2M11.2902.4

Université de Montréal

Visualisation interactive des résultats de simulation de matériel modélisé avec SystemC

Par

Michel Reid

Département d'informatique et recherche opérationnelle

Faculté des arts et des sciences

Mémoire présenté à la faculté des études supérieures
en vue de l'obtention du grade de
Maître ès sciences (M.Sc.)
en informatique

Août, 2001

© Michel Reid, 2001

QA
76
N54
2001
v.032

Université de Montréal

Faculté des études supérieures

ce mémoire intitulé

Visualisation interactive des résultats de simulation de matériel modélisé avec SystemC

présenté par :

Michel Reid

a été évalué par un jury composé des personnes suivantes

Michel Boyer	président-rapporteur
El Mostapha Aboulhamid	directeur
Guy Bois	co-directeur
Houari Sahraoui	membre du jury

Mémoire accepté le : 12 octobre 2001

Sommaire

La compétition féroce entre les fabricants de produits électroniques, dont l'évolution est extrêmement rapide, pousse vers des temps de développement très courts et une complexité toujours grandissante. Les méthodologies existantes ne suffisent plus à la demande et de nouvelles méthodologies sont proposées pour répondre à ces besoins.

L'une d'entre elles est la conception au niveau système (*system level design*) qui nécessite que la fonctionnalité du système soit modélisée, au tout début du processus de conception, dans un langage pouvant nous fournir un modèle exécutable. Parmis les langages qui se disputent cette place, il y a le SystemC.

Lancé en septembre 1999 par les compagnies, Synopsys, CoWare et Frontier Design, SystemC est une librairie, au code source ouvert en C++, pour la conception au niveau système, de systèmes à composantes logicielles et matérielles, ainsi que l'échange de propriété intellectuelle.

Lorsque le modèle, écrit en SystemC est compilé, l'application exécutable qui en résulte sert à la simulation, dans le but de valider le modèle. Pour le moment, il n'y a pas d'interface usager graphique officielle pour contrôler et visualiser les résultats de la simulation. La visualisation des résultats est soit textuelle, soit disponible uniquement à la fin de la simulation.

Ce mémoire présente notre expérimentation dans le but de construire une interface usager graphique pour la simulation de SystemC.

Pour ce faire, nous avons utilisé une approche orientée objet, de manière à augmenter la réutilisation de notre application, pour créer une interface de communication permettant l'encapsulation des deux composantes, soit la simulation de SystemC et l'application interface usager graphique. Cette interface de communication permet l'évolution indépendante de ces deux composantes.

Nous avons pris bien soin de ne pas déplacer le fardeau du travail de construction de l'interface usager vers l'utilisateur du SystemC, soit le concepteur de système, en prenant soin de ne pas (ou peu) lui imposer de nouvelle syntaxe. De plus, nous avons construit l'interface entre les deux composantes de façon à ce que tout modèle en SystemC puisse être simulé sans utiliser notre interface graphique et ce, sans ralentissement significatif dû à notre interface de communication.

En plus de la flexibilité d'utiliser ou non notre interface graphique, nous offrons à l'usager la possibilité de choisir, au moment de l'exécution, les signaux qu'il veut voir et de changer sa sélection à tout moment au cours de la simulation.

Notre approche peut aussi être généralisée pour lier, avec la simulation SystemC, d'autres types d'applications, tout en permettant une évolution indépendante des deux partis.

Mots clés

SystemC, langages de descriptions de matériel, interface usager graphique, approche orienté objet, conception au niveau système.

Table des matières

Liste des figures vii

Liste des sigles viii

Chapitre1. Introduction 1

Chapitre 2. Langages de description de matériels 5

2.1 Verilog 7
2.2 VHDL 8
2.3 C/C++ 8
2.3.1 SpecC *9*
2.3.2 Cynlib *10*
2.3.3 SystemC *10*
2.4 Superlog 12
2.5 UML 13

Chapitre 3. Problématique 14

Chapitre 4. La librairie SystemC 16

4.1 Modules 19
4.2 Types de données 20
4.3 Ports et signaux 22
4.4 Processus 23
4.4.1 sc_main *25*
4.5 Architecture du SystemC 26
4.5.1 sc_object *27*
4.5.2 sc_module *28*
4.5.3 sc_port *28*
4.5.4 sc_signal_base *29*
4.5.5 sc_signal *29*
4.5.6 sc_object_manager *29*
4.5.7 sc_simcontext *29*

Chapitre 5. La librairie graphique QT 31

5.1 Communication inter-objets 32
5.2 Internationalisation 33
5.3 Widgets 34

Chapitre 6. Méthodologie 36

6.1 Approche orientée objet et design patterns 36
6.2 Notre implémentation 39
6.3 Description de la classe interface 41
6.4 Interface graphique 43
6.5 Comparaisons avec l'interface graphique pour Win32 44
6.6 Construction de l'interface usager graphique (GUI) 46
6.6.1 Prototype de l'Interface Usager Graphique 50
6.6.2 Changements requis au code du modèle de l'usager 51

Chapitre 7. Coût en terme de vitesse de simulation 52

Chapitre 8. Changements annoncés pour la version 2.0 de SystemC 57

8.1 Modélisation du temps de simulation 57
8.2 Sensibilité dynamique et évènements 58
8.3 Interfaces, ports et canaux 59
8.4 Impact sur notre travail 63
8.4.1 Mise à jour des signaux 65
8.4.2 Les ports 66

Chapitre 9. Problèmes non résolus et travaux futurs 67

Chapitre 10. Conclusion 69

Bibliographie 72

Annexe I : Création de l'Interface Graphique ix

Fenêtre principale ix
Menus x
Spin box x
Barre de statut xi
Contrôle de la simulation xi
Affichage du temps de simulation et mise à jour des signaux. xiii
Accès à la liste des modules xiii
Tracer les signaux xiv
La classe my_sc_observer xiv
Classe signal_info_base xv
Classe signal_info xvi
Classe signal_window xvii
Changements nécessaires dans le modèle xvii

Remerciements xix

Liste des figures

Figure 1 : Exemple de connexions de ports par des signaux 21
Figure 2 : Exemple de lecture et d'écriture sur un port 23
Figure 3 : Architecture partielle du SystemC 1.1 Beta 28
Figure 4 : Architecture modifiée du SystemC 42
Figure 5 : GUI de CODECSiL-AnsLa 45
Figure 6 : La méthode notify_interface() de la classe sc_simcontext 47
Figure 7 : définition de la méthode notify_interface() de la classe sc_signal 48
Figure 8 : Séquence d'appels pour rendre la valeur du signal accessible au GUI 49
Figure 9 : Trace d'un signal booléen 50
Figure 10 : Temps de simulation selon les cas testés 54
Figure 11: Construction de la classe sc_signal<T> 61
Figure 12 : Architecture du SystemC 2.0 62

Liste des sigles

API	***A**pplication **P**rogrammer's **I**nterface*
EDA	***E**lectronic **D**esign **A**utomation*
GUI	***G**raphical **U**ser **I**nterface*
HDL	***H**ardware **D**escription **L**anguage*
IP	***I**ntellectual **P**roperty*
OSCI	***O**pen **S**ystem**C** **I**nitiative*
OVI	***O**pen **V**erilog **I**nternational*
PLI	***P**rogramming **L**anguage **I**nterface*
RTL	***R**egister **T**ransfer **L**evel*
SDL	***S**ystem **D**escription **L**anguage*
SOC	***S**ystem **O**n **C**hip.*
STL	***S**tandard **T**emplate **L**ibrary*
UML	***U**nified **M**odeling **L**anguage*
VCD	***V**alue **C**hange **D**ump*
VHDL	*VHSIC **H**ardware **D**escription **L**anguage*
VHSIC	***V**ery **H**igh **S**peed **I**ntegrated **C**ircuits*

Chapitre1. Introduction

La compétition féroce entre les fabricants de produits électroniques, tels les téléphones cellulaires, les consoles de jeux, les ordinateurs, décodeurs pour la télévision par satellites et autres produits dont l'évolution est extrêmement rapide, pousse vers des temps de développement très courts et une complexité toujours grandissante. Les méthodologies existantes ne suffisent plus à la demande et de nouvelles méthodologies sont proposées pour répondre à ces besoins.

Une solution proposée est la conception au niveau système [6] (*system level design*) qui implique, entre autre, que la fonctionnalité du système soit modélisée, au tout début du processus de conception, dans un langage pouvant nous fournir un modèle exécutable. La validité du modèle peut donc être testée très tôt lorsque les corrections sont moins coûteuses. Le modèle est ensuite partitionné en composantes matérielles et logicielles. Après plusieurs itérations pour explorer les différents choix d'implémentation, le concepteur garde celle qui lui convient le mieux. Le matériel sera synthétisé vers un circuit intégré alors que la partie logicielle sera exécutée sur un ou plusieurs processeurs. Les langages de description de matériel (HDL) courants comportent des lacunes qui n'en font pas des candidats idéaux pour être utilisés durant toute la durée du processus de conception.

Pour le moment, il semble que la majorité des modèles fonctionnels sont d'abord écrits dans un langage de programmation, surtout C et/ou C++. Lorsque le

modèle est validé, il est ensuite traduit (manuellement ou automatiquement) vers un HDL pour être simulé et synthétisé. Si le modèle comporte des parties en logiciel, la simulation requiert des outils additionnels pour émuler le processeur ciblé par le logiciel.

Plusieurs compagnies ont développé des langages maison ou encore des librairies en C++ pour pouvoir modéliser des systèmes.

En septembre 1999, les compagnies, Synopsys, CoWare et Frontier Design annoncent la distribution gratuite de SystemC [22] qui se veut une librairie, au code source ouvert, en C++ pour la conception au niveau système, le co-design matériel/logiciel ainsi que l'échange de propriété intellectuelle. Cette librairie peut être compilée sur plusieurs plates-formes en utilisant un compilateur standard de C++.

Un modèle écrit en SystemC et compilé produit une application exécutable qui permet la simulation du modèle. Présentement, il n'y a pas d'interface usager graphique accompagnant la librairie de SystemC, nous pouvons générer un fichier de résultats qui ne seront visibles qu'après la fin de la simulation.

Pour suivre les résultats durant la simulation, l'usager doit incorporer des appels à `printf` ou `cout` dans le code du modèle. L'interface est textuelle seulement, alors suivre et interpréter les résultats risque d'être difficile, surtout pour de gros modèles. De plus, si l'usager décide de voir des résultats additionnels, il doit ajouter les instructions nécessaires au modèle et recompiler le code.

Notre objectif est de construire une interface usager graphique pour la simulation de SystemC. Comme SystemC est une librairie C++ au code source

ouvert qui peut être compilé sur plusieurs plates-formes, nous allons utiliser une librairie graphique gratuite, par exemple, QT pour bâtir cette interface. Ceci va raccourcir le temps de conception et permettre la réutilisation. Nous voulons que notre interface puisse s'intégrer à la simulation sans que l'usager ait à apprendre et utiliser de nouvelle syntaxe. Par exemple, nous pourrions créer une nouvelle classe de signaux qui ferait appel directement à l'interface usager mais ceci aurait placé le fardeau du travail vers l'usager.

Nous recherchons une solution qui ne ralentira pas la simulation de façon trop marquée mais qui apportera une compréhension accrue des résultats de simulation. Comme SystemC évolue rapidement, nous aurions intérêt à ce que le lien entre SystemC et notre interface usager graphique ne soit pas très étroit pour qu'ils puissent évoluer indépendamment l'un de l'autre tout en gardant intact leurs moyens de communications.

Ce projet est un travail de débroussaillement qui permet de voir la possibilité d'étendre le SystemC, sans trop changer le code source existant afin de d'offrir une simulation interactive et par le fait même, explorer les possibilités et limitation de la licence de type *Open Source* ainsi que l'évolutivité indépendante de SystemC et des outils connexes. Puis, avec des résultats préliminaires à l'appui, nous pourrions proposer des changements au comité de l'***O**pen **S**ystem**C** **I**nitiative* (OSCI) [22], leur exposer les problèmes qui demandent des modifications plus majeures et, s'ils démontrent un intérêt marqué, s'attaquer à ces problèmes.

Toute autre approche implique que nos changements ne seront jamais inclus dans la version officielle de SystemC et donc, à chaque nouvelle version disponible

du code source, nous devrions refaire plusieurs modifications pour que notre interface fonctionne.

Selon les termes de la licence, nous pouvons rendre public les modèles créés en SystemC, mais pas une version modifiée de SystemC, OSCI se réserve tout droit d'accepter ou non, les modifications proposées. Ainsi, à moins que nos modifications soient acceptées par ce comité, nous n'aurions d'autre initiative que de documenter exhaustivement, tous les changements au code source nécessaires et rendre public cette documentation, sans le code du SystemC, pour permettre à d'autres de créer un GUI selon notre méthode.

Dans le chapitres 2, nous allons discuter des tendances pour les langages de description de matériels (HDL). Le troisième chapitre aborde la problématique alors que les deux chapitres suivants abordent en profondeur la librairie de SystemC version 1.1Beta et la librairie graphique QT.

Les chapitres 6 et 7 exposent la méthodologie que nous avons utilisée, notre expérimentation et les résultats, alors que le chapitre 8 traite de la version de SystemC 2.0, les nouveautés et l'impact qu'elle aura sur notre travail. Le chapitre 9 discute des problèmes non résolus et des travaux futurs alors que le chapitre 10 conclut ce mémoire.

Le travail relié à ce mémoire a fait objet, d'une publication [20] et une généralisation de ce travail fut abordée dans un autre article [21].

Chapitre 2. Langages de description de matériels

La spécification de matériel se classe naturellement en 4 niveaux d'abstraction [8] : algorithmique, modulaire, précis au niveau cycle (*cycle accurate*), et RTL.

Le niveau algorithmique ne spécifie que le comportement d 'un modèle, sans détail spécifique à l'implémentation. Au niveau modulaire, le modèle est partitionné en composantes qui communiquent à travers un protocole clairement énoncé. Ce partitionnement implique la notion d'objet ainsi qu'un mécanisme de communication comme, par exemple, les ports des HDL.

Un modèle précis au niveau cycle contient la notion d'horloge ainsi que la possibilité de créer une cédule des événements réagissant aux changements de la valeur de l'horloge. Le niveau RTL spécifie l'implémentation des événements mais sans spécifier l'implémentation technologique. Ce niveau implique la notion de machine à états.

La conception au niveau système implique une description des besoins et fonctions indépendamment de l'implémentation, donc au niveau algorithmique, la possibilité d'explorer des choix entre matériel et logiciel, de décrire des interfaces et des protocoles sans avoir à décrire explicitement le matériel. Tous les besoins doivent être décrits dans un seul modèle cohérent. Les concepteurs veulent aborder un modèle comme une collection d'objets, donc une syntaxe et sémantique orientée objets serait souhaitable [10]. En ayant un modèle complet à un haut niveau

d'abstraction il est possible de le valider et d'explorer les différentes options avant de raffiner le modèle vers un plus bas niveau d'abstraction. Le raffinement se termine au niveau ou les outils sont capables d'offrir des résultats satisfaisant.

Verilog et VHDL sont les langages de descriptions de matériel les plus utilisés dans l'industrie en ce moment. Ceux-ci ont des lacunes, ils ne sont pas bien adaptés pour représenter un circuit complexe à un très haut niveau d'abstraction. L'industrie doit donc étendre ces langages ou en créer des nouveaux. Pour le moment, la partie matérielle est surtout décrite au niveau RTL en Verilog ou VHDL de façon à avoir accès aux outils de synthèse qui existent pour ces langages, donc, à court ou même moyen terme, tout nouveau langage devrait pouvoir être utilisé en complément à ces deux langages[10]. Les concepteurs d'outils EDA offrent déjà des outils pour traduire une description à un plus haut niveau d'abstraction vers un format de HDL qui est synthétisable.

La question à se poser est quel est le langage le plus approprié pour la prochaine méthodologie? Nous devons tenir compte que le changement anticipé n'est pas uniquement une façon plus compacte et plus puissante de décrire du matériel, c'est plutôt une approche de co-design matériel/logiciel [9]. Bien que nous ne puissions répondre à cette question avec certitude, nous pouvons passer en revue quelques-uns des langages en lutte.

