fields.

`Emitter/Receptor:` these are the two `sc_modules` that implement a communication application in our example. One of the nice construct that comes from the Composite pattern methodology can be seen in Figure 7.18 where the `Emitter` creates a packet (line 3 and 4 of Figure 7.17) and send it (line 6) through a generic dataline (line 8 of Figure 7.18).

```
1  void Emitter::emit()
2  {
3    TCP_packet *tcp_packet = new TCP_packet();
4    IP_packet *packet = new IP_packet(4, 5, 2, tcp_packet);
5    packet->set_location(this);
6    out.write(packet);
7  }
```

Figure 7.17: *Emitter's SC_METHOD function*

As discussed in section 7.3.7, common root object can be quite interesting for modules, especially if a module does not have to "understand" the data he is working with (e.g. a generic TDM (Time Division Multiplexer) that would take `sc_datatype` as an input line and output two different `sc_datatype`). Such modules are then reusable "as is", where the user does not have to change the I/O port definitions of the module but naturally, if the module has to perform some operation on the data, it would require some code to be re-engineered inside the method(s) of the module. On line 5 of Figure 7.17 we have inserted a command that is not related to the Composite pattern itself but on the common root type where some function can be provided to help the introspection without a heavy implementation cost. A property has been added to the object instance and provides information about the situation of the data. It can be access back with the `get_location()` method which returns a pointer to the owner module. Although it might have been implemented as a `sc_attribute`

7.3.6 Deriving from `sc_object`

In our example implementation we first decided to derive from our base class `sc_datatype` the "super" base class `sc_object`. This is a choice in the implementation, there are other ways. Our choice was based on the fact that we wanted to show the "full features" potential of our solution without entangling ourselves into the speed issue.

After having done some quick experimentation with the shown solution (deriving directly from `sc_object`), we found the speedup was not so good (see section 7.6). A better envisaged way was to remove some unwanted features from `sc_object` and move them to an intermediate abstract class, we named it `sc_structural_element`. It is important to avoid breaking objects that originally derived from `sc_object`, so we had to make sure the newly created classes (`sc_structural_element` and the lighter version of `sc_object`) were to integrate together the original properties and methods of the old `sc_object` class.

Our choice to split class functionality between a new class and the older one was motivated by many factors:

- it can be used to introduce a clear semantic separation between a "general" SystemC object and an object which is part of the structure of the model;
- the kind string and kind() method of `sc_object` are useful to realize introspection;
- the unique name assigned to each instantiated `sc_object` might be useful for debugging purpose (but is not necessary to achieve introspection);
- the registration of objects into SystemC's internal object vector is not necessary

We did the experiment and the performances were much better than with our first approach.

7.3.7 One type to rule them all

Now that we have a common root class for all our type, the `sc_signal<>` and ports (`sc_in<>`, `sc_out<>`, `sc_inout<>`) classes can be modified accordingly (line 8 of Figure 7.18). All the modules in the model now become more "generic" and this eases their creation where they now exchange the same kind of information: `sc_datatypes`. It may be desirable to change the definition of the signal and ports so every user would be forced to use the Composite methodology (the solution we present) and render their signal passing interface as "standard" but this approach trashes the legacy code that would then have to be wrapped.

```
1  #include <systemc.h>
2  #include "Emitter.h"
3  #include "Receptor.h"
4
5  int sc_main (int argc , char *argv[])
6  {
7    sc_clock clock;
8    sc_signal<sc_datatype *> data_line(NULL);
9
10   Emitter emitter("Emitter");
11
12   emitter.clk(clock);
13   emitter.out(data_line); // generic ! Could be emitting anything...
14                           //    ...inheriting from sc_datatype !
15
16   Receptor receptor("Receptor");
17
18   receptor.clk(clock);
19   receptor.in(data_line); // generic also ! Replace at will !
20
21   sc_start(clock, 10);
```

current implementation of SystemC the `sc_signal<T>` upon template unrolling, creates immediately at least two distinct instance of the `T` type (one to preserve the current value, on to preserve the next value, so that the event-driven paradigm can be applied). This is good for `int`s and `bool`s but for our `sc_datatype` (which is abstract and can not be instantiated directly), in our current implementation, it gets hard for SystemC to instantiate this class directly. The next section will discuss this matter in details.

7.3.8 Passing data on signals using a pointers

By using polymorphism, the use of pointers is appropriate since it permits to pass a generic interface through the user model while the compiler keeps track of the real nature of the objects. On the opposite, if we cast a reference of a class into another different reference (e.g.: a super class), we then "change" the nature of the child for the compiler. The compiler will blindly obey to the requested cast by considering the object as being of the other type (a super class instance in our example). This kind of casting is not permitted. On the other hand, when using a super class pointer, the behavior falls under the polymorphic rule of C++ where if a method is declared as being `virtual`, the most specialized available form of the method will be used.

Another point in favor of using pointers rather than templates is that it eliminates the need of having to implement the tracing method, the assignment operator, the comparison operator and the stream operator which are clearly highlighted as required for users' datatypes in the SystemC's LRM [16] and more detailed in the user guide [18].

One must be careful though that by using pointers over `sc_signals` brings some issues:

- the simulation semantics is a little bit different as for scalar types. As if you where dealing with packets, two distinct `int`s being sent over the signal while being the same value will not trigger an event, while on the other hand, even if two packets have the same content, they will be distinct and trigger an event.

7.3.9 Legacy Code

There is a way to convert to and from `struct` constructions to help the utilization of legacy code. The cost is to impose to the creator of the datatype to implement two different methods to realize the conversion. The first one we named it `convert_to_struct()` and the second one `convert_from_struct()`. These are supplied in `sc_datatype` and declared as pure virtual so they have to be implemented in derived classes.

Of course, these methods once implemented are tied to the implementation of the legacy code they wrap. Platform dependent implementations causes a lot of problems but that could be seen more as a C++ portability problem. But still, a user that would like to see its implementation of the datatype to be platform independent, would have to supply many different implementation of these two methods.

A pitfall to avoid is to create multiple classes (e.g. `foo_linux`, `foo_windows`, `foo_sun...`) inherited out of a single type because specialized version of each platform will then be necessary for each objects, crating too much leaf classes. A better approach is to use the Bridge or the Visitor pattern [5] to perform this kind of task. The Visitor pattern, although breaking a little bit the rules of data encapsulation, would permit to transfer the OS conversion responsibility to another class. The .Net package used this kind of technique where a Convert class has the responsibility of converting any

We have implemented a `sc_convert` class that can be extended using the inheritance mechanism.

7.3.10 Abstraction layer from the OS

As described in the previous section, although the class itself is platform dependent, our approach can be used to render the type platform independent. By wrapping the type, we create an abstraction layer between the OS and the model. When the end user instantiates a given structure, he is not forced to know how it is implemented and on which OS it has to deal with, just as an ADT (Abstract Data Type) where the implementation (linked list, array...) can be hidden behind a common "interface". But then often there are associated methods which must be wrapped along with the data information. Of course, the structural type class is a nice place where to put this but there is the need to evaluate first if the cost of wrapping everything is worthy especially if portability is not a necessity for the project. Naturally, if the goal to be achieved is to create a reusable library of portable datatypes, our solution can supply the needed class organization.

7.3.11 Benefits and drawbacks

Benefits

- It supports many other design patterns (see section 7.9).
- It renders data types single rooted; each data can be reinterpreted as a specific given common rooted class.

more "generic" dynamically (section 7.3.5).

- It speeds up the development of models by passing loosed typed signals.
- The new "type" object can be created/modified after the execution has started (section 7.3.3).
- It separates the datatype from other SystemC construction and restrict the usage to specific classes if properly used.
- Ease introspection, hence interoperability.
- It also brings another abstraction level in-between high level transactional and low level detailed signals.

Drawbacks

- It burdens the single instance with a lot of information that has to be stored and managed.
- Some static optimization that were possible before with static types could not be possible anymore (see section 7.3.3).
- The choice of atomic types into an aggregation is restricted to those defined in the system or the ones that can be created (via inheritance) by the user. If extended, these user-defined types can be recognized by other third party software as the proprosed "system" (`sc_datatype`, `sc_aggregate`, ...) types, but their augemented nature might recognized without prior knowledge of it (supplying the definition of the derived class to the 3^{rd} party program).