Une solution évidente est d'étendre les HDL courants, Verilog et VHDL. Une autre catégorie de candidats sont les langages de programmation de haut niveau tels C, C++, Java ou des extensions de ceux-ci. Une troisième catégorie sont des

langages de spécifications tels SDL et UML, quoique dans leurs cas, nous aurons à les étendrent vers le bas, vers le monde physique [9].

2.1 Verilog

Verilog est un langage pour modéliser au niveau comportemental et RTL ou à de plus bas niveaux d'abstraction. Malheureusement, la sémantique de Verilog est ambiguë lors de certaines situations à l'exécution et donc, deux simulateurs pourraient avoir des résultats différents à partir d'une même description. De plus, Verilog n'est ni fortement typé, ni orienté objet. Un de ses attraits est la flexibilité de sa syntaxe, mais l'absence d'approche orientée objet fait qu'il est mal adapté à la conception au niveau système.

IEEE et OVI travaillent à améliorer le langage, les changements visent à corriger des erreurs et manques dans le standard courant, rehausser le modèle de *timing* du langage et améliorer la capacité de description comportementale de modèles. Les deux premiers changements devraient apporter une amélioration immédiate du langage. Le troisième changement proposé n'est pas assez puissant pour permettre l'utilisation exclusive du Verilog pour la conception d'un système sur puce (SOC), cette proposition va introduire des changements fondamentaux au langage, augmentant ainsi sa complexité [10].

Verilog offre toutefois la possibilité de lier un modèle avec du logiciel écrit en C/C++ à travers une interface de langage de programmation (PLI). Cette particularité du langage a été rehaussée et pourrait bien jouer en faveur de celui-ci malgré son manque de possibilité d'approche orientée objet.

2.2 VHDL

Par rapport à Verilog, VHDL est plus rigoureux et plus difficile à apprendre, par contre, sa syntaxe et sa sémantique sont plus rigides [10]. Sa capacité de modélisation couvre du niveau comportemental jusqu'au niveau transistor. Par contre, seul un sous-ensemble des constructions du langage est synthétisable. VHDL est aussi en cours de re-standardisation, mais les changements seront minimes à l'exception de l'ajout de variables partagées qui sont un mécanisme amélioré pour modéliser la mémoire.

VHDL a aussi une façon de se lier à des programmes en C/C++, appelée *foreign interface*. Étrangement, la plupart des fournisseurs d'outils EDA ont ignoré cette caractéristique du langage dans leurs produits.

Le VHDL n'est pas un langage orienté objet, un groupe de travail de IEEE a le mandat de le transformer en langage orienté objet, il semble que ce groupe fait beaucoup de progrès.

2.3 C/C++

Puisque les systèmes conçus contiennent, de plus en plus, des blocs de matériel et d'autres de logiciel, une approche utilisant un langage de programmation tel C/C++ semble intéressante [10][11][12]. Par contre, C/C++, à la base, n'offre pas les mécanismes nécessaires pour la description de matériel, il manque les concepts de concurrence, signaux, réactivité ainsi que des types de données reflétant la réalité du matériel.

Il y a deux approches pour ajouter ces concepts au langage. La première est d'étendre la syntaxe en ajoutant des mots clés au langage standard [8][11]. Cette approche implique le développement de nouveaux compilateurs, simulateurs et outils de synthèse pour manipuler la nouvelle syntaxe. C'est l'approche utilisée par SpecC [8] dont nous discuterons plus loin. L'autre approche utilise des librairies de classes pour modéliser le matériel, cette approche nécessite un langage orienté objet, dans ce cas-ci le C++. La syntaxe du langage ne change pas, les concepts pour le matériel sont implémentés dans de nouvelles classes de C++. Nous pouvons donc utiliser les compilateurs et outils de productions existants pour bâtir la simulation. Cette approche à été utilisée par SystemC ainsi que CynApps (maintenant Forte Design Systems) pour Cynlib.

2.3.1 SpecC

SpecC fournit une méthodologie de co-design pour la spécification, la modélisation et la conception de systèmes embarqués et ce, au niveau système. SpecC est une extension du langage C pouvant supporter les hiérarchies structurelle et comportementale, la concurrence, les transitions d'états, le *timing* et le traitement d'exceptions [10]. Le consortium qui s'occupe de SpecC a entrepris des discussions avec l'OSCI (SystemC) sur la possibilité d'une coopération [10]. Depuis, plusieurs concepts de SpecC, qui faisaient défaut à SystemC pour la conception au niveau système, se retrouvent dans la spécification de SystemC version 2.0. Ces concepts sont les interfaces, les canaux et les événements.

SpecC étant une extension du langage C, il est nécessaire de se procurer leur compilateur pour pouvoir compiler et simuler les modèles écrit en ce langage. Le

compilateur, le simulateur ainsi qu'une suite de tests sont disponibles à partir de leur site web [23] et ce gratuitement.

2.3.2 Cynlib

Cynlib est une collection de classes en C++ facilitant la modélisation de la concurrence présente dans le matériel ainsi que des types de données représentant des champs de bits de tailles arbitraires [10]. Cynlib est disponible de Forte Design Systems gratuitement à travers un programme de licence *open-source* sur leur site web[24].

Cynlib contient un simulateur pour un les modèles précis au niveau cycle. CynApps offre aussi un synthétiseur, Cynthesizer [12], qui transforme le code, raffiné au niveau RTL des modèles Cynlib vers du RTL synthétisable de Verilog ou VHDL, au choix [9][10].

2.3.3 SystemC

SystemC est un effort de standardisation pour la modélisation de système utilisant les langages C/C++ qui a été initié par plusieurs compagnies active dans l'industrie : Synopsys, Frontier Design et Co-Ware entre autres. Tout comme Cynlib, c'est une collection de classes en C++, disponible gratuitement selon une licence de type *open source*, permettant, à l'origine, de modéliser du matériel au niveau *cycle accurate*. Bien que le code source soit écrit en C++ en utilisant une approche orientée objet, la clientèle cible de SystemC est l'ingénieur matériel possédant une expertise en C (non orienté objet) et en VHDL. Le changement vers l'approche orientée objet serait donc un gros pas pour les concepteurs de matériel

qui devrait être entrepris graduellement [9]. Plusieurs versions de SystemC ont paru depuis, ajoutant le niveau RTL, et même, il y a quelques semaines, la version 2.0 beta, qui devrait permettre la conception au niveau système. Cette version a bénéficié de la collaboration de SpecC pour incorporer certains concepts, interfaces, canaux et événements présents dans SpecC et non dans les versions précédentes de SystemC. Ces concepts améliorent, entre autres, l'abstraction entre communication et implémentation et permettent maintenant à l'OSCI [22] de solliciter des contributions des concepteurs de systèmes, pour des extensions à la librairie SystemC. De plus, les récents changements en matière d'abstraction des communications facilitent le développement et l'intégration de blocs de propriété intellectuelle (IP) ce qui contribue à diminuer le temps de production et de réutiliser des solutions dont la fiabilité est établie.

Malgré plusieurs nouveautés à la version 2.0 beta facilitant l'utilisation de l'approche orientée objet pour la conception de modèles, la méthodologie expliquée dans le guide d'utilisateur est similaire à la version précédente de SystemC et ne donne aucune indication sur la façon d'utiliser certaines de ces nouveautés.

SystemC permet aussi la définition de procédure de tests homogène applicable à toutes les phases de la conception, les tests sont générés à un niveau d'abstraction quelconque et sont ensuite utilisés pour valider le raffinement vers un niveau d'abstraction plus bas [13].

La compagnie CODECSiL-AnsLab a développé une interface usager graphique pour la version de SystemC sous Windows [18], nous allons comparer leur méthodologie à la notre plus loin dans ce mémoire.

Dans ce mémoire, nous consacrons un chapitre pour une description détaillée du SystemC 1.0 ainsi qu'un chapitre sur les spécifications formelles du SystemC 2.0 ainsi qu'une discussion sommaire sur la version 2.0 beta dont le code source a été rendu publique très récemment.

SystemC semble être la solution en C++ la plus médiatisée en ce moment et comme l'appui de l'industrie est sans cesse grandissant, il nous est permis de croire que SystemC pourra bénéficier des caractéristiques intéressantes des autres approches en C++ et même d'autres méthodologies utilisant d'autres langages. Ceci pourrait contribuer à imposer SystemC comme standard pour la modélisation, en C++, de systèmes sur puces.

2.4 Superlog

Nous pouvons voir Superlog comme un nouveau langage ou comme une extension de Verilog. Superlog fournit un langage pour tous les niveaux d'abstractions nécessaires pour la conception de systèmes. Ce langage est un mélange de C++, Verilog et VHDL tout en conservant une syntaxe semblable à Verilog [10]. Il supporte les hiérarchies, événements, le *timing* et la concurrence comme VHDL et Verilog. De VHDL et C++ il emprunte les concepts de récursivité, tableaux et pointeurs et du C++ il prend la possibilité de supporter des processus dynamiques, de communications, les interfaces, les machines à états et les files. Superlog utilise un modèle qui réagit aux événements et utilise la mémoire partagée comme mécanisme de communication des données. Puisque Superlog englobe Verilog, il peut s'avérer plus facile à apprendre pour un concepteur utilisant déjà Verilog; en effet, nous pouvons toujours écrire du Verilog tout en ayant la

possibilité d'élever le niveau d'abstraction[12]. Co-Design [25] qui a conçu le langage planifie placer Superlog dans le domaine publique

2.5 UML

UML est une collection de notations pour capturer la spécification d'un système logiciel [14][15]. L'industrie tente de déterminer s'il est possible de représenter du matériel avec UML. La compagnie Cadence considère UML comme une source possible pour son système de co-design matériel/logiciel VCC. Ils pensent que le Embedded UML Real Time pourrait permettre de modéliser les fonctionnalités avant de prendre des décisions sur le partitionnement [14][15].

La compagnie de logiciels Project Technology a développé un prototype de compilateur UML à VHDL [15][16], ce produit n'a pas encore été mis en marché.

Le débat actuel est : pouvons-nous utiliser UML tel quel ou devons-nous envisager une extension du standard pour pouvoir modéliser du matériel. Plusieurs des diagrammes de UML peuvent représenter des vues intéressantes d'un système en matériel ; toutefois, UML semble avoir des faiblesses pour pouvoir exprimer la concurrence, les diagrammes d'états ne peuvent représenter qu'une forme très spécialisée de concurrence, pas assez générale pour représenter un système sur puce. Bien que certains pensent que la modélisation est déjà possible avec la version courante du langage, tous sont d'accord que certains changements seraient bénéfiques pour la modélisation de matériel, dont l'ajout de diagramme de *timing* et un modèle réactif représentant les effets d'un événement [14][15].

Chapitre 3. Problématique

Lors de la simulation d'un circuit en SystemC, la version officielle permet de créer un diagramme représentant les variations de la valeur d'un signal dans le temps. Celui-ci est inscrit dans un fichier qui doit être visualisé qu'après la fin de la simulation. De plus, ce fichier doit être lu par un logiciel spécialisé qui n'est pas fourni avec le SystemC, par exemple, Waveform Viewer de Synopsys.

Le choix des variables et signaux à tracer se fait avant la compilation et s'applique à toute la durée de la simulation. Pour un modèle d'un système de grande taille, demandant de vérifier beaucoup de valeurs, la vitesse de simulation chute drastiquement et de plus, on court le risque d'obtenir un énorme fichier qui pourrait créer des problèmes d'espace disque.

Pour avoir accès à des résultats durant le cours de la simulation, on doit insérer, dans le code du modèle, des appels aux fonctions standard du C++ telles `printf` et `cout` pour afficher une valeur dans une fenêtre de type console, donc texte seulement. Ces appels sont donc compilés avec le modèle, ce qui implique que, si l'ont veut afficher les valeurs que va prendre un signal que nous n'avions pas tracé auparavant, nous devons arrêter la simulation, éditer le code du modèle, recompiler celui-ci et enfin, redémarrer la simulation.

De plus, l'affichage à la console sera séquentiel. Comme, à chaque cycle de la simulation, plusieurs processus concurrents sont simulés, retrouver les valeurs

d'un même signal réparties sur plusieurs cycles devient plutôt ardu. En effet, si nous avons besoin de suivre la trace d'un grand nombre de signaux, il ne faut que quelques cycles pour que les premières valeurs disparaissent de la zone visible de la fenêtre.

En utilisant la méthodologie suggérée de SystemC, le nombre de cycles d'horloge que durera la simulation est fixé à la compilation, et l'exécution se fait sans interruption pour cette durée.

Bien sûr, nous pouvons toujours intégrer dans le modèle des instructions qui nécessitent l'intervention de l'usager avec un message du style « appuyer sur une touche », leurs positions dans le code du modèle, donc le moment de leurs occurrences à l'exécution, seront encore une fois fixées à la compilation.

Une alternative serait l'utilisation d'un débogueur, mais ceux-ci changent selon la plate-forme sur laquelle nous travaillons. Aussi, seules les valeurs courantes globales, ou encore locales aux processus courant, sont disponibles.

Comme le code source de SystemC est disponible, il est certain que nous pouvons y ajouter une interface usager graphique, cependant, comme le SystemC évolue très rapidement, nous devons nous efforcer de créer une interface graphique qui serait réutilisable malgré les changements de versions. Le SystemC, tel qu'il est présentement ne s'y prête guère, nous allons donc proposer quelques changements pour améliorer cela.

Chapitre 4. La librairie SystemC

SystemC n'est pas un nouveau langage, ni même une extension du langage C++, il n'ajoute aucune nouvelle syntaxe au C++. C'est plutôt une librairie de classes, écrites en C++. Donc pour quelqu'un qui a des notions de C++, modéliser du matériel n'est qu'une question d'apprendre à utiliser ces nouvelles classes. Nous pouvons aussi utiliser pratiquement n'importe quel outil de développement adapté au C++.

SystemC permet de créer un modèle, précis au niveau cycles (*cycle accurate*), d'une architecture matérielle, de l'interface pour un système sur puces (*System On Chip*) et de conceptions au niveau système.

Cette nouvelle librairie permet aux concepteurs de circuits de créer des modules, processus, d'ajouter de la communication, entre processus et entre modules, à travers des ports et des signaux pouvant s'adapter à des données d'une multitude de types. Elle introduit aussi le « *timing* », la concurrence et les comportements réactifs nécessaires pour modéliser une architecture système.

Pour utiliser SystemC, nous devons obligatoirement connaître le langage C, par contre, ça ne demande qu'une connaissance minime de C++ et aucune de l'approche Orientée Objet.

La modélisation elle-même, du moins au niveau RTL, suit une sémantique semblable à VHDL ou Verilog, ainsi, les concepteurs de systèmes devraient pouvoir

rapidement apprendre à utiliser SystemC. Par contre, à des niveaux d'abstraction plus élevés, la modélisation avec SystemC peut être très différente de celle de VHDL ou Verilog. À ces niveaux, le SystemC a l'avantage d'être plus accessible à quiconque connaît le langage C.

Voici quelques-uns des avantages de suivre la méthodologie proposée par SystemC :

- La conception crée un modèle exécutable au tout début des étapes de raffinement du modèle permettant de tester tout de suite la validité des fonctionnalités du système.
- La modélisation du système n'utilise qu'un seul langage et n'a donc pas besoin d'être traduite dans un autre langage de description de matériel (HDL) ce qui peut introduire des incohérences entres les deux modèles.
- Le système peut être modélisé du niveau comportemental et raffiné itérativement jusqu'au niveau RTL.
- Les mêmes tests peuvent être utilisés à tous les niveaux de raffinement de la conception du système pour s'assurer qu'il n'y a pas d'inconsistances entre deux itérations.
- SystemC permet le co-design logiciel/matériel. Il est possible de créer une description d'une architecture système complexe contenant des composantes matérielles et d'autres, logicielles. Le matériel et la communication sont modélisés en utilisant la sémantique issue de la

librairie de SystemC alors que le logiciel est libre d'utiliser toute la puissance de l'approche orientée-objet du C++.

- SystemC supporte un grand nombre de types de données, de ports et de signaux.

Malheureusement, le guide de l'usager qui accompagne le code source du SystemC, escamote toute notion d'orienté objet, celui-ci nous présente une recette utilisant des `struct` et non `class`, nous devons utiliser une macro SC_CTOR qui appelle le constructeur approprié tout en nous empêchant d'avoir recours à l'héritage.