- The solution slows the development of datatypes; each time the end-user wants to insert a new datatype in his system, he must create a new class (instead of a new `struct`).
- It brings loose typing, where the type cast can not be checked at compilation time (but remember to use `dynamic_cast<>()` since it is cast safe).
- The "type" is now dynamic, the tradeoff is always the same; between static constructions and speed versus dynamic construction and slowdowns.
- For now, sensitivity is restricted to the whole aggregation.

7.3.12 Alternative solutions to limitations

Lightening the overall structure

If speed is really an issue, there some solutions:

- Library Linking: we can first build two different implementations into distinct libraries; one for debugging; the other for speed, we then choose the right one by linkage at compilation time.
- Conditional Compilation: although macros might be controversial and fading away from modern programming languages there is however a part in the world of macros that are and will be subsisting for a long time because of it usefulness; conditional compilation.

The technique of using `#define` and `#ifdef` would help to describe in the same

Splitting data from methods

One can bring an argument where we could, in an attempt to lighten the class, separate the data instance from the datatype class. In other words, for each distinct datatype, we would create another lighter class (`sc_int_data` for example) that would maintain only the data itself along with a pointer toward the singleton datatype instance (`sc_int`, if we continue with our example).

This is a bit useless unless the leaf datatype class is very heavy. One has to know that one of the object oriented objectives is to lighten the structure of the program by factoring the functionality (methods) from the data (property). This is realized seamlessly by the compiler which usually instantiate the properties of the class in the heap, without the methods which are kept apart. Commonly, an underlying pointer (in the same manner as for the "`this`" reserved C++ pointer works) is passed to the methods indicating on which data the method call was requested. Except for static methods, that is why a class method always requires an instance to be applied to: because the compiler subtly pass a pointer of this instance. In the case of static methods, there is no pointer to be passed to the method because the methods is then global to the class and instance independent.

In brief, attempting to separate data and methods to save memory is useless because the compiler would be doing the same task redundantly.

Scripting with sed, perl and cpp

If the creation of new type classes is really a concern and becomes a burden, sed (Stream EDitor) perl and cpp (C Preprocessor) can come in help. All there is to do, is to create two different "template" file (not to be confused with C++ template

7.4 Second part: structural solution – simple Composite for structural elements

Now we propose to look at the `sc_modules` `sc_signals` and how we can manage to collect, store and retrieve the information about how the user defined model is structured.

Once the user creates a module, he can instantiate by encapsulation a load of internal sub modules. This relation is unfortunately not well organized; the designer has the liberty of choosing any variable name for its nested module. Freedom of choosing the name of the variables is good but at the price of preventing a standard common access point.

As seen in the introduction, there is a way of getting the model hierarchy but this involves passing by some strings construction. When dealing with direct intraprocess execution, a more rapid appropriate methodology would be helpful.

7.4.1 Base class – `sc_structural_element`

The purpose of `sc_module` is well known; if we were to match SystemC constructs with those of VHDL, the `sc_module` would be the closest in line for the role of "entity"-"architecture" pair. We use inheritance via the `SC_MODULE()` macro, and those who like to see Object Oriented constructions, opt to use the "`class (...) : public sc_module`" declaration. A quick look into [19] reveals that the `sc_prim_channel` and `sc_module` have, not only a lot of `wait()` and `next_trigger()` methods in common, but also the same implementation.

The redundancy is an indication that the system has gone into a lot of incremental

of the intersecting classes. It is rather a long term investment where a better and clearer separation of semantics and responsibilities can be achieved, where if a potential extension needs these functionalities, a new inheriting sibling can be created from the intersected root class, augmenting this way the previously joint concepts.

So we propose the creation of a new class called `sc_structural_element` (Figure 7.19) which should be an abstract class containing the unification of everything that is a part of the structure of the user model. We consider important that such things as modules, signals, channels and ports be identified under a common root other than `sc_object`. In the same manner we have done with datatypes, we could use the idea of the Composite pattern applied to the structure. The `sc_module` would derive directly from `sc_structure_element` and be the container, the exact equivalent of `sc_aggregate` of section 7.3.

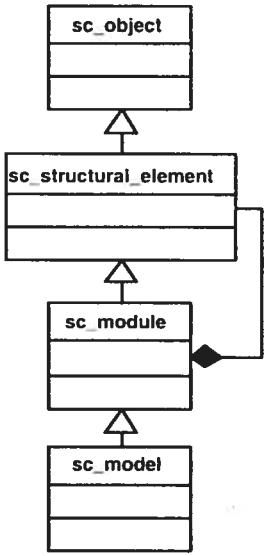


Figure 7.19: *Partial class diagram of the Composite pattern applied to SystemC's*

`sc_process` from `sc_structural_element` indicating processes can be nested inside modules. Some questions now must however be answered; such thing as `sc_process` and `sc_mutex` which are at this current time derived from `sc_object`, are those to be or not to be considered as structural elements? With the structure of Figure 7.19, `sc_prim_channel` (which is the base class of `sc_signal`) and `sc_port_base` (which is the base class of all ports) could be derived all directly from `sc_structural_element`, as they derive right now from `sc_object`.

Now that we have a new "model" class, we could instantiate as many models as we like inside the same executable and represent the distinct models with different instances of this class. This would enable us to tag all the previously dangling modules in the simulation now as being part of a specific model. An example of application would be a direct cross validation between a golden behavioral model and a RTL implementation model of the same design. SystemC can at the present time support many independent modules, so reuniting the modules under distinct `sc_model` would only bring clearer code and better flexibility.

7.4.2 Introspecting channels

As discussed in section 7.2.6, one of the major problems with channels for structural introspection is their lack of connectivity. To resolve this problem, we could enforce the registration of modules needing to access a given channel. Before starting the simulation, a module could issue a `register(this)` request of the given channel which could record the passed pointer of the module inside an internal list. Each time a call to a module method is requested, the callee's pointer would be checked against the registered list of pointers. If the callee's pointer is missing, that module has not

a lookup can be done in $O(\log(n))$. It is even easily done with the help of the STL `set<>` container [11] or a `map<>`.

As usual, a simple solution as conditional compilation can be of help to remove the verification and speed up the execution time, but necessitate recompilation of the model code.

7.5 Third part: Semantic isolation of models, simulation context and simulators

We propose in this section to create a clearer line between elements we think it is important to isolate. This next solution is based on the idea of the previous section where it is used as a starting point. An example of instantiation will be shown at the end of this section, demonstration how simple addition to `sc_main` can clarify things into the mind of the end users. This will also bring some semantic standard to help the introspection of SystemC.

In Figure 7.20, we show a UML (Unified Modeling Language) [20] use case diagram that could be applied to any HDL (which can be simulated and verified) environment. This kind of diagram is typically used to ensure good communication between the customer (end user) and the analyst, ensuring agreement of the overall system functionality. UML being methodology independent [20], it does not specify anything about how to transfer this model into code, nonetheless, it is clear to us, only by looking at the diagram, that the end user model(s) justifiably takes an important place in the system. However, when looking into SystemC's source code, the model concept is absent.

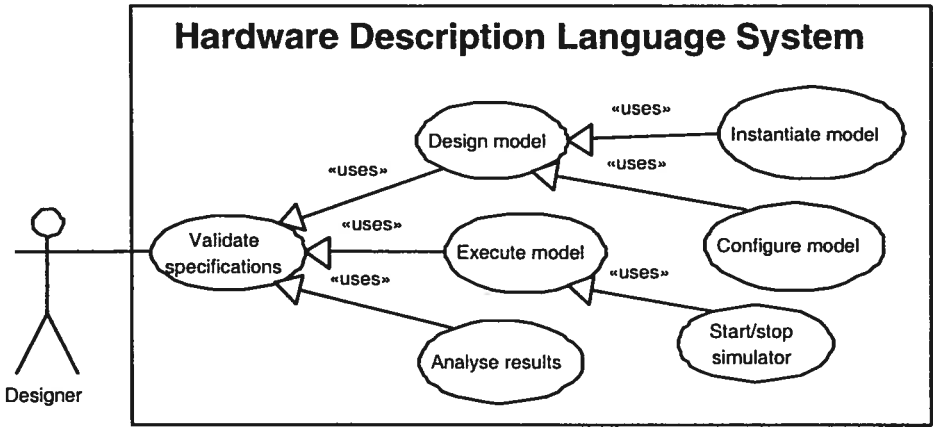


Figure 7.20: *Generic use case diagram applied to any simulation capable HDL*

1. A SystemC simulation is not currently required to have a top module. For example, in the multiprocessor paper (chapter 3) we could have put the five DLX stages globally into the `sc_main`. When would have come the time to instantiate 128 of this processor, we would had a hard time doing it if we would not had put the stages into one class easy to instantiate.

2. There is currently no distinction between a benchmark stimuli module, a benchmark verification module, a benchmark display module and the model itself. This fact has low impact on simulation where it is unimportant to know what is the purpose of the module, as long as we simulate it, but for a GUI or a synthesis tool, it get much important.

For at least these two reasons, should we be forced to put a description on the `sc_module`, we would have to admit the description is too large: "something that can be simulated". We think it is important to flag a model as being a `sc_model`,

Here is some suggestion that we think might clarify the relation between the end user model and the SystemC's core:

- The end user's model should always be referred in its whole by a single class. This way, this "model" could be instantiated at will. The "model" should have the potential to contain an interface to its external world in the likely case we would like to embed the model into a higher hierarchical model.

- The simulator should be well defined and extensible via polymorphism (by creating a base class interface and deriving new implementations), even if this means to supply only `start()` and `stop()` methods. We can easily imagine a simulator implementation that would queue the process update request in random order to verify that the model is well defined, or another implementation of the simulator transparently distributed over TCP/IP.

- The end user should be aware of what kind of simulator he is using (event driven, or some other MoC (Model of Computation) [17] if available) by letting him instantiate the simulator of his choice. Even if with SystemC there is, for the moment, only one kind of MoC simulator available this could change in the future and would be great to have a flexible system that could instantiate such features.

- The end user should be able to bind a specific simulator to a specific model.

- The end user should be able to specify which simulator he wants to synchronize inside an eventual multiple simulator environments.

Following these recommendations, here is a list of classes that would fulfill them

a module and be instantiated inside a larger model (the case where a processor can be the DUT (Design Under Test), or it can be part of a motherboard model under test). Would help in the distinction of a benchmark module and the DUT itself.

sc_simulator_base: should be an interface for any simulator implementation. Would instantiate a `sc_sim_context` to help him realize the simulation.

sc_sim_event_driven: would be the current implementation of SystemC's event-driven kernel.

sc_simcontext: (facultative) would have to be transformed into an interface. Facultative since it can be considered to be in a one-to-one relation with the specialization of the simulator class so it can be nested into the simulator class. Would be more useful if there is a need to "save" and "load" simulation contexts for example, or if we permit a simulator to control multiple independent models simultaneously, we could implement that one context match to one model but the solution does not require it.

sc_sim_context_event_driven: (facultative) would be the implementation of the transformed base class for the event driven simulator.

From the other perspective, it could be interesting to flag benchmark modules as being benchmarks (something as `sc_benchmark`), it could fit as a sibling of `sc_model` in Figure 7.19.

This new concept might bring some questions about what is to be considered as a model. One could argue that even a single register in our DLX model might be

would get automatically deactivated when the model would get connected to a larger one.

There is a lot of way of organizing the hierarchy depending on how we want to assign the semantic. For example, we could draw a scheme where the `sc_model` would be an organizing blocks containing only sub-modules and sub-models but it would not be permitted to contain executable processes. The modules would then be the leaf objects of the hierarchy, able to contain processes. We did not choose this scheme because we find it more pointlessly restrictive than the one we proposed.

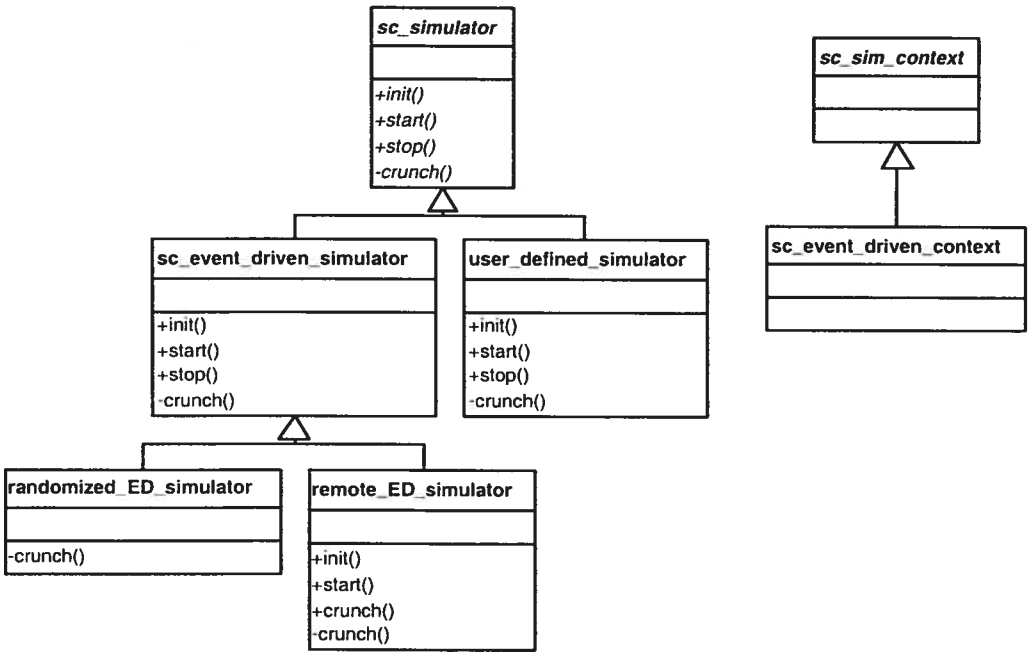


Figure 7.21: *Class diagram of new `sc_simulator` and `sc_sim_context` hierarchy*

In Figure 7.23 we show what a `sc_main` source code might look like when using such structure as in Figure 7.21 and Figure 7.22. The simulator (Figure 7.21) is now

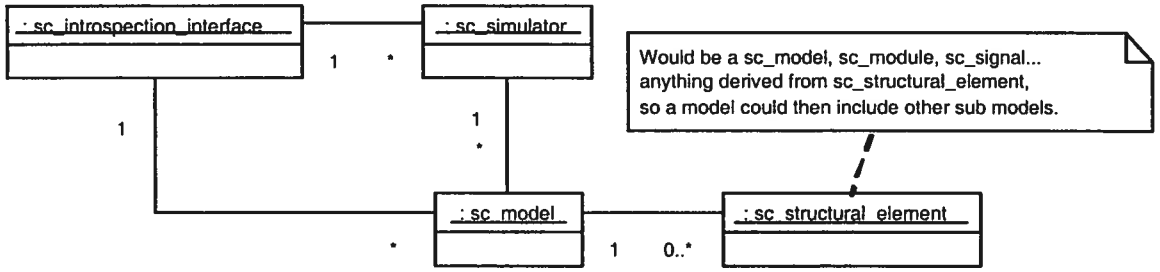


Figure 7.22: *Object diagram showing the instantiation possibilities of the new proposed structure*

same executable context and/or multiple models for a single simulator.

Line 7 and 8 of Figure 7.23 shows an intention of two DLX processors, line 11 and 12 creates two independent simulators that get bound to the model with lines 15 and 16. This would assign a simulation task to a simulator. In the case where multiple models is supported per simulator, we might want to instantiate only one simulator, controlling both models at the same time. We could even envision giving an IP address to a specialized simulator, asking the simulator to simulate the model on another machine. Line 19 would synchronize both simulator so that delta cycles are executed synchronously, enable cross validation of models during runtime.

7.6 Introspection and interoperability of the overall solutions

Figure 7.24 show an illustrative example of introspection using the Composition pattern for the structure and the Composition of the datatype. The line 4 shows an important element: a common entry point to introspection. Of course, we could con-

```
1  int sc_main (int argc , char *argv[])
2  {
3    //clock creation
4    sc_clock clock;
5
6    //creation of the models
7    sc_model *dlx_model_1 = new dlx_processor("DLX_Processor", clock);
8    sc_model *dlx_model_2 = new dlx_processor("DLX_Processor", clock);
9
10   //creation of the simulators
11   sc_event_driven_simulator sim_1 = new sc_event_driven_simulator();
12   sc_event_driven_simulator sim_2 = new sc_event_driven_simulator();
13
14   //bind the simulator with the model (could be passed to previous constructor)
15   sim_1.bind(dlx_model_1);
16   sim_2.bind(dlx_model_2);
17
18   //(optional) synchronize both simulations
19   sim_1.sync(sim_2);
20
21   //start both simulators (which are in sync)
22   sim_1.start();
23 }
```

Figure 7.23: *Semantically cleaner and clearer code example of the* **sc_main** *the end user would create*

part of `sc_simcontext`, a common interface would be more than useful.