Quelques classes ont bien plusieurs constructeurs, mais un seul est documenté dans les manuels de l'usager des différentes versions de SystemC.

La librairie de SystemC ajoute les concepts suivants aux C++ :

Modules : Cette classe permet de modéliser la hiérarchie du système. Cet objet n'est qu'un contenant pour d'autres modules ou des processus et n'a aucune fonctionnalité. Les modules contiennent aussi des ports, constructeurs, des données et des fonctions membres.

Processus : Ils sont utilisés pour décrire la fonctionnalité, ils sont contenus dans des modules. Ceux-ci s'exécutent en parallèle. Il y a trois types de processus. Le constructeur de ce type d'objet est escamoté derrière des macros comme `SC_METHOD`.

Ports : Utilisés pour la communication entre les modules, ceux-ci peuvent être des ports d'entrées, de sorties ou encore entrées/sorties.

Signaux : Les signaux peuvent servir à communiquer entre les modules s'ils sont liés à un port. Ils permettent aussi la communication entre différents processus contenus dans un même module.

Horloges : SystemC ajoute un type spécial de signaux pour veiller à la cadence de la simulation. Il est possible d'avoir plusieurs horloges, une d'elles sera l'horloge principale, les autres auront une phase arbitraire dérivée de la principale.

Protocoles de communications : SystemC a une sémantique de communication à plusieurs niveaux pour différents niveaux d'abstraction.

Dans les sections suivantes, nous allons élaborer sur les concepts comme les modules, les ports, les signaux ainsi que leurs types de données et des processus.

4.1 Modules

Les modules sont la base pour partitionner la conception d'un système. Les gros modèles seront habituellement divisés en plusieurs modules pour représenter des parties distinctes du système. Chaque module peut alors être développé par un groupe différent. Comme un module permet d'encapsuler les données et le traitement, l'important est de bien définir son interface avec l'extérieur et sa fonctionnalité qui sera implantée grâce aux processus qu'il contient.

SystemC nous propose une macro `SC_MODULE` qui sert de constructeur pour ce type d'objet. Cette dernière initialise aussi les variables locales au module. Un module peut contenir plusieurs autres éléments tels des ports, signaux locaux, données, d'autres modules et des processus.

Les données peuvent être conservées dans des variables locales au module qui ne sont donc pas visible de l'extérieur.

4.2 Types de données

SystemC permet comme types pour les données, les signaux et les ports, d'abord, tous les types standard du C++, ainsi que des types, uniques à SystemC. Ces types sont :

- sc_bit : type comprenant un bit à deux valeurs possibles.
- sc_logic : un bit à quatre valeurs possibles, soient 0 (faux), 1 (vrai), X(inconnu) ou Z(haute impédance).
- sc_int : entier signé de un à 64 bits (déterminé par `template`).
- sc_uint : entier non-signé de un à 64 bits.
- sc_bigint : un entier signé de taille arbitraire.
- sc_biguint : entier non-signé de taille arbitraire.
- sc_bv : un vecteur de bits à deux valeurs de taille arbitraire.
- sc_lv : un vecteur de bits à quatre valeurs de taille arbitraire.
- sc_fix : valeur point fixe signée.
- sc_ufix : valeur point fixe non-signée.
- sc_fixed : valeur point fixe signée dont la longueur, précision et autres paramètres sont déterminés par un `template`,
- sc_unfix : valeur point fixe non-signée utilisant un `template`.

De plus, l'usager à la possibilité de définir ses propres types en utilisant des structures (`struct`) définies à partir des types hauts et/ou les types de bases du C++.

```
SC_MODULE(stage_IF)
{
  // Déclarations des ports
  sc_inout<int>            PC; // Program counter
  sc_out<sc_uint<32> >     IR; // Instruction register

// ... le reste des déclarations de la classe
}

SC_MODULE(stage_ID)
{
  sc_inout<sc_uint<32> > IF_ID_IR; // Inst. reg. vient du IF
  sc_inout<int>   PC;              // Program counter (du IF)

// ... le reste des déclarations de la classe
}

int sc_main(int ac, char *av[])
{
 // déclarations des signaux
 sc_signal<int> PC("PC");
 sc_signal<sc_uint<32> > IF_ID_IR("IF_ID_IR");

 // ... suite des déclarations

 stage_IF S1("stage_IF");    //declaration du module S1
 S1.PC(PC);        // signal PC est lié avec le port PC de IF
 S1.IR(IF_ID_IR); // signal IR est lié avec le port IF_ID_IR de IR

 // ... suite des liens des autres ports de IF

 stage_ID S2("stage_ID");  //declaration du module S2
 S2.PC(PC);                // signal PC est aussi lié avec le port PC de ID
 S2.IF_ID_IR(IF_ID_IR);// signal IF_ID_IR est lié avec le port IF_ID_IR de ID

 // ... suite des liens des autres ports de ID
}
```

Figure 1 : Exemple de connexions de ports par des signaux

4.3 Ports et signaux

Les ports servent à interfacer les modules. Il y a trois types de ports : `in` (entrée), `out` (sortie) ou `inout` (entrée/sortie). Nous devons aussi, spécifier avec un gabarit (*template*), le type de données qui transiteront par le port. Ce type peut être n'importe lequel parmi ceux énumérés précédemment.

Chaque port est lié à un signal, le type de données qui seront transmis par ce signal doit être le même que celui spécifié pour le port. Le lien entre le port et le signal sera créé lors de l'instanciation du module comme nous pouvons observer à la Figure 1.

Un signal connecte le port d'un module avec le port d'un autre module. Le signal va transporter la valeur d'un port vers l'autre. Le port lui, fournit la direction du transfert. Les signaux correspondent donc, au niveau physique, aux fils qui interconnectent les composantes.

Lorsque la méthode `read()` d'un port est appelée, la valeur du signal auquel il est lié est retournée, alors que, lorsque la méthode `write()` est invoquée, la valeur du signal sera mise à jour lorsque le processus, qui invoque la méthode, aura terminé son exécution ou sera suspendu. La Figure 2 nous montre un exemple de lecture et écriture sur un port.

Les signaux peuvent avoir un ou plusieurs lecteurs et un seul émetteur sauf dans le cas de signaux résolus qui peuvent en avoir plusieurs. Notez que ces derniers s'appliquent uniquement aux « Tri-State Bus ».

```
// un processus du module IF
void stage_IF:: process_IF()
{
// Début du code ...
  IR.write(ins);       // on écrit le contenu
  PC.write(addr+4);   // des variables sur le port
}

// un processus du module ID
void stage_ID:: process_ID()
{
  IR = IF_ID_IR.read(); // on lit la valeur sur le port et
  pc = PC.read();        //  on la place dans une variable
// ... reste du code
}
```

Figure 2 : Exemple de lecture et d'écriture sur un port

Un cas spécial de signaux est l'horloge (`sc_clock`), ces signaux génèrent la cadence et sont utilisés pour synchroniser les évènements lors de la simulation.

4.4 Processus

Les processus implémentent les fonctionnalités des modules, ceux-ci permettent une émulation du comportement du système matériel qui effectue plusieurs opérations en parallèle.

Certains processus se comportent comme des fonctions, l'exécution débute lors de l'appel, les instructions du processus s'exécutent séquentiellement jusqu'à la fin. Les autres sont appelés au début de la simulation et contiennent une boucle infinie, ils s'exécutent jusqu'à une instruction `wait` qui suspend l'exécution active du processus jusqu'à l'occurrence d'un évènement. Ce type de processus a une liste de sensibilité, un changement de valeur d'un des signaux de cette liste va provoquer le retour à l'exécution active du processus.

Lorsque la valeur d'un signal change, le noyau de simulation de SystemC identifie les processus qui y sont sensibles et démarre leurs exécutions. Les expressions contenues dans le processus s'exécutent alors séquentiellement jusqu'à la fin du processus ou à un appel de la fonction `wait()`.

Le SystemC permet trois types de processus :

Method Process : Ce type de processus est activé par un changement de valeur d'un signal auquel il est sensible, une méthode s'exécute en entier et retourne le contrôle au noyau de simulation. Ce type de processus ne peut être suspendu et ne doit pas contenir une boucle infinie.

Thread Process : Ce type de processus peut être suspendu et réactivé, celui-ci contient un appel à la fonction `wait()` qui suspend l'exécution jusqu'à un événement sur un signal de sa liste de sensibilité. Dans ce cas, l'exécution reprend la où elle était et continue jusqu'au prochain `wait()`.

Clock Thread Process : Ce type de processus ne peut être activé que sur un seul front d'une horloge, c'est donc un processus synchrone. Le processus contient une boucle infinie qui s'exécute jusqu'à une instruction `wait_until()`. Chaque fois que le front d'horloge auquel le processus est sensible est actif, la condition du `wait_until()` est évaluée, et lorsqu'elle devient vraie, l'exécution reprend.

La méthode `wait_until()` ne fonctionne qu'avec des expressions sur le type signal booléen (sc_signal<bool>), de plus nous devons utiliser la méthode `delayed()` sur le signal pour obtenir la bonne valeur.

Les deux derniers types de processus ci-haut utilisent des boucles infinies qui s'exécutent continuellement; SystemC fournit une façon, le *watching*, pour sortir de la boucle ou réinitialiser le processus lors de l'occurrence de certains événements. Ceci est utile, par exemple, si nous avons besoin de modéliser le `reset`.

Pour le *watching*, nous devons spécifier la ou les conditions à examiner dans le constructeur du processus. Lorsqu'une de ces conditions devient vraie, l'exécution s'interrompt et reprend au début du processus, le code qui gère cette interruption doit donc être placé au tout début du processus, avant la boucle infinie. S'il y a plusieurs conditions à examiner, nous devons aussi tester quelle condition a été observée. Les expressions examinées sont testées à chaque `wait( )` ou `wait_until( )` de la boucle. Le *watching* est actif durant toute la simulation, sur l'ensemble du processus.

Pour les processus synchrones, il est possible d'utiliser une autre forme de *watching* qui ne va s'appliquer qu'à une région du processus, ceci nous permet d'observer des conditions sur différents signaux à des endroits différents du code du processus, cette forme s'appelle le *local watching*. Cette forme est moins prioritaire que la forme globale. Les conditions seront examinées à chaque front de l'horloge à laquelle le processus est sensible.

4.4.1 sc_main

Fonction principale pour le modèle, c'est elle qui crée les instances de tous les modules et qui génère l'horloge principale. Cette fonction, tout comme la

fonction `main` en C, peut recevoir des arguments en ligne à l'appel du binaire exécutable.

4.5 Architecture du SystemC

Dans la section précédente nous avons abordé la sémantique du SystemC du point de vue conception de modèle. Pour ajouter aux fonctionnalités du SystemC, nous profitons du fait que le code source est accessible.

Une partie importante du travail que nous avons accomplie, a été de fouiller le code, non documenté, du SystemC. Cette section nous permettra d'explorer le code source du SystemC et de comprendre, un peu plus, le déroulement de la simulation. Par contre, il serait souhaitable qu'il existe déjà des mécanismes pour permettre d'étendre le SystemC sans pour autant changer le code source.

La fonction `sc_start( )` correspond au début de la simulation, celle-ci est définie dans la classe `sc_simcontext`. Si nous regardons son code dans la librairie SystemC, nous voyons qu'elle appelle la méthode `initialize()` et ensuite, selon les paramètres de `sc_start()`, une des méthodes suivantes sera appelée : `simulate()`, `simulate_forever()` ou `sc_cycle()`.

Dans `initialize()`, nous trouvons un appel à `initial_crunch()` alors que les trois autres méthodes ci-dessus ont un appel à la méthode `crunch()`, ces méthodes gèrent tout le traitement qui se fait entre deux changements de valeur de l'horloge, elles contiennent une liste des signaux à mettre à jour à la fin de chaque delta cycle. Ces méthodes représentent donc une bonne partie de l'ordonnanceur de SystemC.

Les méthodes et processus asynchrones sont activés chaque fois qu'un signal auquel ils sont sensibles change de valeur, comme ceci peut ce produire plusieurs fois avant que l'horloge change de front, un delta cycle contient deux étapes, qui sont réputées se produire instantanément par rapport au temps de simulation. Ces étapes sont les suivantes :

- Tous les processus et méthodes asynchrones dont un signal inclus dans leurs listes de sensibilités change de valeur sont activés, ceux-ci sont exécutés en entier ou jusqu'à une instruction `wait()` selon le cas.

- Les signaux qui se sont fait attribuer une nouvelle valeur à l'étape précédente voient cette valeur mise à jour. Si aucun signal n'a changé de valeur le temps de simulation avance vers le front d'horloge suivant. Sinon, nous retournons à l'étape précédente pour le début d'un nouveau delta cycle.

La Figure 3 est une représentation partielle, en UML, de la structure des classes qui forment le SystemC ainsi que leurs relations. Dans ce schéma, un nom en italique signifie que la classe est abstraite ou la méthode est virtuelle. Un membre précédé par "+", "-" ou "#" indique que le membre est public, privé ou protégé. Ce qui suit est une description des différents éléments de la Figure 3 :

4.5.1 sc_object

Cette classe est la classe de base pour beaucoup d'objets de SystemC. Elle contient des méthodes de bases et des propriétés qui nous permettent d'identifier et classifier des objets de SystemC.

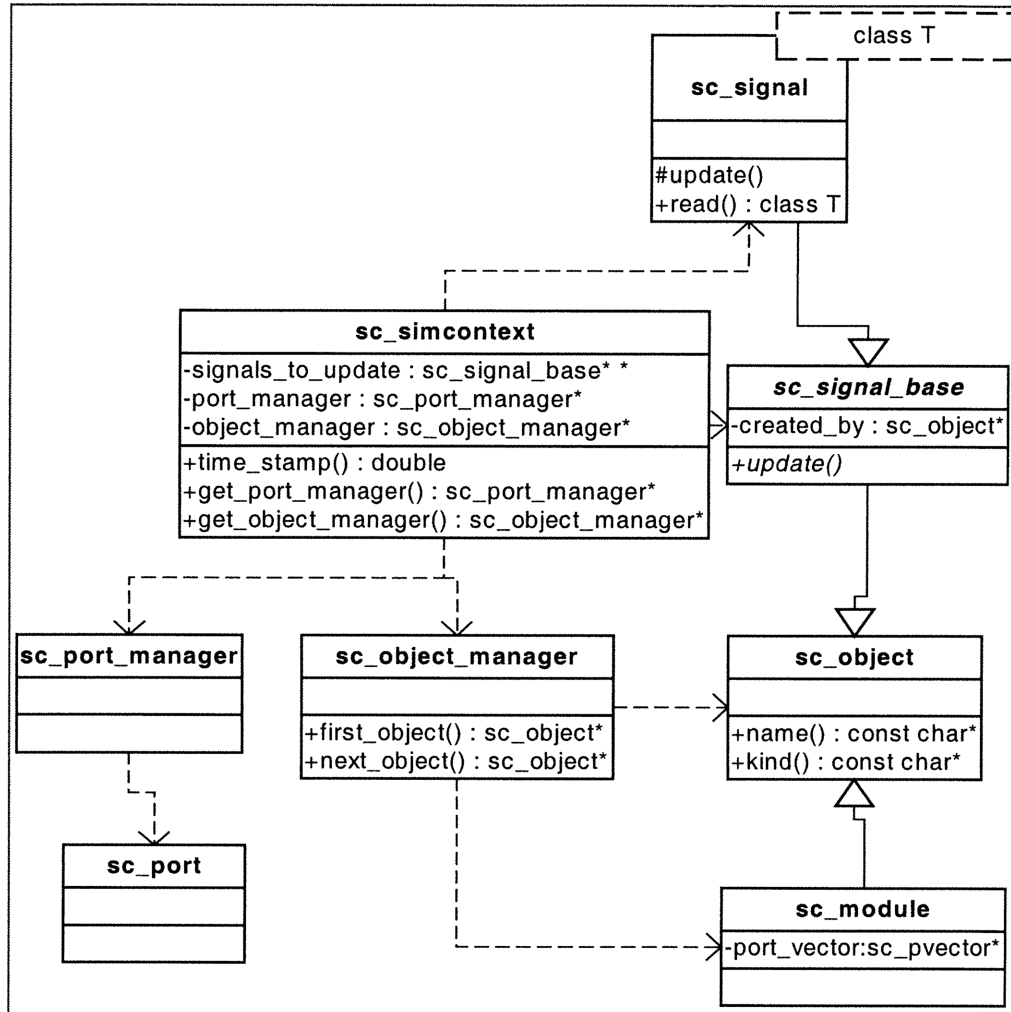


Figure 3 : Architecture partielle du SystemC 1.1 Beta

4.5.2 sc_module

Cette classe permet de modéliser la hiérarchie du système. Cet objet n'est qu'un contenant pour d'autres modules ou des processus.

4.5.3 sc_port

Les objets de cette classe permettent la communication entre les modules. Il y a trois types de ports dans SystemC, ports d'entrée, de sortie et entrée/sortie.

4.5.4 sc_signal_base

La classe de base pour tous les signaux, celle-ci contient des méthodes de bases quel que soit le type d'un signal. À noter, cette classe est abstraite, nous ne pouvons donc pas en créer une instance, elle ne sert que pour la dérivations d'autres classes spécialisées.

4.5.5 sc_signal

La classe dérivée de `sc_signal_base`, celle-ci contient comme information la valeur du signal qui est un gabarit. Cette classe permet donc de créer des instances de signaux de plusieurs types différents. Ces signaux sont non-bloquants, le `read` est immédiat alors que le `write` provoque une mise à jour du signal au prochain cycle.

4.5.6 sc_object_manager

Cette classe gère les différents objets de SystemC (modules, signaux, horloges, etc.), elle contient une liste, privée, de ceux-ci. Dans cette classe, il y a des méthodes publiques, telles `first_object( )` et `next_object( ),` qui permettent d'accéder cette liste.

4.5.7 sc_simcontext

Cette classe contrôle la simulation, celle-ci contient la méthode `crunch()` qui est la méthode qui s'occupe de faire tout le traitement nécessaire pour gérer les delta-cycles et faire avancer la simulation jusqu'au prochain front d'horloge. La

classe contient aussi la liste des signaux à mettre à jour, le temps de simulation ainsi que des méthodes permettant de démarrer, arrêter et avancer la simulation d'un pas.

Maintenant que nous avons vu la librairie de SystemC, le prochain chapitre nous décrit la librairie graphique QT, les composantes de cette librairie pouvant nous permettre d'ajouter une interface graphique à la simulation en SystemC, ainsi que les possibilités offertes pour la réutilisation, l'extensibilité et la souplesse de l'utilisation.

L'implémentation des classes du code source du SystemC, telles que vues sommairement à la Figure 3 est à la base de la méthodologie que nous allons exposer au chapitre 6.

Chapitre 5. La librairie graphique QT

QT est un cadre de développement d'interface usager graphique multi-plates-formes implémenté en tant que librairie en C++. QT est supporté sur Windows 95/98/NT/2000, Linux, Solaris, etc. Sous certaines conditions, une version Unix/X11 de QT est disponible gratuitement pour le développement de logiciels gratuits avec code ouvert.

QT fournit un API (*Application Programmer's Interface*) qui est identique pour toutes les plates-formes. Donc, une application créée sur une plate-forme peut-être portée sur une autre plate-forme, nous avons juste à recompiler l'application et la lier avec la librairie QT de cette plate-forme. Ceci a l'avantage de gagner beaucoup de temps de développement et de maintenance si nous voulons écrire des applications multi-plates-formes.

Dans ce chapitre, nous aborderons certains concepts qui, bien que nous ne les ayons pas tous utilisés, contribuent à rendre notre travail réutilisable et extensible.

QT émule l'apparence (*look and feel*) de la plate-forme sur laquelle l'application est exécutée. Tous les éléments visuels de QT sont implémentés avec une apparence dynamique et cette apparence peut être changée même durant l'exécution. QT peut émuler les styles suivants : Motif (X11), CDE, Windows, SGI, Motif Plus et Platinum (MacOS). Il est aussi possible de personnaliser l'apparence de notre application, dans ce cas tous les éléments visuels vont se conformer à ce nouveau style.

5.1 Communication inter-objets

Pour permettre de structurer le code des applications en composantes indépendantes et réutilisables, QT offre un mécanisme de communication inter-objets qu'ils appellent *signals and slots* que nous traduirons par messages et créneaux pour ne pas confondre avec les signaux qui servent à la communication à l'intérieur du modèle en SystemC. Par ce mécanisme, les objets peuvent émettre anonymement un message qui fera exécuter une fonction d'un créneau d'un autre objet.

Voici les fondements du mécanisme messages et créneaux :

- Toute classe qui définit un message ou un créneau doit hériter de la classe de base `QObject` de QT.
- Une classe dérivée de `QObject` peut définir n'importe laquelle de ses méthodes comme étant un créneau.
- Une classe dérivée de `QObject` peut se définir capable d'émettre des messages. Un message doit avoir un nom et une liste de paramètres.
- Le message d'un `QObject` peut être connecté au créneau d'un autre `QObject`, de plus, si le message et le créneau sont tous les deux déclarés public, la connexion peut même être entreprise par un troisième objet.
- Un QObject peut émettre un message à tout moment.

Lorsqu'un QObject émet un message, la fonction créneau à laquelle il est connecté s'exécutera immédiatement. Les paramètres sont passés par l'objet

émetteur à la fonction créneau. Émettre un message est donc comme appeler une fonction sauf que l'objet qui émet le message ignore quelle fonction créneau de quel(s) objet(s) sera exécutée. Ceci facilite le développement de classes réutilisables et indépendantes de l'application.

Un message peut être connecté à zéro ou plus fonctions créneaux. De même, une fonction créneau peut être connectée à plusieurs messages. Le nombre et le type de paramètres à passer ne sont pas fixés d'avance, c'est vraiment comme déclarer une fonction. Le mécanisme est robuste au niveau du passage des paramètres. Si un message essaie de se connecter à une fonction créneau avec un ou des paramètres dont les types ne concordent pas, la connexion sera ignorée et un avertissement sera émis. Par contre, si le nombre de paramètres du message est plus grand que celui de la fonction créneau, les paramètres superflus seront ignorés.

5.2 Internationalisation

QT permet l'utilisation de n'importe quelle langue et ensemble de caractères. Il est facile de changer de langue et ce, même durant l'exécution. Le programmeur peut utiliser la fonction `tr()` avec une chaîne de caractères comme paramètres avant de l'afficher. Cette fonction va consulter la table de traduction courante et retourner le texte traduit en conséquence, si aucune traduction existe pour la chaîne, la valeur du paramètre sera retournée.

QT offre aussi comme outil `findtr()` qui cherche dans le code source les paramètres de la fonction `tr()` qui n'ont pas de traduction dans la table courante et produit un fichier texte avec ces chaînes et un endroit où le développeur peut écrire la traduction manquante. Ces fichiers peuvent ensuite être utilisés par un autre outil,

msg2qm(), qui créer les tables de traductions utilisées par QT. L'outil `mergetr()` permet de fusionner des fichiers de traduction pour incorporer les traductions manquantes à une table de traduction.

5.3 Widgets

Les éléments de base pour la création d'interface usager graphique avec QT sont les *widgets*. La classe fondamentale pour les *widgets* de QT est `QWidget`, tous les classes de *widgets* héritent directement ou indirectement de cette classe. QT met à notre disposition un grand nombre de classes de *widgets* pour divers objets tels des fenêtres, des boutons, des menus, etc.

Pour plusieurs outils permettant la conception d'interfaces usagers, il y a deux types d'éléments pour une interface, soit les contrôles et les contenants. QT ne fait pas de différence fondamentale entre les deux, en effet n'importe quel *widget* peut être l'un ou l'autre, la distinction va plutôt dépendre d'une relation parent-enfant. Un *widget* qui contient d'autres *widgets* est le parent de ceux-ci. La fonctionnalité de ces différents objets est surtout implémentée en connectant les messages et créneaux de ceux-ci entre eux.

Il est facile de créer des *widgets* personnalisés. En effet, en plus d'hériter de toutes les propriétés et méthodes nécessaires de la classe de base `QWidget`, nous n'avons qu'à implémenter les méthodes virtuelles de cette dernière qui nous sont utiles pour définir le comportement souhaité de notre nouveau *widget*.

En plus des *widgets*, QT offre la classe `QPainter` qui permet de tracer des lignes, polygones, ellipses, splines, images, etc. Cette même classe supporte aussi

les transformations sur ces graphiques comme les rotations et le changement de taille, ainsi que la gestion des couleurs.

Au chapitre suivant, nous allons discuter des avantages que l'utilisation de cette librairie nous a apportés pour construire notre interface usager graphique. Une description détaillée des instructions QT que nous utilisons se trouve à l'annexe 1 de ce mémoire.

Chapitre 6. Méthodologie

Le code de SystemC étant ouvert, nous aurions pu apporter des modifications dans celui-ci sans nous soucier du fait que nous nous éloignons de plus en plus de la version officielle avec chaque changement.

Par exemple, nous avons besoin de connaître tous les signaux d'un module, il est facile de créer une nouvelle classe de modules dérivée de la classe `sc_module` (voir Figure 3 à la page 28) qui contient toutes les méthodes nécessaires pour accéder les signaux internes du module. Par contre, ceci changera la syntaxe à utiliser par le concepteur du design et rendra l'échange de modèle difficile puisque non standard. De plus, les changements seront à refaire pour toute version ultérieure de SystemC. Si nous nous engageons dans cette voie, nous allons multiplier le nombre d'interventions nécessaires sur le code source du SystemC.

6.1 Approche orientée objet et design patterns

Un avantage des méthodologies utilisant C++ est que nous pouvons tirer avantage de l'approche orientée objet. Une plus grande abstraction peut être obtenue en capturant les concepts d'un modèle dans des objets C++ et de les offrir en tant que librairies de classes aux concepteurs de systèmes [11] comme par exemple, les FIFO et autres canaux élémentaires offerts en SystemC 2.0. La clé pour la réutilisation est d'avoir une séparation claire entre la fonctionnalité d'une

composante et sa communication. Cette méthodologie est la conception basée sur l'interface (*interface based design*) où l'interface de la composante peut être raffinée indépendamment du comportement interne de la composante et vice-versa. Le polymorphisme de concert avec la surdéfinition des opérateurs permettent à une méthode ou un opérateur, d'agir différemment selon l'objet pour lequel il est appelé. La structure du programme reste donc pareille [11][17] et c'est la responsabilité de l'objet d'implémenter les différences.

L'héritage permet la réutilisation de la structure d'une (ou plusieurs) classe de base ainsi que l'extension de cette classe. Un programme est considéré extensible s'il peut être adapté à de nouvelles tâches sans modifier son code source [19]. Ceci est possible grâce à l'héritage, en effet, pour ajouter de nouvelles fonctionnalités à une classe, nous pouvons la dériver vers une nouvelle et ajouter la fonctionnalité à cette dernière. Le code de la première classe demeure donc inchangé. L'utilisation d'un gabarit (*template*) pour une classe ou méthode permet de généraliser le comportement de celle-ci pour tout type de données, ceci permet donc de réutiliser une classe générique et laisser le soin au compilateur de générer les objets et méthodes seulement pour les types de données où c'est nécessaire.

En génie logiciel, il existe un catalogue de problèmes récurrents et de solutions types appelés *design patterns* [7], ceux-ci contiennent quatre éléments essentiels :

- Un nom pour l'identifier
- La description du problème, et le contexte où la solution est applicable

- Une solution type
- Les conséquences de cette approche.

Puisque la solution suggérée est habituellement générale et dans le but d'augmenter la réutilisation et la flexibilité de la solution. Les conséquences de la solution suggérée est souvent des compromis au niveau de l'espace mémoire ou de la vitesse d'exécution.

Un exemple est le *design pattern* ***Observer*** qui définit une dépendance « un à plusieurs » entre un objet (sujet) et ses dépendants (observateurs) qui seront avertis lors de la mise à jour de certaines valeurs du sujet. La documentation de *Observer* indique le contexte où il est applicable, entre autres lorsqu'un sujet veut avertir un observateur de changements de valeurs sans se soucier de l'implémentation des observateurs, donc lorsque le sujet et les observateurs ne sont pas liés serrés ensemble. La solution implique une classe `observer` qui est abstraite et dérivée pour chaque observateur, celle-ci contient une déclaration purement virtuelle d'une méthode *update()* qui sera redéfinie dans tous les observateurs, la solution prévoit aussi des mécanismes pour attacher et détacher les observateurs.

Un exemple de la structure des classes composant la solution type est donnée dans la documentation du *pattern*. Voici un exemple de la déclaration de la classe abstraite dont les observateurs vont hériter ainsi qu'un exemple de la classe pour le sujet.

```
class Subject;

class Observer {
public:
      virtual ~Observer();
      virtual void update( Subject * theChangedSubject) = 0;
protected:
      Observer();
};

class Subject {
public:
      virtual ~Subject();

      virtual void Attach(Observer *);
      virtual void Detach(Observer *);
      virtual void Notify();
protected :
      Subject();
private:
      List<Observer*> *_observers;
};
```

Nous voyons que la solution proposée est très générale, réutilisable, facile à étendre. En effet, cette solution permet un nombre indéterminé d'observateurs quelconque, puisque le sujet ne connaît pas le comportement de ces observateurs, une extension qui semble facilement réalisable est de permettre certains observateurs d'être liés à plusieurs sujets.

6.2 Notre implémentation

Pour faciliter la réutilisation de notre application nous nous sommes inspirés de la méthodologie de la conception basée sur l'interface (*interface based design*) pour permettre la communication entre les deux applications, soit la simulation du modèle en SystemC et l'interface usager graphique. Cette méthodologie a l'avantage de faire abstraction des détails de l'implémentation du GUI, donc peut être réutilisée pour une autre application nécessitant les mêmes communications avec la simulation de SystemC.

Nous avons concentré les nouveautés que nous ajoutons dans une nouvelle classe que nous nommons `sc_observer`. Le choix du nom de la classe vient du fait que notre méthodologie semble correspondre à une spécialisation de l'approche proposée par le *design pattern* ***Observer*** [7], d'ailleurs, un projet connexe de généraliser cette interface entre SystemC et des applications quelconques est maintenant en cours.

La classe `sc_observer` peut plus facilement se greffer à n'importe qu'elle version de SystemC, ainsi, en changeant de version, nous n'avons besoin que d'introduire, à divers endroits stratégiques dans le code source du SystemC, quelques instructions pour interagir avec notre nouvelle classe.

Cette classe pourra ensuite contenir toutes les méthodes et membres nécessaires pour servir d'interface avec une autre librairie. Pour notre prototype, l'interface ne passe qu'un seul message, celui-ci indique qu'un signal a été mis à jour et fournira la nouvelle valeur. La librairie qui est liée au SystemC correspond à une interface usager graphique.

Nous essayons de préserver le niveau de protection des données qui transitent dans SystemC, notre classe ne fait qu'accéder les valeurs de certaines données sans possibilité de les changer.

Les changements que nous apportons ne doivent pas empêcher l'utilisation de SystemC sans notre interface graphique.

Nous nous efforçons de ne pas imposer de nouvelle syntaxe pour la modélisation du matériel avec SystemC en utilisant notre interface graphique.

6.3 Description de la classe interface

La classe que nous créons doit permettre l'utilisation du SystemC sans qu'il n'y ait d'instance de cette classe de déclarée, nous allons nous déclarer un membre et quelques méthodes statiques qui sont illustrés dans la Figure 4 :

- `bound_instance` qui est un pointeur sur une instance de cette classe (ou d'une classe dérivée) qui est initialiser à NULL pour indiquer, qu'à l'origine, il n'y a pas d'instance de créée.
- La méthode booléenne `is_bound()` qui retourne faux si `bound_instance` est NULL, vrai sinon.
- La méthode `bind()` qui nous servira à changer la valeur de `bound_instance` lorsqu'il y aura une instance de cette classe ou d'une classe dérivée.
- La méthode `get_bound_instance()` qui retournera un pointeur sur l'instance en question.

Pour le reste de la définition de la classe, nous allons déclarer toutes les méthodes qui nous seront utiles pour notre interface graphique. Ces déclarations seront pure virtuelles de façon à obliger la redéfinition de ces méthodes dans une classe dérivée (polymorphisme), mais aussi, ne permettra pas d'instance de cette classe, mais uniquement de classes dérivées (abstraction).