Unfortunately, we did not implement this solution, so the code of Figure 7.24 has not been compiled and it is shown as an illustrative example. Because of the Composite pattern, both of the datatype and structure introspection can be done with recursive construction. Obviously, the Composite pattern generates object trees and the recursion is one of the simple methods when working on trees.

The information gathered this way is then ready to be used directly, in the case where we are dealing with homogeneous languages. When dealing with cross-language or cross-platform, an adapter could be generated to work with the `sc_introspection_interface`, transforming string queries into direct queries, back and forth.

The important part of this methodology is the common introspection point holding all the important information about the overall system. This interface would be close to the one we had presented at DATE (chapter 4) where the Observer pattern [5] was presented as a methodology for introspecting signals. As before, this introspection interface would serve well the SystemC community by acting not only as a Facade pattern (which reduces the coupling between systems by giving a common access point to a system) but also as a base class for an Adapter pattern (imagine deriving `sc_introspection_interface` into `my_TCP_introspection` which would listen on a TCP port and would be reimplementing all the introspecting methods accordingly).

By being a Facade, this enables a wrapper generator as SWIG [14] to process and deal with only one class, making the task easier for the tool and reducing the probability of failure as discussed previously in section 7.1.3. The fact that the interface is made available through a class is not a problem. One could think that this would block the access from a non object oriented language but this is not the case, since

```
1  //(considering initialization of model(s) has been done at this point)
2
3  //get the first of all models
4  sc_model *model = sc_introspection_interface::get_first_model();
5
6  //introspect the model
7  sc_string *model_name = model->get_name();
8  sc_string *model_type = model->get_type();
9  sc_structural_element *element = model->get_first_structural_element();
10
11 //get the hierarchy
12 while (element != NULL)
13 {
14   sc_structural_element *sub_element;
15   sc_signal<sc_datatype *> *signal;
16   sc_datatype *datatype;
17
18   //if it is a module...
19   switch (element.getID())
20   {
21     case SC_STRUCTURE_MODULE:
22       //... get the sub-elements (structural introspection)
23       sub_element = element->get_first_structural_element();
24       while (sub_element != NULL)
25       {
26         //...do something...
27         sub_element = element->get_next_structural_element();
28       }
29       break;
30
31     case SC_STRUCTURE_SIGNAL:
32       //... get the datatype (datatype introspection)
33       signal = element;
34       datatype = signal->read();
35       if (datatype->is_aggregate())
36         //... go down the datatype structure
37       else
38         //... do something with the value type
39       break;
40   }
41   element = model->get_next_structural_element();
```

The successful PLI interface used exactly the same kind of methodology where normalized global functions where made available to be called from another application linked with the simulator. The only distinction we made is that we enhance the methodology by reuniting the functions (methods) under a common access point (the class).

At the line 4 of Figure 7.24, we first ask the introspection interface to get the information about the first model of the execution context, suggesting there might be more than one inside available. When binding a model to a simulator, a link should be made between them using a pointer that would allow knowing which simulator is bound to which model, but this kind of information could be also available via the interface as the connection of Figure 7.22 shows. Back to the code, line 9 is a result of the Composite pattern where the first element fetch from the list could be any `sc_structural_element`. Once a module is found (line 19 and 21), we can introspect its content recursively (line 23) with the same method we used for introspecting the model. When signals are found (line 31), their data can be introspected using the first solution we presented (line 34 and 35).

7.7 Performance

Performance is a little issue when dealing with better OO construction that speedup the design phase. Even so, we must make a sanity check to be sure we are not entangling the program with clumsy slower construction and inspect the nature of the problem (if any) to see if it can be corrected.

We ran the original `pkt_switch` which is a 4×4 multicast helix packet switch

The `struct` declaration was replaced by a class deriving directly from `sc_aggregate`. The two `sc_int<>` where converted into pointers to fit our methodology but the four bool where not inserted into our introspection methodology because we had not created a wrapper for these natural type. All the `sc_signal<struct pkt>` were changed to `sc_signal<sc_datatype *>`, generalizing this way the helix.

As describe earlier (see section 7.3.6), during the first tests, the performances were awful, about 70% slower! We made the assertion that this was caused by the fact that we deriving directly form `sc_object`, in doing so, we were inheriting the registration into the SystemC's kernel and the unique string ID assigned to each instance of the `sc_object`. So we created an intermediate class to move some unwanted features in it (as described in 3.5). Although we did not implement the natural C++ types wrappers, although we did not measured introspection cost because it is not trivial to isolate the measure of the mechanism from the introspecting tool, we succeed in taking the performance loss down to about 5% which more than acceptable. However, this represents only a point on curve and should not be generalized. The readers should expect some more performance loss when the natural wrappers will be implemented and much more when doing introspection.

7.8 Conclusion

We presented different methodologies that are approved by the software community (through the patterns design). We demonstrate the use of the different patterns through simple examples which we proved by the SystemC's modifications that are easily done.

of the common rooted datatype, at the cost of loosing the immediate legacy code utilization.

The other solutions of the second and third part where we present a better structural introspection and semantics clarification requires also some adaptation to the end user code but nothing drastic. On the contrary, it brings better Object-Oriented coding style, a much clearer semantic by modifying the structure of the model (not the internal code).

Even though the OSCI does not adhere to our philosophy, the solutions we presented can be used in a proprietary interface which, of course, would not evolve with SystemC at the same ease of a one that would be maintained directly by the consortium.

7.9 Future work

We will implement some useful features we did not produce for our example (Visitor "accept" method, type conversion methods (`to_int()`, `to_string()`, `to_float()`, ...), legacy structure import/export mechanism (`convert_from()`, `convert_to()`), ...). We will also look at the interaction between the unified type and the unification of modules that could be possible using the same kind of methodology. The ports and the signals should be modified to accept only `sc_datatype`.

Here are some design patterns [5] that could help in increasing the capabilities of our methodology:

The Builder pattern: permits to construct complex objects in parts (helps building Composites).

overall structure can be useful for creating prototypes.

Adapter pattern: This pattern exists in two different forms (Compositional and Multiple Inheritance). Can help to wrap foreign user datatypes.

Visitor pattern: permits to delegate some responsibilities to a single class instead of what would normally have to be implemented through out the whole hierarchy. It is also used to add new functionality to a set of related classes without having to modify the root classes.

Facade pattern: give a common access point to a system, reducing the coupling between systems.

Acknowledgments: We would like to thank James Lapalme for his discussions and corrections.

7.A Appendix A

"Knowledge scope" of interoperable environments

In this section, we present a few ways of realizing the interoperability between models. The non-exhaustive list of requirements will be built in sorted order from the simplest to realize up to the hardest but also, in a balanced way, from the most restrictive up to the most flexible and interoperable.

Obviously, a SystemC user model will be "aware" of the constructions of the SystemC library and the C++ language (Figure 7.25) and by being built "over them", it will be able to use the available features of those entities. By using an introspection

at run time. Introspection brings a supplementary execution cost, so it is usually not really recommended to use this mechanism on a model (or program) when it can be avoided and replaced by something better, as polymorphic constructions (see section 7.1.2). The model will likely have a "knowledge scope" which obviously includes itself at 100%, where the programmer who build the model knows everything of it and can program the model accordingly.