Une fois ces méthodes déclarées, nous pouvons ensuite placer des appels à celles-ci dans le code du SystemC et celui-ci va compiler. Si de plus, avant chaque appel, nous vérifions, grâce à la méthode `is_bound` s'il y a une instance d'une

classe dérivée, alors nous sommes certains qu'aucune méthode pure virtuelle de notre classe interface ne sera appelée sans avoir été redéfinie. Ceci implique que nous pouvons donc utiliser le SystemC même sans créer une interface, le coût en temps ne sera alors que de quelques tests du genre : `if(sc_observer::isbound())`.

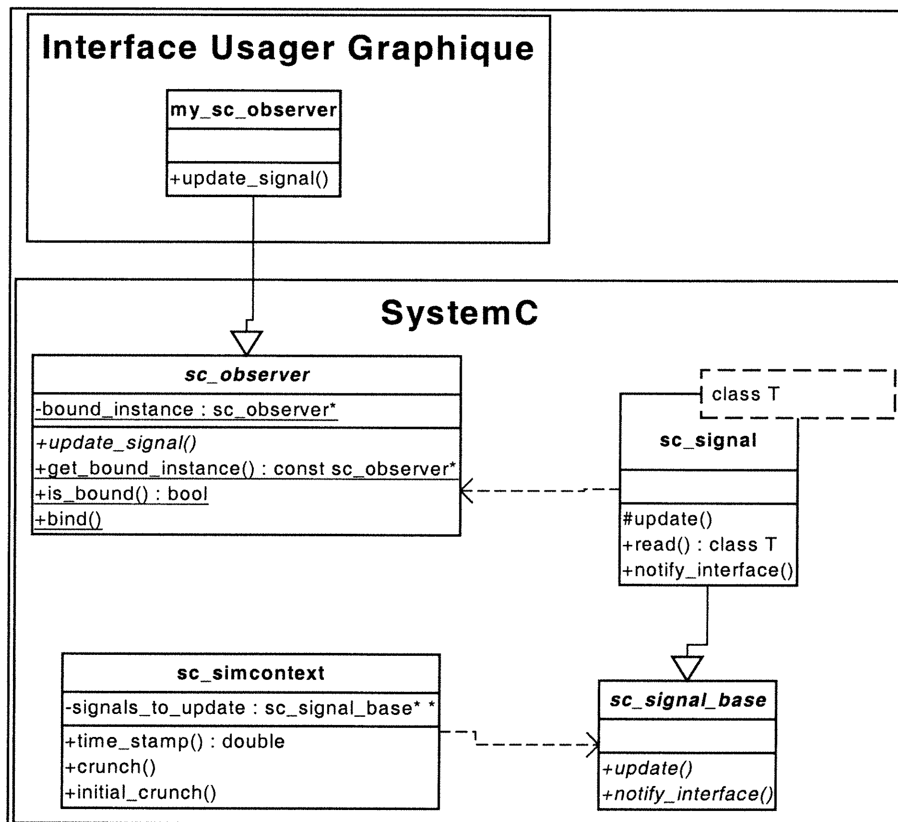


Figure 4 : Architecture modifiée du SystemC

Définie de cette manière, la classe `sc_observer` ne peut tenir compte d'une seule classe dérivée à la fois, nous pourrions généraliser la classe de manière à ce qu'elle puisse servir d'interface avec plusieurs classes dérivées, pour ce faire, nous aurions à changer le type de `bound_interface` d'un pointeur à, par

exemple, une liste de pointeurs et changer les trois méthodes statiques en conséquence.

Pour faire le lien avec SystemC, notre interface usager graphique (GUI) devra avoir une classe, par exemple : `my_sc_observer`, qui héritera de `sc_observer`. Dans `my_sc_observer` nous allons redéfinir toutes les méthodes pures virtuelles de `sc_observer`. Bien sûr, si une méthode de l'interface ne nous est pas utile, nous pouvons toujours la redéfinir vide. En plus de ces méthodes, la librairie que nous avons créée pour le GUI utilise les méthodes publiques du SystemC, par exemple, `sc_start()`, `sc_stop()`.

6.4 Interface graphique

L'objectif est donc de lier, à travers la classe `sc_observer`, la simulation de modèles en SystemC à la librairie graphique QT.

Nous allons prendre avantage de l'approche orientée objet avec laquelle QT a été implémenté pour automatiser, autant que possible, la création de notre application. Nos menus pour les modules et pour les signaux vont se créer à l'exécution en s'adaptant au contenu du modèle en SystemC. Nous utiliserons des méthodes publiques de SystemC pour nous créer une liste de modules et les informations transmises à travers `sc_observer` pour créer une liste de signaux. Notre application parcourra ces listes pour remplir ses menus et lier ceux-ci automatiquement à la fonction créneau correspondante. Donc notre application sera automatiquement adaptable au modèle.

Grâce à QT, notre application sera très extensible, nous pourrons facilement ajouter de la fonctionnalité à notre application, par exemple, nous pourrions utiliser des objets QSocket de QT pour permettre l'échange d'informations à travers un réseau.

QT facilite aussi la réutilisation, nous pouvons compiler notre application, sans changement, sur plusieurs plates-formes, nous pouvons aussi, avec très peu de travail additionnel, permettre le choix de la langue d'affichage et ce, même au moment de l'exécution.

6.5 Comparaisons avec l'interface graphique pour Win32

L'interface usager de la compagnie CODECSiL-AnsLab a une méthodologie très différente de la notre [18], en effet, ils utilisent MFC (*Microsoft Foundation Class*) pour les objets graphiques, ils ne font aucun changements dans le code ouvert de SystemC, mais imposent à l'usager, le soin de placer, dans le modèle, les appels aux méthodes du GUI. Comme exemple, à la Figure 5, ils montrent les feux de circulation d'une intersection, la fenêtre pour l'interface usager est une boîte de dialogue qui contient un affichage très spécifique au système modéliser.

Donc, les avantages de notre méthodologie sont :

- La portabilité, nous pouvons recompiler notre travail sur toutes les plates-formes où SystemC est disponible.
- La transparence vis-à-vis l'usager, celui-ci n'a pratiquement rien à changer dans son modèle pour pouvoir utiliser notre GUI. Avec la

méthodologie de CODECSiL-AnsLab, l'usager doit placer lui-même les appels à l'interface graphique dans le code de son modèle.

- La réutilisation, notre GUI s'adapte automatiquement au modèle, le lien est entre la simulation et notre application et non entre le modèle et le GUI.
- La flexibilité, nous pouvons décider à l'exécution les signaux que nous voulons afficher.

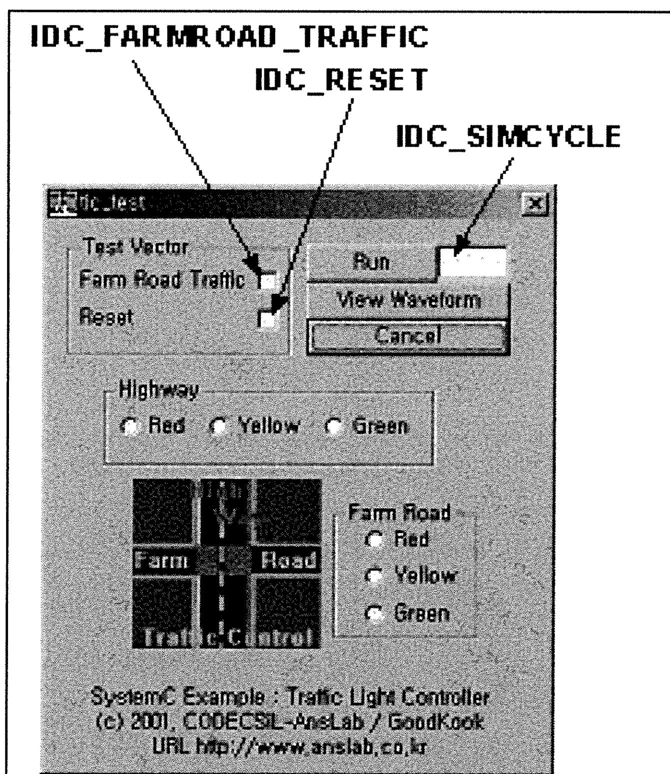


Figure 5 : GUI de CODECSiL-AnsLa

Par contre, à chaque changement de version de SystemC, nous devons faire les changements au code source de SystemC qui sont expliqués au chapitre suivant. Le modèle de CODECSiL-AnsLab lui va fonctionner tel quel sur toute nouvelle

version de SystemC. Malheureusement, un concepteur de système risque de changer de modèle plus souvent que de version de SystemC.

6.6 Construction de l'interface usager graphique (GUI)

Cette section donne plus de détail sur notre implémentation de la classe `sc_observer` ainsi que les modifications que nous avons fait pour permettre le passage de messages vers cette classe interface lors de la mise à jour des signaux de la simulation.

Pour notre interface usager, nous avons besoin de contrôler la simulation, nous utilisons pour cela, les méthodes publiques suivantes de la classe `sc_simcontext`: `sc_start()`, `sc_initialize()`, `sc_time_stamp()`.

Nous voulons aussi afficher la liste des modules, nous parcourons la liste des objets de `sc_object_manager` en utilisant les méthodes `first_object()` et `next_object()`. Pour chaque objet `sc_object` de la liste, nous vérifions si la méthode `kind()` retourne `sc_module`. Si oui, nous avons accès au nom du module par la méthode `name()`.

Pour les signaux, nous voulons, en plus d'obtenir le nom du signal, avoir accès à sa valeur. Malheureusement, la liste des signaux contient des pointeurs sur des objets de type `sc_signal_base`. Or, cette classe ne contient aucune donnée. De plus, nous ne pouvons même pas faire une conversion de type vers le type `sc_signal` puisque nous ne connaissons pas le type du signal qui est un gabarit.

Pour y avoir accès, nous allons devoir apporter quelques modifications au code original du SystemC et ce, en plus d'ajouter la classe sc_observer dont nous avons discuté précédemment.

Dans les méthodes crunch() et initial_crunch() de la classe sc_simcontext, nous avons accès à la liste des signaux à mettre à jour. Nous avons donc créé une nouvelle méthode pour cette classe, nous l'avons nommée notify_interface(), celle-ci prend en paramètres une liste de pointeurs sur des objets sc_signal_base, ainsi qu'un entier correspondant au nombre de pointeurs dans la liste. La Figure 6 nous montre la définition de cette méthode.

```
void  sc_simcontext::notify_interface(sc_signal_base**  signals_to_update,
int lastof_signals_to_update)
{
  int i;

  //for each signals we have to update, notify the interface
  for (i = lastof_signals_to_update; i >= 0; --i)
  {
    sc_signal_base* const sig = signals_to_update[i];
    sig->notify_interface();
  }
 }
```

Figure 6 : La méthode notify_interface() de la classe sc_simcontext

Tel que discuté précédemment, la méthode vérifie s'il y a une interface de liée. Sinon, l'exécution se fait selon le code original du SystemC. Si oui, pour chaque pointeur dans la liste, nous appelons la méthode notify_interface() de la classe sc_signal_base. Les appels à cette nouvelle méthode sont placés dans crunch() et initial_crunch() après la mise à jour des signaux et avant le remplacement des éléments de la liste.

Dans la clase sc_signal_base, nous déclarons la méthode notify_interface() virtuelle, car nous allons redéfinir celle-ci dans la classe sc_signal<T> qui en est la classe dérivée. Dans cette dernière, nous avons accès à la valeur du signal. Donc, lorsque par polymorphisme, la méthode notify_interface() est appelée, nous sommes certains qu'il existe une classe dérivée de notre classe abstraite sc_observer. Nous pouvons donc appeler une méthode déclarée dans sc_observer et redéfinie dans sa classe dérivée en lui passant, entres autres, la valeur du signal.

```
template< class T >
void sc_signal<T>::notify_interface()
{
  sc_observer::get_bound_instance()->update_signal(this, cur_value);
  }
```

Figure 7 : définition de la méthode notify_interface() de la classe sc_signal

Dans notre cas, nous appelons une méthode update_signal() avec un pointeur sur le signal (pour pouvoir l'identifier) et la valeur du signal. C'est ce que nous voyons à la Figure 7. Dans la classe sc_observer, nous devons donc avoir une déclaration de cette méthode qui est pure virtuelle, dont un des paramètres devrait être de type gabarit.

Or, si la classe sc_observer est une classe gabarit, nous devons avoir au moins une instance de cette classe par type de signal utilisé dans le modèle. Mais nous voulons avoir qu'une seule instance dérivée de sc_observer. Pour arriver à un résultat similaire avec une instance, nous n'avons pas trouvé d'autre solution que d'utiliser la surdéfinition de fonction pour déclarer une méthode

`update_signal()` pour chaque type possible de donnée. Nous nous retrouvons avec une seule instance dérivée qui peut traiter tous les types pour lesquels nous surdéfinissons la méthode.

La Figure 4 de la page 42 illustre la hiérarchie partielle du SystemC modifié. La partie qui est dans l'interface graphique n'est pas obligatoire pour que la simulation fonctionne. Dans la classe `my_sc_observer`, nous devons définir chaque méthode virtuelle de la classe `sc_observer`. Au besoin nous déclarons des méthodes vides.

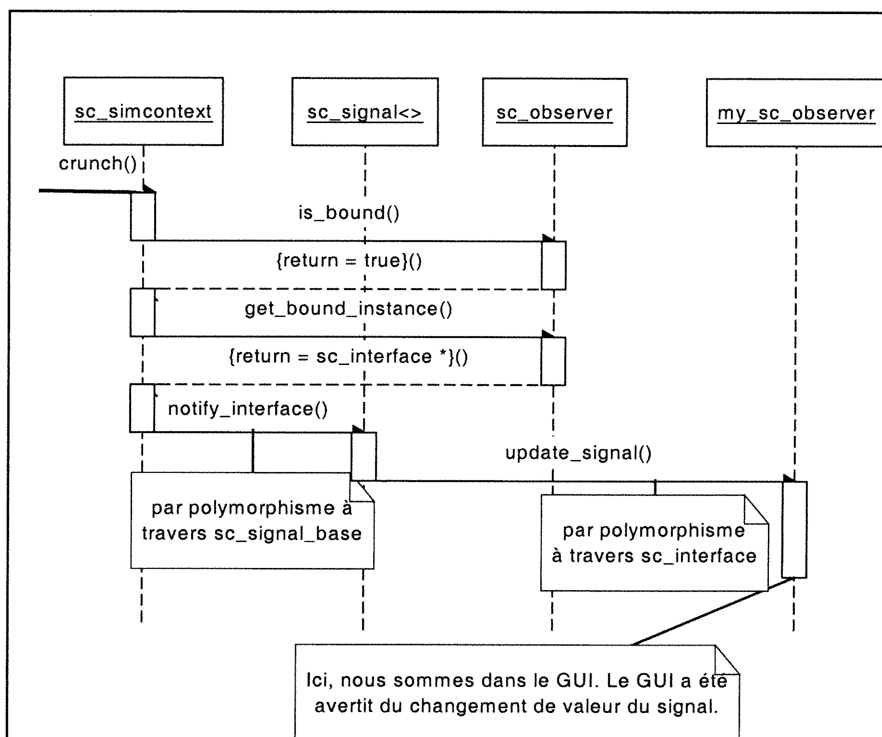


Figure 8 : Séquence d'appels pour rendre la valeur du signal accessible au GUI

La Figure 8 présente la séquence d'appels pour avertir l'interface usager que la valeur du signal a changée et transmettre la nouvelle valeur.

À ce stade ci, nous pouvons maintenant bâtir l'application que sera notre interface usager. Pour ce faire, nous allons utiliser la librairie graphique QT pour les divers éléments visuels de l'interface. Lorsque des objets comme des listes, des files ou des vecteurs sont nécessaires, nous utilisons la librairie STL [26].

6.6.1 Prototype de l'Interface Usager Graphique

Pour notre premier prototype, nous pouvons afficher la trace d'un signal booléen tel que montré à la Figure 9 et lister dans une fenêtre tous les changements de valeurs des signaux de tous les types de bases du C++ ainsi que les types simples de SystemC.

Malheureusement, il nous reste toujours du travail pour les types complexes du SystemC, c'est-à-dire, les types de données qui contiennent un gabarit, par exemple `sc_uint<32>`.

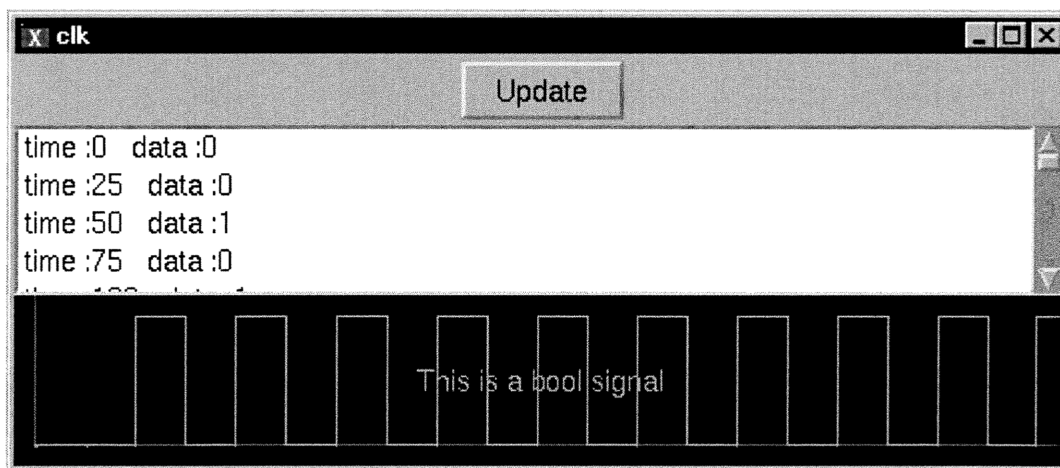


Figure 9 : Trace d'un signal booléen

Dans le cas de signaux de type défini par l'usager, ceux-ci ne sont pas pris en compte par notre classe `sc_observer`, il faut donc que l'usager ajoute lui-même une définition virtuelle de la méthode `update_signal()` pour ce nouveau type,

en plus, il faut redéfinir cette méthode dans la classe dérivée du GUI, dans notre cas, cette classe est my_sc_observer. Si le nouveau type est composé de types de base de C++ nous devrions contourner le problème élaboré précédemment.

Pour le moment, nous ne mémorisons que les valeurs des signaux qui ont été sélectionnés par l'usager. L'enregistrement des valeurs débute aussitôt que le signal est sélectionné. La vitesse de simulation ne devrait pas être affectée par les signaux qui n'ont pas été sélectionnés.

6.6.2 Changements requis au code du modèle de l'usager

Nous pouvons compiler notre application indépendamment du modèle et le lier en tant que librairie. Pour pouvoir utiliser l'interface usager, il faut remplacer l'appel à la fonction sc_start() dans la fonction sc_main() du modèle par les déclarations et appels nécessaires pour créer la fenêtre principale. De plus, pour améliorer la présentation, l'usager devrait utiliser un constructeur de la classe sc_signal<T> non-documenté pour initialiser le nom du signal. L'annexe I contient un guide qui explique en détails, les étapes à suivre pour créer une interface usager.

Cette solution a été implémentée avec la version SystemC 1.1 beta, au chapitre suivant, nous allons tester le coût de notre travail sur la vitesse de simulation d'un système, alors que dans le chapitre 9 nous allons discuter des nouveautés apportées avec la version 2.0 beta de SystemC ainsi que l'impact que cette nouvelle version aura sur notre travail.

Chapitre 7. Coût en terme de vitesse de simulation

Pour nous donner une idée de l'effet de nos changements sur la vitesse de simulation, nous avons fait un petit modèle qui consiste en une horloge et un signal booléen `output` dont la valeur change à chaque front montant de l'horloge. Pour chaque test, nous simulons le modèle pour exactement 100 cycles sans interruption.

À chaque cycle d'horloge, nous écrivons à la console le temps du système. Puisqu'en utilisant notre application nous devons avoir initialisé la simulation avant de choisir le signal pour lequel nous voulons enregistrer les valeurs, nous allons laisser tomber les premières valeurs de chaque test pour s'assurer que tous les signaux ont été initialisés avant de commencer les comparaisons. Ce que nous mesurons est, pour chaque cycle d'horloge, le temps écoulé depuis le début de la simulation.

Nous avons testé les cas suivants :

a. SystemC original, sans nos modifications et sans aucun affichage de résultat ou enregistrement de valeurs.

b. SystemC avec nos modifications mais sans lier notre interface usager. Ce cas diffère du précédant uniquement par l'exécution des tests dans `crunch()` pour déterminer s'il y a une interface de liée.

c. Avec notre interface usager liée mais sans enregistrer aucun signal. Donc en plus des tests du cas précédent, il y a dans ce cas le passage des

messages vers l'interface usager pour avertir des changements de valeurs pour les signaux. Par contre, aucune valeur n'est conservée.

d. SystemC original où la valeur du signal `output` est tracée grâce à une instruction `cout` (affichage à la console en C++) qui est redirigé vers un fichier. Donc, les valeurs ne sont visibles qu'à la fin de la simulation.

e. Comme le précédent, sauf que le `cout` écrit directement à la console. Ici, les différentes valeurs de `output` sont visible durant la simulation.

f. SystemC sans changement, qui n'utilise pas de sortie vers la console (`cout` ou `printf`) et où le signal output est tracé dans un fichier de type `vcd` tel que préconisé par le guide d'utilisateur de SystemC. Les résultats peuvent donc être accédés à la fin de la simulation, en utilisant un logiciel d'une tierce partie tels ***Dynotrace*** et ***Synopsys Waveform Viewer***.

g. Utilisant notre GUI et enregistrement du signal `output`. Il est à noter que, dans ce cas ci, nous n'avons pas interrompu la simulation pour faire afficher les résultats, bien que nous aurions pu, donc, les résultats ne sont visibles qu'après les 100 cycles.

h. Nous avons aussi testé avec notre GUI, l'enregistrement des deux signaux, soit `output` et `clock`.

La Figure 10 présente nos résultats. Nous avons trouvé que, lorsqu'il n'y a aucun signal de tracé, le SystemC modifié (cas b) et l'original (cas a) ont, à peu près

la même vitesse de simulation. Par contre, lier notre interface (cas c) augmente le temps de simulation d'environ 20%.

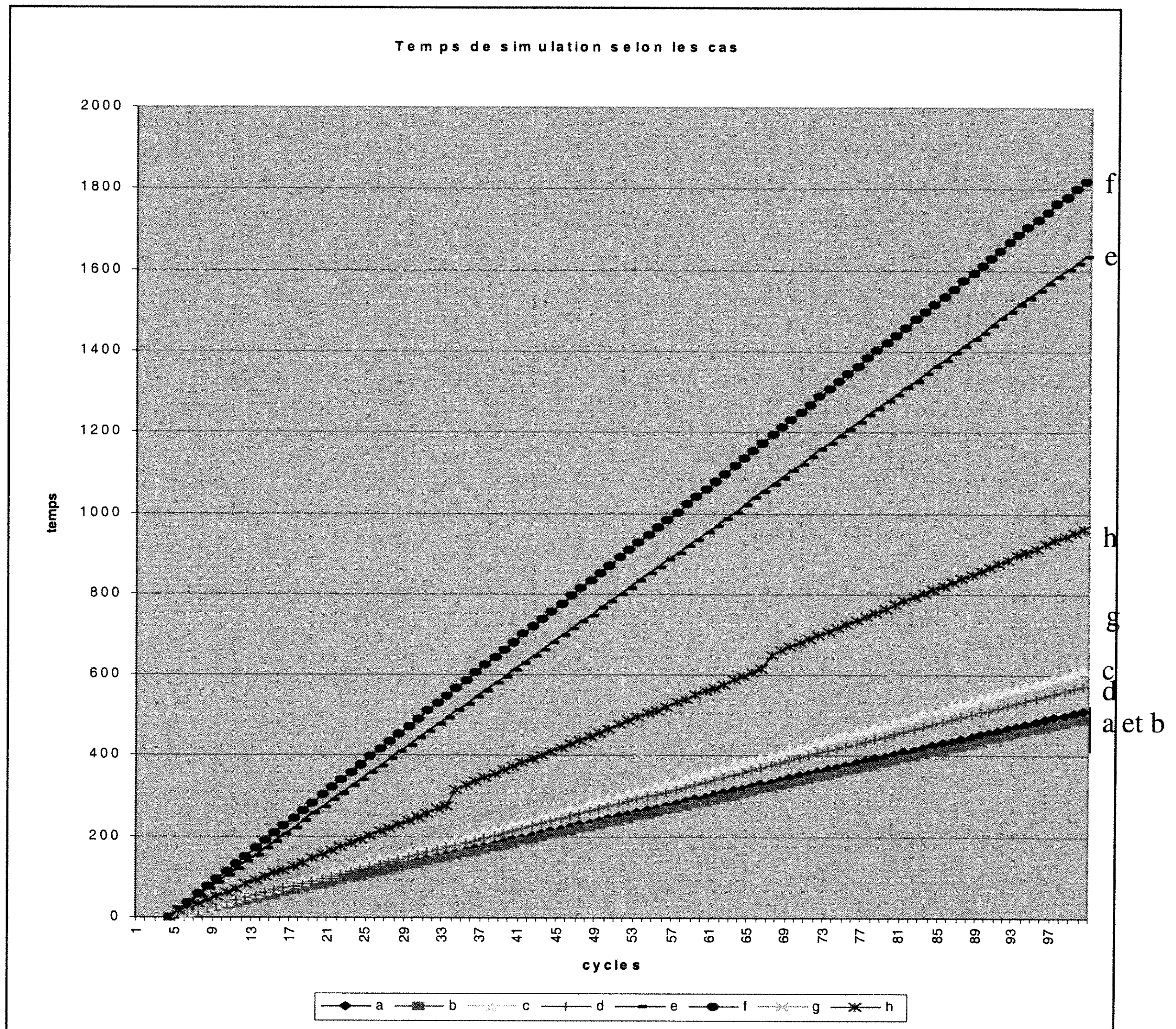


Figure 10 : Temps de simulation selon les cas testés

a) SystemC original sans affichage de résultat
b) SystemC modifié, sans lier notre interface usager
c) Avec interface usager liée, sans enregistrer de résultat
d) SystemC original, un résultat avec cout redirigé vers un fichier
e) SystemC original, un résultat avec cout à la console
f) SystemC original, résultat dans un fichier VCD
g) Enregistrement d'un résultat par l'interface usager
h) Enregistrement de deux résultats par l'interface usager

Lorsque l'on trace un signal, soit `output`, le temps de simulation le plus rapide correspond au SystemC non modifié dont les valeurs du signal envoyées vers la console sont redirigées vers un fichier soit le cas d. En comparaison, enregistrer le signal grâce à notre interface (cas g) implique que la simulation est 30% plus longue, alors que faire tracer le signal dans un fichier `vcd` par le SystemC original (cas f) prend 320% du temps pour le cas d. Par contre, si nous envoyons les résultats directement à la console en se servant de `cout`, le temps de simulation est alors 285% du temps pour le cas redirigé.

Donc, il y a un accroissement non négligeable du temps de simulation en utilisant notre interface par rapport à utiliser des `printf` ou des `cout` dans le code, du moins en autant que ceux-ci sont redirigés dans un fichier. Par contre, notre interface apporte beaucoup de flexibilité, en effet, nous pouvons arrêter momentanément la simulation, consulter les résultats, choisir un autre signal et continuer l'exécution de la simulation. Sans notre interface, tout le code pour tracer les signaux doit être écrit avant la compilation, si nous voulons consulter les résultats d'un autre signal, nous devons terminer la simulation, faire les changements dans le modèle, recompiler puis redémarrer la simulation.

De plus, si plusieurs signaux sont tracés, les valeurs de tous ces signaux seront affichés en ordre séquentiel d'exécution, la console (ou le fichier) va contenir les valeurs entrecoupées de plusieurs signaux. Il sera donc difficile, lorsque le nombre de signaux grandit, de suivre les valeurs par signal. Dans notre interface, chaque signal a sa propre fenêtre, donc les valeurs d'un signal sont isolées des autres signaux, donc plus facile à se retrouver.

Notre test n'est pas très rigoureux, nous voulions avoir une idée de l'ordre de grandeur du coût en temps de simulation avant de décider s'il est intéressant de généraliser cette approche avec, entres autres, les *design patterns*. Il faut noter que, pour le test nous utilisons qu'un processus synchrone, donc lorsque la méthode `crunch()` est exécutée, il ne devrait avoir qu'un delta cycle et donc, le test qui vérifie s'il y a une interface de liée ne devrait s'effectuer qu'une fois par front d'horloge. Un modèle qui contient beaucoup de processus asynchrones aurait effectué ce test plusieurs fois par cycle ce qui pourrait influencer fortement les résultats du test.

Chapitre 8. Changements annoncés pour la version 2.0 de SystemC

Dans le version 1.0 de SystemC, les processus asynchrones (*Thread Process*) synchrones (*Clock Thread Process*) ne sont pas exécutés lors de la phase d'initialisation de la simulation, alors que les blocs asynchrones (*Method Process*) peuvent être exécutés.

Pour être conséquent avec les langages de description de matériel, tel VHDL, la version 2.0 de SystemC exécutera tous les processus à l'étape de l'initialisation de la simulation.

8.1 Modélisation du temps de simulation

Le temps en SystemC 1.0, correspond à un modèle relatif à valeur réelle, l'horloge utilise le type double et les unités de temps n'ont aucun lien avec le temps absolu, tel des secondes, nanosecondes, etc. Pour permettre à des compagnies ou des particuliers de développer des modèles et les distribuer, il devrait être possible de spécifier une unité de temps absolue.

La prochaine version utilise un modèle de temps absolu et à valeur entière. Le temps est représenté en utilisant une valeur entière non-signée de 64 bits. La résolution par défaut sera de une picoseconde et l'unité de temps par défaut est de une nanoseconde. SystemC 2.0 contient deux méthodes permettant de changer ces

deux dernières valeurs, soient `sc_set_time_resolution()` et `sc_set_default_time_unit()`.

Quelques problèmes avec ces différents modèles de temps :

- Dans le modèle réel, le plus grave problème est lié à la précision entre deux opérandes dans un calcul de temps, si une très petite valeur de temps est ajoutée à une très grande valeur, il est possible que le résultat soit la très grande valeur, donc la très petite a été perdue puisque plus petite que la précision possible.
- Dans le modèle entier, il arrive que l'on ait débordement de la valeur, si celle-ci est trop grande pour être représentée avec le nombre d'octets disponibles.

Si le modèle entier peut représenter des valeurs de temps suffisamment grandes, alors il est préférable au modèle réel.

8.2 Sensibilité dynamique et évènements

La version courante de SystemC ne permet qu'une liste de sensibilité statique, les processus ne sont sensibles qu'aux changements de valeurs sur les signaux ou ports qui sont dans la liste. De plus celle-ci ne peut être changée durant toute la simulation.

Pour la version 2.0, SystemC annonce la possibilité d'utiliser la notion d'événements pour provoquer l'activation d'un processus. La liste de sensibilité statique est encore supportée, mais la méthode `wait()` a été modifiée de manière à accepter comme paramètres, des objets de la classe `sc_event` qui sont les

évènements. Un processus peut donc avoir une liste de sensibilité (statique) et contenir des instructions `wait()` avec un ou plusieurs évènements dans son code pour augmenter sa sensibilité.

Les objets de la classe `sc_event` ne contiennent aucune donnée, ceux-ci ont une méthode `notify()` qui sert à déclencher un événement, cette méthode, utilisée sans paramètre, envoie un message immédiatement aux processus qui sont suspendus et attendent cet événement. Utilisée avec un paramètre, soit une valeur de temps, le message sera envoyé après ce délai. Le temps spécifié peut correspondre à un delta cycle ou encore un temps absolu.

Il existe aussi une fonction `notify()` pour faire le même travail, celle-ci prend un paramètre de plus que la méthode du même nom, soit l'événement pour lequel nous désirons envoyer le message.

Lors de l'utilisation d'un `wait()` avec un paramètre, la liste de sensibilité statique est remplacée, momentanément, par l'attente de l'événement spécifié en paramètre. Cet événement peut être, soit une unité de temps, soit un objet de la classe `sc_event` ou encore, une expression composée, soit de `ou` logique entres plusieurs objets `sc_event` ou de `et` logique entres des objets `sc_event`. Un mélange de `et` et de `ou` n'est pas permis. Lorsque l'expression est satisfaite, le processus redevient actif et sa liste de sensibilité statique est rétablie.