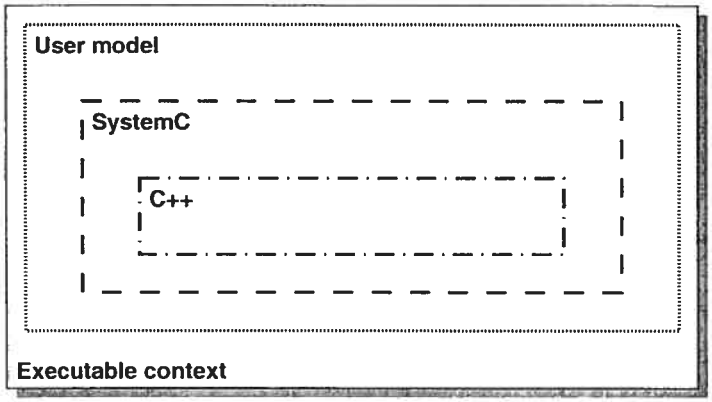


Figure 7.25: *"Knowledge scope" of a single SystemC model versus the scope of SystemC itself*

By opposition, two different models (or one model and a tool), built by two different parties will have distinct constructions (Figure 7.26). As illustrated, although they might be inside the same simulation context (compiled and linked together), they are knowing a priori nothing about each other, they will be "aware" only of their sub-layer (C++ and SystemC). Should they communicate and cooperate together, there are two choices:

- Whether we code these models as if they were one, in which case the scheme of

one.

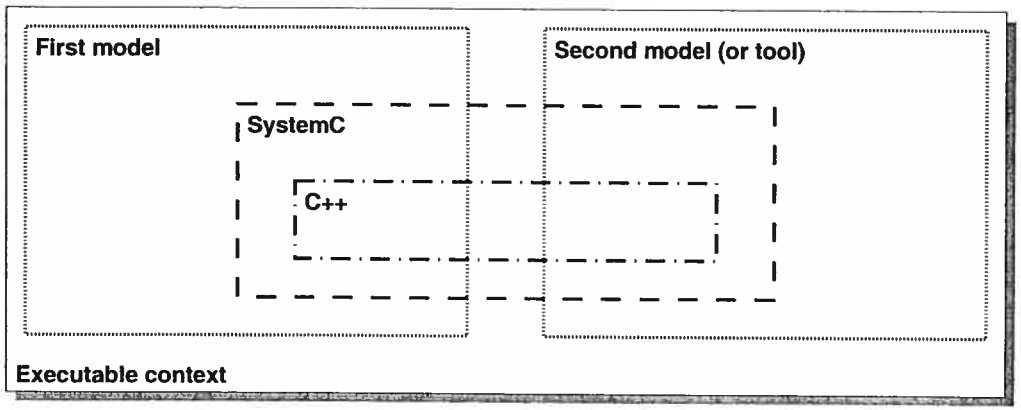


Figure 7.26: *"Knowledge scope" of a SystemC model versus the "Knowledge scope" of another model (or tool)*

C++ supply the RTTI mechanism (Real Time Type Information, see section 7.1.2) which brings introspection to datatypes but does not bring a solution to the structural analysis. Unfortunately, yet SystemC has little capabilities to help someone who would like do introspection (see section 7.2).

In Figure 7.27 we show that we could add a communication library over SystemC to enable two models inside the same execution context to communicate together.

If there is the need for more flexibility, there is a way to compile each model separately and have the communication/synchronization passed over some interprocess channel (TCP/IP, shared memory, DDE (Dynamic Data Exchange), ...) (Figure 7.28 and Figure 7.29). At least in the case of SystemC, synchronization of the simulators is vital to keep the validity of the event-driven simulation paradigm. The figures are similar because the techniques are quite similar where the only distinction is that the

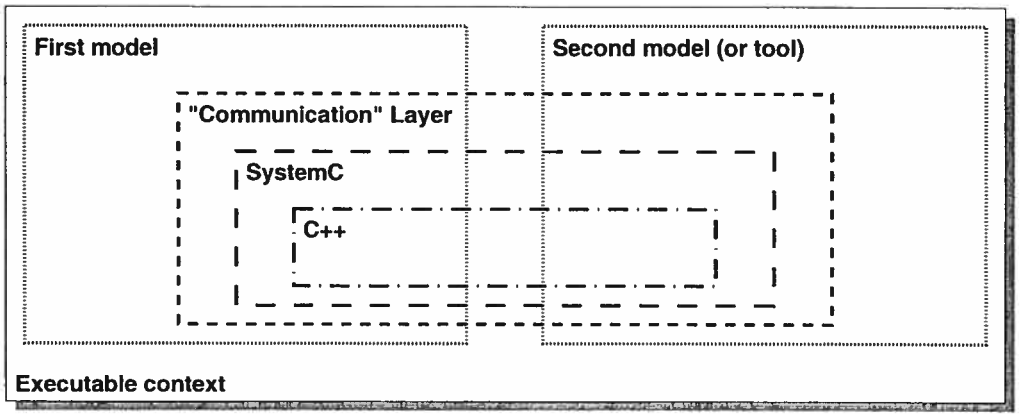
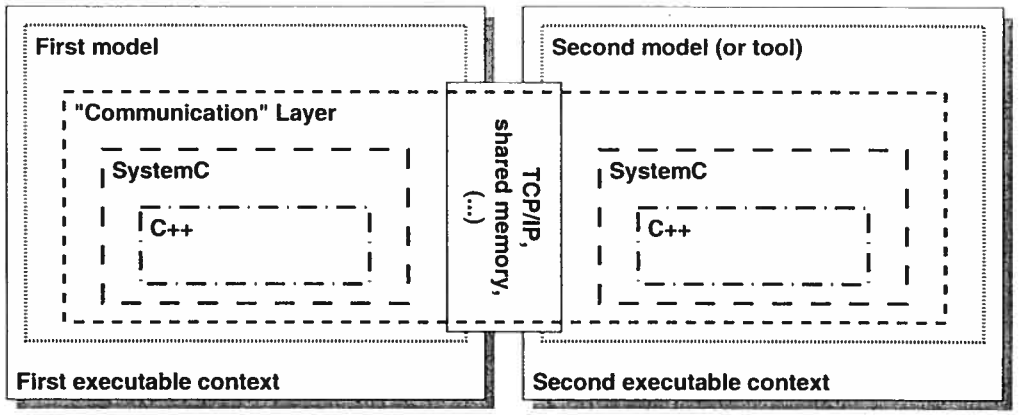


Figure 7.27: *Interoperability of two models in the same simulation context; "Knowledge scope" of a SystemC models include a "communication" layer which passes the "knowledge" between models*



While the datatype introspection mechanism of SCV was interesting for the case of Figure 7.26 and Figure 7.27, it is rather cumbersome for the situations of the later figures where executables are constructed apart (see section 7.1.2 for more details).

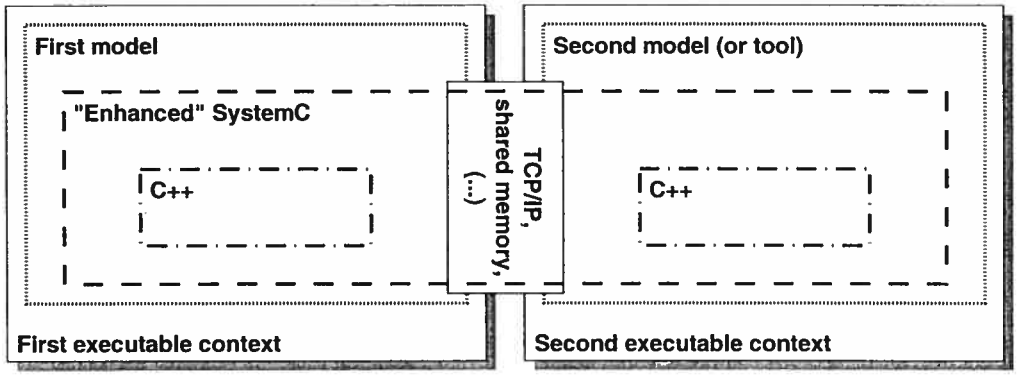


Figure 7.29: *Second approach; augmenting SystemC with enough introspection capabilities to enable homogeneous HDL interoperability between models and/or distributed simulations*

We show in Figure 7.30 and Figure 7.31 that interoperability with foreign language (such as VHDL or Verilog and which can be called co-simulation in the HDL domain) is possible through at least two different schemes. Wrapping the foreign model is a first solution (Figure 7.30) where some code (in some language, not necessarily the language of the foreign model) will be written to add introspection and interoperable control functionality (start, stop, ...). This might allow a SystemC model to gain control and collaborate with a foreign model but a one way wrapper usually does not allow the other way around.

One of the greatest difficulty of creating such interface for a low level communication and collaboration requires converting datatypes between languages. An integer in C++ does not have the same internal representation as in VHDL and the data

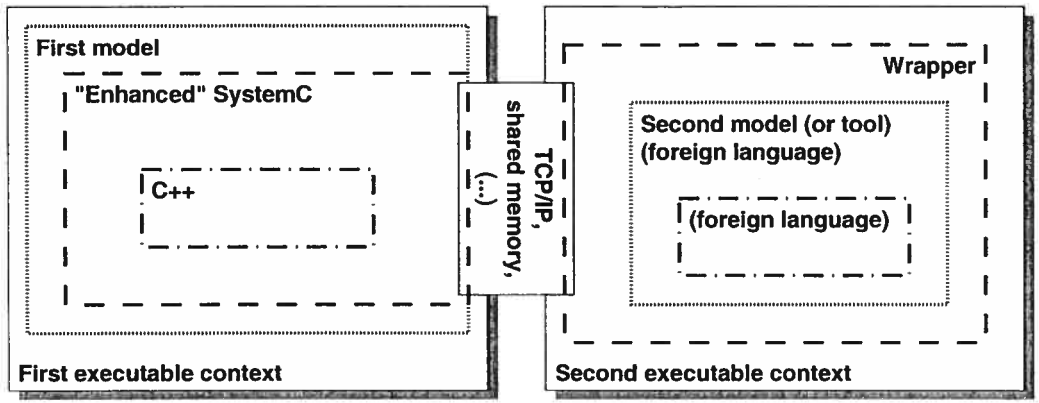


Figure 7.30: *Possible "knowledge scope" of heterogeneous HDL interoperability; a not so trivial one-way wrapper is needed between the master and the foreign slave model*

A two way interface as shown in Figure 7.31 would allow this possibility. PLI (Programming Interface Language) (as describe in section 7.1.2) is a two-way interface which permits to set call back functions and control the simulation of Verilog. It could be used to control an external application from the Verilog model. The greatest introspection library for SystemC would be two-way such as presented in the figure 7.31 of this section but at a communication cost. A SystemC model could then control and be controlled by other models/tools. Interprocess interoperability is implicit when intraprocess interoperability is available; if we can send the data over some TCP/IP channel, we can logically get it locally. If the communication layer implement directly the introspection, a worst case scenario would be to make the process communicate with itself over the interprocess channel, "emulating" this way an intraprocess communication. Obviously, the fastest approach is to use homogeneous interoperability (no need to convert anything) inside the same executable context (no need for interprocess communication). Of course, using a TCP/IP has its advantages for it can

Of course, such things as TCP/IP will help in realizing a cross platform communication but it will not necessarily remove the needs of adapting the data structure for example, where a 32 bits integer on a first 32 bits machine might need to be transferred into a 64 bits machine format, even though we are using the same language. Knowing this, we could consider the cross platform adaptation difficulty to be more or less of the same level of difficulty as the cross language one.

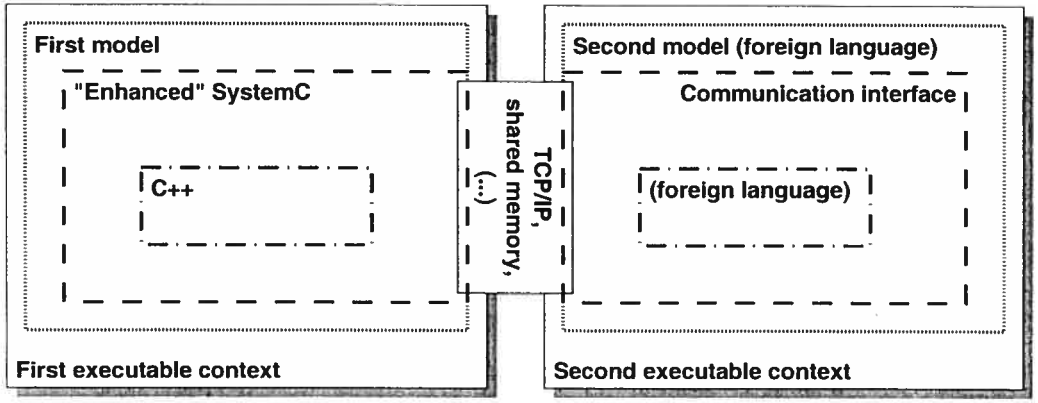


Figure 7.31: *The distinction we can make between a wrapper and a communication interface; both models can be master or slave and they typically gain "knowledge" of each other via the communication interface and the introspection*

There are other ways of constructing interoperability and introspection, we presented only a few. As for example, a way of doing a little bit different than the approaches presented in this section is to use parsing and static analysis to create wrappers.

Bibliography

[1] James Lapalme, El Mostapha Aboulhamid, Gabriela Nicolescu, Luc Charest, François-Raymond Boyer, Jean-Pierre David, and Guy Bois. ESys.Net: a new solution for embedded systems modeling and simulation. In *Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools*, pages 107–114. ACM Press, 2004.

[2] Open SystemC Initiative (OSCI). Systemc home page, 2001. `http://www.systemc.org`.

[3] Synopsys. CoCentric SystemC compiler, 2003. `http://www.synopsys.com/products/cocentric_systemC/cocentric_systemC.html`.

[4] Open SystemC Initiative (OSCI). Scv (systemc verification library) home page, 2002. `http://www.systemc.org/projects/scv`.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, 1994.

[6] Luc Charest and El Mostapha Aboulhamid. A VHDL/SystemC comparison in handling design reuse. In *Proceedings of 2002 International Workshop on System-*

[8] Pierre Paulin, Chuck Pilkington, and Essaid Bensoudane. Stepnp: A system-level exploration platform for network processors. *IEEE Des. Test*, 19(6):17–26, 2002.

[9] Frederic Doucet, Sandeep Shukla, and Rajesh Gupta. Introspection in system-level language frameworks: Meta-level vs. integrated. In *Design Automation and Test in Europe Conference & Exhibition*, pages 382–387, Munich, Germany, 2003. IEEE Computer Society.

[10] Stuart Sutherland. *The VERILOG PLI Handbook: A User's Guide and Comprehensive Reference on the VERILOG Programming Language Interface.* Kluwer Academic Publishers, 1999.

[11] Silicon Graphics Computer Systems Inc. Standard template library programmer's guide, 2001 1993. `http://www.sgi.com/tech/stl`.

[12] Sun Microsystems. Java API documentation. `http://java.sun.com`.

[13] Microsoft. .Net API documentation. `http://msdn.microsoft.com`.

[14] Dave Beazley and et al. SWIG (Simplified Wrapper and Interface Generator) home page. `http://www.swig.org`.

[15] Object Management Group (OMG). CORBA specification. `http://www.omg.org`.

[16] Open SystemC Initiative (OSCI). Systemc lrm (library reference manual). `http://www.systemc.org/projects/systemc/document/SystemC_v201_LRM`.

[17] Edward Lee. Ptolemy web site, 2003. `http://ptolemy.eecs.berkeley.edu`.

[20] Object Management Group (OMG). UML specification, 2003. `http://www.omg.org`.

Chapitre 8

Conclusion

Il apparaît clair, à la lumière des éléments proposés dans cet ouvrage, que la modélisation orientée objet est en plein essor dans le domaine du génie matériel, et ce, pour un certain temps. L'utilité des motifs de conception fut démontrée, que ce soit pour la construction d'outils qui serviront la cause du domaine ou encore pour le développement des modèles eux-mêmes ; de bons ingénieurs systèmes ne sauraient s'en passer.

Bien que les outils de synthèse ne la permette pas encore totalement, il fut cependant démontré qu'une approche orientée objet s'avère être non seulement possible, mais surtout, utile pour le design de modèles matériels à haut niveau. Elle apporte beaucoup plus de solutions aux problèmes de programmation, qui furent résolus depuis longtemps par le génie logiciel, que de problèmes eux-mêmes. Avec l'aide d'un modèle de processeur DLX, nous avons montré l'efficacité de la programmation orientée objet pour le monde du matériel.

Dans le chapitre 3 « SystemC Performance Evaluation Using a Pipelined DLX Mul-

facilitant ainsi l'exploration architecturale.