8.3 Interfaces, ports et canaux

La version 2.0 du SystemC va généraliser, en un petit noyau, le langage. Ce noyau contient des objets de base tels les événements, les interfaces, les ports et

canaux. Ceux-ci peuvent être ensuite, dérivés pour former des objets plus compliqués et très spécialisés. Le noyau devrait être maintenant assez général pour ne susciter, à l'avenir, que très peu de changements.

Ces objets forment la base pour la modélisation des communications et de la synchronisations. Tout semble en place pour que les améliorations futures du SystemC ne soient que des nouvelles librairies d'objets spécialisés dérivées des diverses classes de bases du noyau de systemC.

Une interface spécifie les méthodes de communications qui doivent être implémentées dans un canal. Elle ne contient que des définitions purement virtuelles de ces méthodes, aucune implémentation ou donnée. Les interfaces permettent de connecter des ports de communication aux canaux.

Les canaux implémentent une ou plusieurs interfaces par héritage simple ou multiple. Toutes méthodes virtuelles dont le canal hérite doivent être redéfinies. Il y a deux familles de canaux, les canaux de bases et les canaux hiérarchiques.

Les canaux de base ne contiennent aucun processus et ne peuvent accéder directement à d'autres canaux, ceux-ci implémentent une approche requête/mise à jour. Les signaux sont un exemple de canaux de bases.

Les canaux hiérarchiques sont les modules, tout comme pour la version 1.0 du SystemC, ils peuvent contenir d'autres modules, des processus, des données et peuvent même accéder directement à d'autres canaux.

Pour permettre la communication entres les modules et d'autres canaux, il y a les ports de communications. Ceux-ci sont des objets à travers lesquels les modules et leurs processus peuvent accéder l'interface d'un autre canal. Un port est

rattaché à un canal grâce à l'interface que ce dernier implémente. Pour un port, les seules méthodes du canal qui lui sont visibles sont celles qui sont déclarées dans l'interface que ce canal implémente.

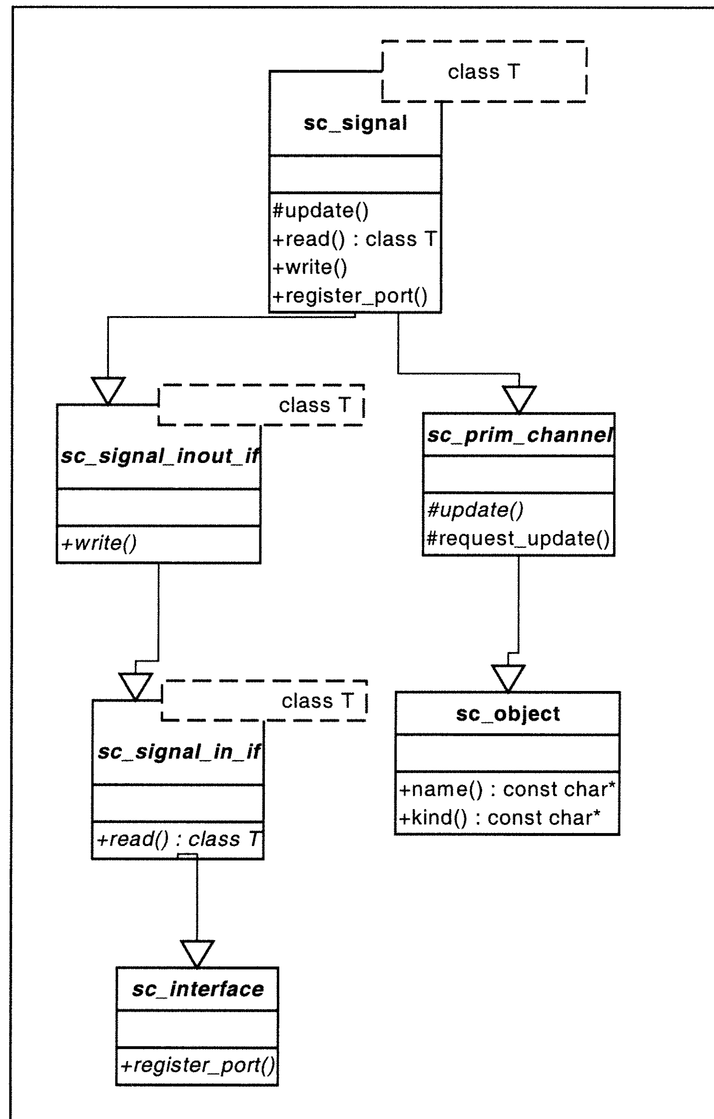


Figure 11: Construction de la classe sc_signal<T>

Ce noyau contient tous les éléments nécessaires pour définir les signaux. Ceux-ci ne font donc plus partie de ce noyau mais sont plutôt implémentés comme un des canaux élémentaires fournis avec le langage.

La construction de la classe des signaux à partir d'objets provenant du noyau du SystemC est illustrée par la Figure 11. Nous y voyons que les signaux proviennent de l'héritage multiple des classes sc_prim_channel et sc_signal_inout_if. La première est une classe abstraite, dérivée de la classe sc_object, qui déclare les méthodes pures virtuelles update() et request_update() qui seront redéfinies dans sc_signal<T>.

La seconde est une classe interface qui déclare une méthode pure virtuelle write(). Cette classe est elle-même dérivée de la classe interface sc_signal_in_if qui déclare la méthode read(). Cette dernière est dérivée de la classe de base sc_interface. Toutes les méthodes de ces classes abstraites seront redéfinies dans la classe sc_signal<T>.

Canaux standards, MOC variées Kahn Process Networks Static Dataflow, etc.	**Canaux spécifiques à la méthodologie** Librairie Maitre/Esclave,etc.
Canaux Élémentaires Signaux, Timer, Mutex, Sémaphore, Fifo, etc.	
Noyau du Langage Modules Ports Processus Intefaces Canaux Évènements	**Types de données** Type logique (01XZ) Vecteurs logique Bits et vecteurs de bits Réels de précisions arbitraires Entiers
Langage C++ Standard	

Figure 12 : Architecture du SystemC 2.0

La Figure 12 donne un aperçu des couches qui composent le SystemC 2.0 [1], nous y voyons qu'en plus des signaux, il y a plusieurs autres canaux élémentaires qui sont inclut dans la librairie du systemC 2.0. Le OSCI met en place des procédures pour permettre aux usagers de contribuer plus facilement, à de nouvelles classes dans la librairie de SystemC. Ces nouvelles classes pourront aussi être téléchargées facilement par quiconque désire les incorporer dans son modèle.

Tous ces changements permettent d'utiliser davantage l'approche Orientée-Objet avec SystemC, par contre, il n'y a pratiquement pas de changements dans la méthodologie proposée dans le guide d'usager de la version SystemC 2.0 Beta .

Ces nouveaux éléments visent à favoriser le développement de librairies spécialisées par les concepteurs de systèmes ainsi que la création et l'échange de blocs IP.

En effet, il est maintenant plus facile de définir les communications de l'implémentation des canaux, ce qui permet le raffinement successif des canaux ou encore l'utilisation de canaux spécialisés écrits par d'autres concepteurs.

8.4 Impact sur notre travail

Notre but étant de permettre l'évolution indépendante de l'interface usager et de SystemC, la version 2.0 et les changements importants qui y sont implémenté, surtout au niveau des signaux, nous permettent, malgré nous, de valider notre approche.

Le code de la version 2.0 Beta n'étant disponible que depuis quelques semaines, nous n'avons pas encore essayé d'adapter notre approche en fonction de

cette version. Par contre, en examinant le code source de la nouvelle librairie de SystemC, nous notons la disparition de la classe `sc_signal_base`, que nous utilisions dans la séquence de passage de messages, à travers notre classe `sc_observer` (voir Figure 8 à la page 49) . De plus, la liste des signaux à mettre à jour, de la classe `sc_simcontext`, n'existe plus. Cette liste, dont la gestion se faisait dans la méthode `crunch()`, était utilisée par nos modifications pour accéder les signaux du modèle et pour bâtir, du coté de l'interface usager, la liste des signaux du modèle disponibles pour le GUI. Donc, les messages de la Figure 8 qui ont leurs points de départ dans la classe `sc_simcontext` devront être relocalisés. Bien sûr, ceux-ci seront toujours dans cette classe mais nous devons maintenant étudier ce qui remplace cette liste des signaux à mettre à jour.

Avant de discuter des possibilités de remplacement pour ces premières étapes, analysons la validité des étapes subséquentes. Sous l'hypothèse que nous réussissons, au moins à chaque changement de valeurs d'un signal, à générer un appel à la méthode `notify_interface()` de la classe `sc_signal<T>`, la méthode `update_signal()`, que nous avons créé pour la version précédente de SystemC, peut passer le message à `sc_observer`, celui-ci peut alors fonctionner sans modification comme dans la version précédente.

Si l'hypothèse est vérifiée, la partie GUI de notre projet n'est nullement affectée par les changements majeurs du SystemC. D'ailleurs, même la nouvelle classe `sc_observer` que nous avons créée peut être utilisée sans changement puisqu'elle reçoit un message d'un objet `sc_signal` qui lui connaît son adresse ainsi que le type et la valeur de sa donnée.

8.4.1 Mise à jour des signaux

Explorons maintenant la nouvelle approche de SystemC pour mettre à jour les valeurs des signaux en quête d'une nouvelle façon de remplacer les premières étapes des messages. À la phase de mise à jour de la méthode `crunch()` nous trouvons un appel à la méthode `perform_update()` d'un objet `sc_prim_channel_registry`, cette classe contient un vecteur de référence vers des objets dérivés de `sc_prim_channel`, qui est une classe de base pour tous les canaux de base de SystemC. De plus, `sc_prim_channel_registry` contient aussi une liste, `m_update_array`, des canaux de base nécessitant une mise à jour, la méthode `perform_update()` de cette classe, appelle, pour chaque élément de cette liste, la méthode `perform_update()` de la classe `sc_prim_channel` qui à son tour, appelle la méthode `update()` qui est virtuelle et est redéfinie dans `sc_signal<T>`.

La liste `m_update_array` de `sc_prim_channel_registry` ressemble, en plus général, à la liste des signaux à mettre à jour des versions précédentes de SystemC. Notons que la classe `sc_signal<T>` profite d'héritage multiple, soit de `sc_prim_channel` et de `sc_signal_inout_if<T>`. Donc, tout comme nous débutions la séquence de messages dans `crunch()` en passant, par polymorphisme, à travers `sc_signal_base`, classe dont `sc_signal` héritait dans les versions précédentes de SystemC, il semble possible de faire quelque chose de similaire à travers `sc_prim_channel_registry`, qui contient la liste des canaux à mettre à jour, ceux-ci sont des objets dérivés de `sc_prim_channel`, qui est abstraite. Parmi ces objets, se trouvent tous les signaux. Si nous ne désirons que

les signaux parmis les canaux de base du SystemC, nous aurons besoin, lorsque nous parcourrons la liste des canaux de base, de vérifier, grâce à la méthode `kind()` de `sc_object`, que l'élément courant de la liste est bien un pointeur vers un signal et donc, par polymorphisme, appeler la méthode `notify_interface()` du signal.

Nous semblons donc avoir trouvé une bonne piste pour implémenter une séquence de messages similaire à nos modifications de la version précédente de SystemC. Si cette piste s'avère la bonne, nous pourrions prendre avantage du fait que, la nouvelle classe de base `sc_prim_channel` est la même pour tous les types de signaux de bases, dont les fifo, sémaphore, mutex, et tout canal de base futur. Ainsi, nous pourrions modifier notre classe `sc_observer` de façon à permettre au GUI d'être au courant de tous ces canaux et non uniquement les signaux.

8.4.2 Les ports

Dans la nouvelle version de SystemC, les ports sont maintenant dérivés, indirectement, de la classe `sc_object`. De plus, il y a aussi une classe `sc_port_registry` qui contient un vecteur avec les références vers tous les ports de la simulation. Nous aurions intérêt à explorer ces changements dans le but de régler les problèmes que nous avons dans la version précédente de SystemC.

Chapitre 9. Problèmes non résolus et travaux futurs

Nous n'avons pas adapté notre approche pour obtenir tous les types de signaux de SystemC, il faudrait tester, sous la version 2.0 de SystemC, si le problème reste entier.

Nous aimerions améliorer la présentation des résultats de simulation si nous pouvions afficher les signaux et les ports selon les modules qui les utilisent. Nous n'avons malheureusement pas encore réussi. Nous avons tenté d'obtenir l'information sur le module du signal lui-même. La classe `sc_signal_base` contient un membre, `created_by`, qui est un pointeur vers le `sc_objet` qui a créé le signal. Étrangement, ce membre n'est pas initialisé pour la plupart des signaux auxquels nous avons accès. Lorsque `created_by` a une valeur non-nulle, il pointe vers un `sc_object` pour lequel les membres `name` et `kind` ne sont pas initialisés.

En fouillant dans le code source, nous trouvons quelques constructeurs pour la classe `sc_signal<T>`, l'un de ceux-ci prend un paramètre qui est une chaîne de caractères correspondant au nom du signal, c'est celui que nous suggérons à l'usager d'utiliser de manière à afficher un nom significatif pour le signal dans le GUI. Un autre constructeur prend en paramètre un pointeur sur un `sc_object`, c'est ce constructeur qui initialise le membre `created_by`. Malheureusement, aucun constructeur ne prend ces deux paramètres et comme nous voulons que l'usager

utilise le premier, nous ne pouvons avoir l'autre en même temps. Dans le SystemC 2.0, le deuxième constructeur n'existe plus.

Une autre voie qui semble intéressante c'est d'approcher le problème du point de vue du module. La classe `sc_module` a un membre nommé `port_vecteur`. Celui-ci semble être un vecteur contenant la liste des ports du module. La prochaine étape devrait être d'explorer cette possibilité.

Comme l'implémentation des ports a subit d'importants changements, dans la nouvelle version de SystemC, nous devrions explorer le code dans l'espoir de trouver des méthodes nous permettant de trouver la hiérarchie du modèle.

Nous devons modifier notre séquence de messages qui avise le GUI des changements de valeurs des signaux pour le SystemC 2.0. Nous pourrions aussi explorer la possibilité pour le GUI de tenir compte de tous les canaux de base de SystemC.

Chapitre 10. Conclusion

Nous avons construit une interface usager graphique pour afficher les résultats de simulation de modèles en SystemC. Nous voulions que le GUI puisse évoluer indépendamment de SystemC et vice-versa. Malgré les changements majeurs apportés par le SystemC 2.0, il semble que la partie interface graphique puisse être utilisée comme telle, les changements à faire sont dans le code de SystemC, nous avons fourni une piste intéressante pour effectuer remplacer les premières étapes de la séquence de passage de messages vers le GUI. Si cette piste s'avère fructueuse, alors la classe `sc_observer` qui nous sert d'interface avec le GUI pourra être utilisée telle quelle, sans changement.

Nous avons aussi réussi à ne pas imposer de changement dans la façon de décrire le modèle, à l'exception du `sc_start()` que l'usager devra remplacer par la déclaration et la création d'une instance de la fenêtre principale de l'interface usager. Un autre changement est aussi fortement suggéré, il s'agit de l'utilisation du constructeur de la classe `sc_signal` qui permet de donner un nom au signal. Bien entendu, l'usager peut aussi enlever tous les `printf` et `cout` de son modèle.

Notre travail est fait de façon à ce que si l'usager n'effectue pas les modifications discutées dans le paragraphe précédent, la simulation de son modèle se fait de manière classique, en utilisant une interface textuelle via une fenêtre console. Dans ce cas, nous croyons que les changements que nous avons apportés

au code original du SystemC n'influencent pas, de façon significative, la vitesse de simulation.

Notre expérimentation nous porte à croire que, lorsque notre interface usager graphique est liée, nous obtenons une simulation qui est plus rapide et offre plus de flexibilité que pratiquement tous les autres cas testés, à l'exception d'appels à des instructions `cout` qui seront redirigés vers un fichier. Par contre, ce cas manque de flexibilité, tout doit être fixé à la compilation, si nous désirons suivre les valeurs de plusieurs signaux, les résultats seront affichés en ordre séquentiel d'exécution des `cout`, donc, il est peu probable que toutes les valeurs d'un signal en particulier seront regroupées.