Lors d'un stage à STMicroelectronics, nous avons réussi à établir que SystemC est un outil performant par rapport à VHDL, lorsque nous avons effectué les comparaisons avec des jeux de tests de niveaux d'abstractions similaires. Étant donné que les modèles n'étaient pas formellement comparables, la performance de SystemC fut, au mieux évoquée, mais généralement omise de différents articles. Le modèle de processeur que nous possédions, écrit pour SystemC, était un modèle complexe (pipeliné à cinq étages) mais pourtant s'exécutait beaucoup plus rapidement que le modèle VHDL du même processeur, non pipeliné. Ceci prouve que, même sans utiliser une représentation d'un niveau d'abstraction supérieur, SystemC est plus rapide. Il est possible d'expliquer que ce fait est probablement dû, en partie, au langage C, qui se trouve très près du langage d'assemblage et au compilateur g++ de GNU, qui optimise relativement bien le C++ (donc le SystemC). L'article du chapitre 3 démontre clairement que les performances de SystemC s'amenuisent avec l'évolution des versions, ceci étant dû logiquement soit aux ajouts de fonctionnalités de la bibliothèque, soit à un mauvais choix de modifications apportées au code source existant.

Le chapitre 4 « A Methodology for Interfacing SystemC with a Third Party Software » a permis d'apporter une vision génie logiciel en introduisant, au domaine de la conception matérielle, non seulement la programmation orientée objet, mais surtout les motifs de conception. Bien que nous ayons été forcés de modifier SystemC afin de respecter, en autre, l'objectif de minimiser l'interférence avec le code de l'utilisateur final, nous avons également respecté l'objectif qui visait à minimiser les modifications apportées à SystemC. Par conséquent, l'article montre qu'il est facile d'utiliser SystemC avec des programmathèques C/C++ existantes, et ce, sans trop de coûts de

SystemC. Cette dernière version permet les canaux abstraits de communication qui ne laissent aucune trace dans le système quant aux connexions entre les différents modules, il devient donc difficile de gérer ou même de montrer les déplacements d'informations à l'intérieur du système.

À l'intérieur de l'article du chapitre 5 « A VHDL/SystemC Comparison in Handling Design Reuse », se trouve une comparaison, en plusieurs points, entre SystemC et l'un de ses prédécesseurs des plus connus dans le domaine : VHDL. Un rapprochement des paradigmes y est présenté, ce qui permet de réaliser à quel point la puissance de la philosophie orientée objet apporte une aisance de programmation et une réutilisation de code accrue.

Les motifs de conception viennent avec une méthode orientée objet, car, sans eux, les designers s'égarent et perdent leur temps à réinventer la roue. Les motifs de conception s'appliquent bien au monde du matériel, il faut simplement prendre le temps d'analyser nos besoins, de trouver des noms aux solutions existantes et finalement les classer dans un livre qui deviendrait la référence du domaine en terme de techniques de modélisation du matériel.

Une meilleure couverture de la programmation orientée objet et des motifs de conception furent présentés dans le chapitre 6 « Applying Multi-Paradigm and Design Pattern Approaches to Hardware/Software Design and Reuse ». Nous prouvons que des solutions orientées objet peuvent être appliquées concrètement aux fins du génie matériel. Ayant illustré l'utilité des motifs de conception appliqués aux outils, avec les chapitres précédents, cette fois, nous montrons que de nouveaux motifs de conceptions peuvent être facilement créés pour le domaine du matériel, et par la suite, mis en œuvre par l'utilisateur dans ses modèles.

types et l'utilisation du motif « composite », non seulement permettent d'introduire un niveau d'abstraction intermédiaire, mais facilitent également l'introspection et l'interopérabilité.

En reprenant chacun des objectifs présentés en introduction dans la section 1.9 :

Améliorer la méthodologie de conception :

- *Apporter au domaine du génie matériel une partie de la vision du génie logiciel.*
  À l'intérieur de tous les chapitres, nous avons utilisé la programmation orientée objet pour les exemples et les modifications du noyau de SystemC. Nous avons également illustré la structure des codes sources par le biais de diagrammes UML, qui est devenu un langage de modélisation incontournable dans le domaine logiciel.
- *Présenter des solutions orientées objet qui facilitent la flexibilité et la réutilisation des modèles des utilisateurs.*
  Avec la topologie de classes présentée dans la figure 3.2 du chapitre 3, nous avons réalisé un processeur et un système de communication concrets, qui utilisent l'héritage et le polymorphisme pour réutiliser les méthodes et propriétés des structures de base.
  Nous avons également montré l'utilité du polymorphisme, à l'intérieur de tous les autres chapitres. Plus spécifiquement, à l'intérieur du chapitre 7, nous avons montré qu'il est parfois souhaitable de substituer l'utilisation d'un mécanisme d'introspection au polymorphisme.
  Bien qu'elle fût utilisée probablement dans chacune des expérimentations de tous les chapitres, la surcharge de méthodes fut explicitement traitée dans les chapitres 5 et 6.

de les encapsuler dans une classe pour, par la suite, pouvoir les instancier à volonté.

- *Introduire les motifs de conceptions qui proviennent du domaine du génie logiciel et qui apportent des esquisses de solutions basées sur l'expérience.*

  Nous avons introduit dans le chapitre 4 les motifs de conception qui proviennent du génie logiciel en les utilisant pour le développement d'outils. Dans les chapitres 5, 6 et 7, nous les présentons comme outils pour l'ingénieur matériel. À l'intérieur du chapitre 6, nous montrons comment créer de nouveaux motifs appliqués au design matériel et comment les documenter. Pour le chapitre 7, nous présentons le motif de l'Observateur à la fois comme technique pour aider le développement d'outils de modélisation et à la fois comme outils pour aider le développement de modèles.

- *Encourager le passage du niveau de modélisation RTL à un plus haut niveau d'abstraction (comme le niveau transactionnel) dans le but de réduire le temps de réalisation du modèle, et ce, afin de valider plus rapidement les premières assertions des spécifications.*

  Bien que nous ayons montré souvent des exemples de niveau RTL, et ce, afin de rejoindre plus facilement les ingénieurs matériel étant habitués à la modélisation de style VHDL, le chapitre 7 montre que la programmation orientée objet peut être utilisée pour modéliser à plus haut niveau. Nous introduisons un niveau intermédiaire entre le niveau RTL (communication par signaux détaillés) et un niveau transactionnel (communication par appels de méthodes et passage de paramètres). Notre niveau intermédiaire utilise des structures pour regrouper les signaux entre eux et ainsi pouvoir envoyer les

quelque peu de la problématique entre la création dynamique du modèle SystemC et la synthèse. Quelques approches possibles sont expliquées.

Valider le choix d'utilisation de SystemC :

- *Analyser le code source du noyau de l'engin de la bibliothèque.*

  Lorsque nous avons réalisé l'article du chapitre 3, nous avons constaté une dégradation de performance. Nous avons dû analyser le code source du noyau de SystemC afin d'en découvrir les causes. Nous avons dû également analyser le noyau de la bibliothèque afin de pouvoir proposer des modifications appropriées à nos besoins aux chapitres 4 et 7.

- *Vérifier la performance.*

  Cet objectif fut principalement traité à l'intérieur du chapitre 3 où nous avons comparé les performances de différentes versions de SystemC. Nous avons, en autre, dissocié les performances de différentes constructions (`SC_METHOD`s et `SC_THREAD`s). Nous avons montré que les `SC_METHOD`s sont beaucoup plus rapides que les `SC_THREAD`s et constituent, par conséquent, un meilleur choix lorsque les threads peuvent être évités.

- *Tester la variabilité dimensionnelle.*

  Nous avons testé la variabilité dimensionnelle avec l'article du chapitre 3, où nous avons fait croître, entre les différentes exécutions, le nombre de processeurs DLXs à l'intérieur de la même simulation. Nous avons par la suite présenté des résultats qui montrent que l'on peut aisément utiliser SystemC pour les besoins de modélisation d'un multiprocesseur.

- *S'assurer du bon fonctionnement du système et de la stabilité du produit.*

  Cet objectif fut réalisé implicitement à l'intérieur de chacun des chapitres par le

multiprocesseur).

Améliorer et modifier SystemC :

- *Moderniser la bibliothèque dans le but de faciliter la méthodologie de développement.*

  Nous proposons à l'intérieur du chapitre 7 des modifications à SystemC qui permettraient un niveau intermédiaire qui accélère l'élaboration et la réutilisation des modèles.

- *Revoir la structure orientée objet de la bibliothèque.*

  Nous révisons la structure même de la bibliothèque en suggérant quelques modifications de classes existantes dans le chapitre 7. Nous avons prôné une solution orientée objet s'intégrant bien à la bibliothèque.

- *Parfaire les mécanismes d'introspections et d'interopérabilité de la bibliothèque.*

  Le chapitre 4 constitue, en soi, une proposition de modification à SystemC visant à aider la création et l'intégration d'outils avec SystemC.

  Le sujet principal du chapitre 7 traite également des modifications que nous pourrions apporter à la bibliothèque, dans le but d'améliorer l'introspection et l'interopérabilité de celle-ci.

Depuis le début, un des messages que nous envoyons au consortium de SystemC est qu'une réflectivité, soit la possibilité de découvrir de manière dynamique les éléments d'un modèle utilisateur de la bibliothèque est primordiale. Bien que SCV fût créé plus ou moins dans cette intention, jusqu'à maintenant, très peu d'informations sont disponibles sur le modèle après son élaboration dynamique, ce qui ne devrait pas être le cas. Avec les récents efforts de la normalisation du langage, il est logique et conséquent que l'OSCI, ayant figé les développements de la bibliothèque, ne cherche

système comme ESys.Net s'insère en tant que phase subséquente possible vers une modélisation plus abstraite et automatisée du matériel.

Nous avons cependant montré que SystemC est un système stable et utilisable pour des applications concrètes. Nous avons également prouvé qu'il est avantageux d'utiliser certains concepts provenant du domaine du génie logiciel pour favoriser l'élaboration de meilleurs modèles. Nous avons apporté diverses solutions pour améliorer non seulement la bibliothèque, mais également la méthodologie de conception en place.

Nous présenterons dans le chapitre 9 divers aspects sur lesquels nous baserons nos recherches futures, pouvant, à notre avis, favoriser l'évolution du domaine matériel.

Chapitre 9

Travaux futurs

Nous allons continuer d'analyser les possibilités d'interconnections entre SystemC et notre nouveau système ESys.Net [32], soit l'interopérabilité entre une bibliothèque C++ et l'environnement .Net de Microsoft qui regroupe différents langages de programmation. Un sujet d'étude également proposé est de tenter de trouver de meilleures méthodes pour rattacher de l'information sur la nature des modules. Par exemple, distinguer un module matériel d'un module logiciel et mieux communiquer d'un niveau d'abstraction à un autre.

Nous aimerions également amorcer des recherches sur XML (eXtensible Markup Language) [33], XMI (XML Metadata Interchage) [34] et les outils CASHE (Computer-Aided Software-Hardware Engineering) [35] afin de trouver un point d'encrage commun ; soit, une manière de passer de VHDL à SystemC à UML [13], tout en gardant un lien entre les éléments échangés. En parallèle à cette recherche, nous allons explorer comment mieux adapter UML au design matériel, comment représenter des données sans tenir compte du niveau d'abstraction ou du langage par exemple.

Bibliographie

[1] Robert Paško, Serge Vernalde, and Patrick Schaumont. Techniques to evolve a C++ based system design language. In *Design Automation and Test in Europe Conference & Exhibition*, pages 302–309, Paris, France, 2002. IEEE Computer Society.

[2] Robertas Damaševičius and Vytautas Štuikys. High level design of soft IPs using C++ and SystemC. *Information technology and control*, 25(4) :54–64, 2002.

[3] David A. Penry and David I. August. Optimizations for a simulator construction system supporting reusable components. In *40th Conference on Design Automation Conference*, pages 926–931, Anaheim, California, USA, June 2003.

[4] Robertas Damaševičius, Giedrius Majauskas, and Vytautas Stuikys. Application of design patterns for hardware design. In *40th Conference on Design Automation Conference*, pages 48–53, Anaheim, California, USA, June 2003. ACM Press.

[5] Robertas Damaševičius. A subset-based comparison of main design languages. *Information technology and control*, 30(1) :48–55, 2004.

[6] Manish Vachharajani, Neil Vachharajani, and David I. August. The liberty structural specification language : a high-level modeling language for component reuse.

[8] Synopsys. CoCentric SystemC compiler, 2003. `http://www.synopsys.com/products/cocentric_systemC/cocentric_systemC.html`.

[9] The ODETTE Project. SystemC plus methodology LRM, 2003. `http://odette.offis.de`.

[10] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software.* Addison Wesley, 1994.

[11] Eike Grimpe and Frank Oppenheimer. Object-oriented high level synthesis based on systemc. In *The 8th IEEE International Conference on Electronics, Circuits and Systems,*, pages 529–534, 2001.

[12] Sowmitri Swamy, Arthur Molin, and Burt Covnot. OO-VHDL : Object-Oriented extensions to VHDL. *Computer*, 28(10) :18–26, 1995. 0018-9162/95/$04.00 (c) 1995 IEEE Features.

[13] Object Management Group (OMG). UML specification, 2003. `http://www.omg.org`.

[14] Stuart Sutherland. SystemVerilog tutorial, 2003. `http://www.ovi.org`.

[15] *IEEE Standard VHDL Language Reference Manual.* IEEE, 1076,2000 edition, 2000.

[16] Stuart Sutherland. *The VERILOG PLI Handbook : A User's Guide and Comprehensive Reference on the VERILOG Programming Language Interface.* Kluwer Academic Publishers, 1999.

[17] Edward Lee. Ptolemy web site, 2003. `http://ptolemy.eecs.berkeley.edu`.

[18] Object Management Group (OMG). CORBA specification. `http://www.omg.org`.

[21] Kjetil Svarstad, Nezih Ben-Fredj, Gabriela Nicolescu, and Ahmed A. Jerraya. A higher level system communication model for object-oriented specification and design of embedded systems. In *Proceedings of the 2001 conference on Asia South Pacific design automation*, pages 69–77. ACM Press, 2001.

[22] Pontus Aström, Stefan Johansson, and Peter Nilsson. Application of software design patterns to dsp library design. In *14th International Symposium on System Synthesis*, Montréal, Québec, Canada, 2001.

[23] Frederic Doucet, Rajesh Gupta, Masato Otsuka, Patrick Schaumont, and Sandeep Shukla. Interoperability as a design issue in C++ based modeling environments. In *Proceedings of the 14th international symposium on Systems synthesis*, pages 87–92. ACM Press, 2001.

[24] Frederic Doucet, Sandeep Shukla, and Rajesh Gupta. An environment for dynamic component composition for efficient co-design. In *Design Automation and Test in Europe Conference & Exhibition*, pages 736–743, Paris, France, 2002. IEEE Computer Society.

[25] Frederic Doucet, Sandeep Shukla, and Rajesh Gupta. Introspection in system-level language frameworks : Meta-level vs. integrated. In *Design Automation and Test in Europe Conference & Exhibition*, pages 382–387, Munich, Germany, 2003. IEEE Computer Society.

[26] Frederic Doucet, Sandeep Shukla, and Rajesh Gupta. The Balboa project home page. `http://www.ics.uci.edu/~balboa`.

[27] Peter N. Green and Martyn D. Edwards. The modelling of embedded systems using HASoC. In *Design Automation and Test in Europe Conference & Exhibition*,

[30] Luc Charest, El Mostapha Aboulhamid, and Guy Bois. Using design patterns for type unification and introspection in systemc. In *Proceedings of 2004 International Workshop on System-on-Chip for Real-Time Application*, pages 19–21, Banff, Canada, July 2004. IEEE Computer Society.

[31] Marco Caldari, MMassimo Conti, Paolo Crippa, Giuliano Marozzi, Fabio Di Gennaro, Simone Orcioni, and Claudio Turchetti. Systemc modeling of a bluetooth transceiver : dynamic management of packet type in a noisy channel. In *Design Automation and Test in Europe Conference & Exhibition*, pages 214–219, Munich, Germany, 2003. IEEE Computer Society.

[32] James Lapalme, El Mostapha Aboulhamid, Gabriela Nicolescu, Luc Charest, François-Raymond Boyer, Jean-Pierre David, and Guy Bois. ESys.Net : a new solution for embedded systems modeling and simulation. In *Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools*, pages 107–114. ACM Press, 2004.

[33] World Wide Web Consortium (W3C). XML specification, 2003. `http://www.w3c.org`.

[34] Object Management Group (OMG). XMI specification, 2003. `http://www.omg.org`.

[35] M. Calha, J.P. Teixeira, and I.C. Teixeira. Hw/sw specification using oom techniques. In *7th IEEE International Workshop on Rapid System Prototyping*, pages 96–101, June 1996.