Notre travail a été accompli tout en gardant les modifications dans le code source du SystemC au minimum, nous avons introduit environ 50 nouvelles lignes de code dans les classes originales de SystemC. La plupart des changements sont concentrés dans une seule classe abstraite qui nous sert d'interface avec notre librairie d'interface usager graphique.

Dans le code du SystemC, nous avons respecté, autant que possible, l'encapsulation des données imposée par les concepteurs du SystemC. Nous avons utilisé les méthodes déjà existantes pour accéder aux informations nécessaires. Les changements apportés dans les classes originales du SystemC nous ont permit d'établir une séquence de passage de messages vers notre classe `sc_observer`.

Notre classe `sc_observer` peut être dérivée telle quelle pour interfacer n'importe quelle librairie ou application ayant besoin de connaître la valeur des signaux d'un modèle, sans pouvoir changer ces valeurs. Nous avons donc atteint

notre objectif de permettre la réutilisation de nos développements. Il est aussi possible de généraliser notre classe pour permettre de lier d'autres types d'applications avec la simulation de SystemC. Dans ce sens, nous offrons la possibilité d'augmenter l'extensibilité de SystemC.

L'utilisation de la librairie graphique QT, pour développer notre application GUI, nous offre plusieurs possibilités intéressantes. En effet, cette librairie a été conçue pour permettre le plus possible la réutilisation. Nous pouvons recompiler notre application sur toute autre plate-forme supportée par QT et obtenir, sans changements de notre code, une application ayant l'apparence standard sous cette plate-forme. Nous pouvons aussi, en construisant les tables de traductions nécessaires, permettre le choix de plusieurs langues pour l'affichage des messages par notre GUI. De plus, nous pouvons facilement introduire de nouveaux objets à notre interface graphique avec peu ou pas de changements dans le code déjà existant.

Bibliographie

[1] Stuart Swan, "An introduction to SystemC Level Modeling in SystemC 2.0", Cadence Design Systems Inc, Mai 2001.

[2] OSCI, "Functional Specification for SystemC 2.0", www.systemC.org

[3] *SystemC Version 1.0 User's Guide*, Synopsys Inc., CoWare & Frontier Design, 2000

[4] *SystemC Version 2.0 Beta-1 User's Guide*, Synopsys Inc., CoWare & Frontier Design, juillet 2001

[5] *QT Cross-platform C++ GUI Application Framework Technical Overview V 1.3*, 2000, 29 pages. Disponible à ftp.trolltech.com/qt/pdf/whitepaper.pdf

[6] D. Gajski, F. Vahid, S. Narayan et J. Gong, *Specification and Design of Embedded Systems*, Prentice-Hall, 1994, 450 pages.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*: Addison Wesley, 1994

[8] R. Allen, D. Gajski, "The case for C/C++ hardware design", EEdesign.com, juin 2000, 6 pages

[9] G. Prophet, "System-level design languages: to C or not to C", EDNmag, 14 novembre 1999, 8 pages

[10] G. Moretti, "Get a handle on design language", EDNmag, 5 juin 2000, 7 pages

[11] D. Verkest, J. Kunkel, F. Schirrmeister, "System Level Design Using C++", *Proceedings of DATE2000*, pp. 74-81, Paris, mars 2000

[12] R. Goering, "Designers cautiously try out C-language tools", EE Times, 8 mars 2001, 4 pages

[13] A. Fin, F. Fummi, M. Martignano, M. Signoretto, "SystemC: A Homogenous Environment to Test Embedded Systems", *Proceedings of the Ninth International Symposium on Hardware/Software Codesign* (CODES2001), pp. 17-22, avril 2001

[14] G. Martin, L. Lavagno, J-L. Guerin, "Embedded UML: a merger of real-time UML and co-design", *Proceedings of the Ninth International Symposium on Hardware/Software Codesign* (CODES2001), pp. 23-28, avril 2001

[15] R. Goering, “UML language eyes system-level design role”, EE Times, 29 septembre 2000, 3 pages

[16] J. Fernandes, R. Machado, H. Santos, “Modeling Industrial Embedded Systems with UML”, *Eighth International Workshop on Hardware/Software Codesign* (CODES2000), pp. 18-22, San-Diego, Californie, mai 2000

[17] T. Lewis, L. Rosenstein, W. Pree, A. Weinand, E. Gamma, P. Calder, G. Andert, J. Vlissides, K. Schmucker, *Object-oriented application frameworks*, Manning, 344 pages, 1995

[18] CODECSiL-AnsLab Co. Ltd, “Interfacing SystemC with Win32 Software”, 27 pages, 23 juin 2001, disponible à : http://www.anslab.co.kr/ip_products/interface_SC_Win32.html

[19] S. Krishnamurthi and M. Felleisen, “Toward a formal theory of extensible software”, *Proceedings of ACM SIGSOFT sixth international symposium on Foundations of software engineering*, pp. 88 - 98, 1998.

[20] M. Reid, L. Charest, A. Tsikhanovich, E. M. Aboulhamid, G. Bois, “Implementing a Graphical User Interface for SystemC”, *The 10th Annual International HDL Conference & Exhibition proceedings*, pp. 224-231, Santa Clara, Californie, mars 2001

[21] L. Charest, M. Reid, E.M. Aboulhamid and G. Bois, “A Methodology for Interfacing Open Source SystemC with a Third Party Software”, *Proceedings of DATE2001*, pp. 16-20, Munich, mars 2001

[22] Site web officiel de OSCI, www.systemc.org

[23] Site web officiel de SpecC, http://www.specc.org

[24] Site web official de Cynlib, www.cynapps.com

[25] Site web official de Superlog, www.co-design.com

[26] Silicon Graphics Computer Systems, *Standard Template Library Programmer's Guide*, http://www.sgi.com/Technology/STL , 1999

Annexe I : Création de l'Interface Graphique

Le texte qui suit documente notre approche pour ajouter une interface usager graphique à un modèle SystemC tout en gardant les changements au modèle même au minimum.

Le pivot de l'interface est la classe `MainWindow`.

La section suivante explique comment, en utilisant QT, cette classe crée la fenêtre principale, incorpore les menus, accède les listes des modules et des signaux et cherche la période de l'horloge.

Fenêtre principale

Pour créer notre application, nous declarons la fonction `CreateMainWindow` comme suit:

```
void CreateMainWindow(QApplication **application,
      MainWindow *mainWindow,    int argc, char *argv[])
{
  // set high color mode & default font
QApplication::setColorSpec(QApplication::ManyColor);
QApplication::setFont(QFont("Arial",10,QFont::Normal));

// creation of an application (handle main loop event)
*application = new QApplication(argc, argv);

// creation of the central window
*mainWindow = new MainWindow(NULL, NULL);

(*mainWindow)->resize(300, 200);
// connected the main window to the application
(*application)->setMainWidget (*mainWindow);
(*mainWindow)->show();
 }
```

Cette fonction crée une instance de `Qapplication` et fait le lien avec `mainwindow`, l'instance de `QmainWindow`.

Menus

Les menus sont des instances de `QmenuBar,QPopupMenu`.

Nous commençons par créer un objet `QMenuBar` :

```
QMenuBar *menuBar;

menuBar = new QMenuBar (this);
```

Ensuite, nous créons les *pop-up* menus. Déclarons les fonctions qui seront appelées par les menus comme étant des créneaux publics (*public slots*) pour la classe `MainWindow`.

Ex. :

```
public slots:
void start();
```

Ensuite, nous créons une instance d'un `QPopupMenu` :

```
QPopupMenu *popMenu;

popMenu = new QPopupMenu();

popMenu->insertItem("Start",this,SLOT(start()), CTRL+Key_A);
```

Ceci affichera ***Start*** comme texte du menu, et, lors d'un click ou un raccourci clavier, appellera la fonction `start()` de cet object.

Pour créer un élément ***quit*** dans le menu, nous nous référons à l 'objet `Qapplication` qui a déjà un créneau quit() de défini.

Pour insérer un *pop-up* menu dans la barre des menus :

```
MenuBar->insertItem ("Simulation",popMenu,1);
```

Ceci affichera le texte Simulation comme nom du menu sur la barre des menus et fera la liaison avec le menu `popMenu`.

Spin box

Un *spin box* est une instance d'un `QSpinBox`.

Pour qu'elle agisse comme conteneur, nous avons besoin d'un objet Qwidget qui sera sont parent, nous utilisons une barre d'outils (*toolbar*).

Nous pouvons spécifier les valeurs min et max ainsi qu'une valeur par défaut.

La *toolbar* est une instance de `QtoolBar`.

```
toolBar = new QToolBar   (this);

spinBox = new QSpinBox  (MIN_STEP_SIZE, MAX_STEP_SIZE,
                                   clk->period(), toolBar);
spinBox->setValue(clk->period());
spinBox->setSuffix("ns");
```

Barre de statut

La barre de statut est une instance de `QstatusBar`

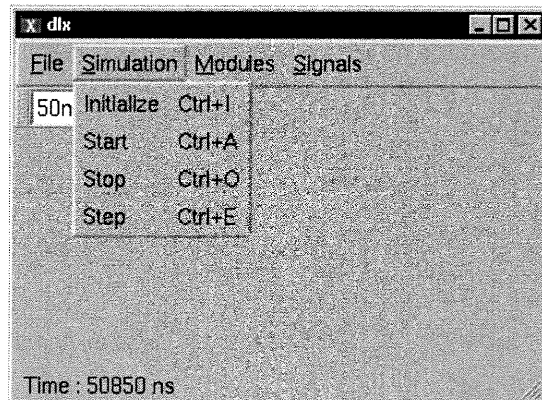
```
StatusBar = new QstatusBar(this);
```

Contrôle de la simulation

Maintenant que nous avons vu comment créer un menu, voici la liste des menus que nous utilisons et comment définir les créneaux (*slots*) qu'ils appellent pour contrôler la simulation.

Menu **start** :

Un des endroits d'où nous appelons la fonction de SystemC `sc_start()`.



Ici, nous permettons à la simulation de faire un pas correspondant à un cycle d'horloge, ensuite, si aucun événement intervient ou change la valeur de `mustrun`, on effectue un autre cycle.

L'appel à `update()` permet la mise à jour de notre liste

```
void MainWindow::start()
{
  mustrun = true;
  while(mustrun)
  {
    sc_start(clk->period());
    update();
    qApp->processEvents();
  }
}
```

Menu **stop** :

Ce menu change la valeur de `mustrun` pour forcer la boucle dans `start()` de se terminer.

```
void MainWindow::stop()
{
  mustrun = false;
}
```

Menu **step** :

Ce menu permet l'exécution de la simulation pour la durée de temps spécifiée par l'usager dans le *spin box*.

```
void MainWindow::step()
{
  sc_start(spinBox[0]->value());
  update();
}
```

Menu **initialize** :

Appelle la méthode de SystemC `sc_initialize()`.

```
void MainWindow::initialize()
{
  sc_initialize();

}
```

Affichage du temps de simulation et mise à jour des signaux.

L'appel à `update()` écrit le temps courant de la simulation, obtenu par un appel à la méthode `sc_time_stamp()` de SystemC.

De plus, nous demandons une mise à jour de tous les signaux de notre liste.

```
void MainWindow::update()
{
  //Update message bar status
statusBar->message(QObject::tr("Time : %1
ns").arg(sc_time_stamp()));

  //for each child...
for(list<SignalWindow *>::iterator i =
                          SignalWindow::signalWindows.begin();
      i != SignalWindow::signalWindows.end(); i++)
    if (*i)
      (*i)->update();
}
```

Accès à la liste des modules

Nous obtenons la liste des modules du modèle directement de SystemC.

La classe `sc_simcontext` de SystemC contient cette information et a des méthodes pour accéder à l'information.

Ces méthodes sont :

```
sc_get_curr_simcontext();

sc_simcontext::first_object();

sc_simcontext::next_object();

sc_object::name();

sc_object::kind();
```

Voici un exemple d'utilisation de ces méthodes pour obtenir la liste des modules et l'afficher dans un *popup* menu.

```
sc_clock *clk;
sc_simcontext *sim_context;

sim_context = sc_get_curr_simcontext();

sc_object* module = sim_context->first_object();
```

```
//  MODULE POPUP MENU
while (module)
{
  if (!strcmp(module->kind(), "sc_module"))
  {
    char temp[500];
    strcpy(temp, module->name());
    strcat(temp, "(");
    strcat(temp, module->kind());
    strcat(temp, ")");
    popupMenu[i]->insertItem(temp, this, NULL);
  }
  else
    if (!strcmp(module->kind(), "sc_clock"))
    {
      clk = (sc_clock *) module;
    }
  module =  sim_context->next_object();
}
```

Nous parcourons la liste des sc_object du modèle à simuler, nous vérifions leurs types et nous gardons un pointeur sur chaque objet qui est du type qui nous intéresse.

Tracer les signaux

Même si nous pouvons accéder à la liste des signaux de la même manière que les modules, nous devons aussi avoir accès à leurs valeurs pour les tracer.

Les signaux qui sont connu dans `sc_simcontext` ont eu leurs types transformés (*typecast*) soit en `sc_object` ou en `sc_signal_base`, après quoi, le véritable type de la valeur du signal, qui est *template*, nous est inconnu.

La classe my_sc_observer

Pour lier le signal à notre application, nous créons la classe `my_sc_observer` qui est dérivée de la classe `sc_observer`.

Le constructeur de `my_sc_observer` appelle `sc_observer::bind(sc_observer *derivedInstance);`

De plus, la classe my_sc_observer redéfinit les méthodes virtuelles de update_signal( ) de sc_observer, et ce, pour chaque type de signal possible.

Par exemple :

```
void my_sc_observer::update_signal(const sc_signal_base* signal,
const bool& data) const
{
  if (mainwindow)
    mainwindow->update(signal, data);
}
```

MainWindow::update(…), est une méthode *template* qui appelle signal_info<T>::update(const sc_signal_base * signal, T &signal_value).

Classe signal_info_base

La classe de base de signal_info. Dans cette classe, nous gardons une *map* statique pour nos signaux quels que soient leurs types.

La déclaration de cette classe est comme suit :

```
class signal_info_base
{
protected :

  static MainWindow *mainwindow;

public :

  static map<const sc_signal_base *, signal_info_base *,
             less< const sc_signal_base * > > info_list;
  static void bind(MainWindow *window);
  static void start_trace(const sc_signal_base *);
  static void update_window(const sc_signal_base *,
                            QPainter *painter=0);
  virtual void start_trace(void) = 0;
  virtual void update_window(QPainter * painter = 0) = 0;

};
```

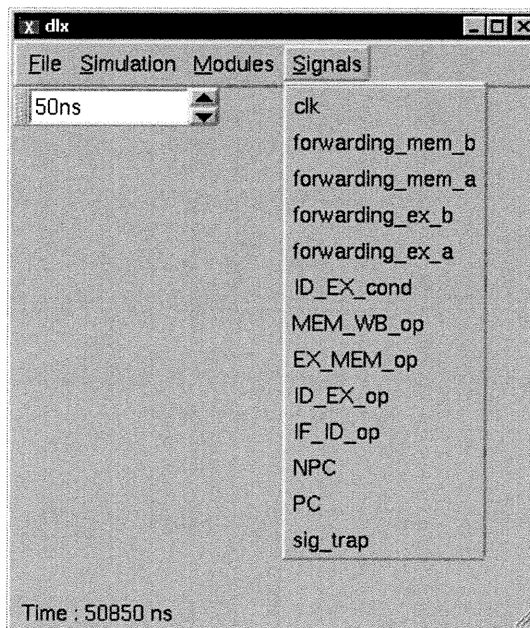
Le membre info_list est une *map* de tous les signaux du modèle, bind(…) est la méthode qui initialise le membre mainwindow.

La méthode start_trace (const sc_signal_base *) va appeler la redéfinition de virtual void start_trace(…) dans la classe dérivée. Une

approche similaire est prise pour `update_window(const sc_signal_base*, QPainter *painter=0)`. Ceci nous permettra de contourner le problème du type de signal qui est *template*.

Classe signal_info

Cette classe est dérivée de signal_info_base, c'est une classe dont le type de donnée est *template*.

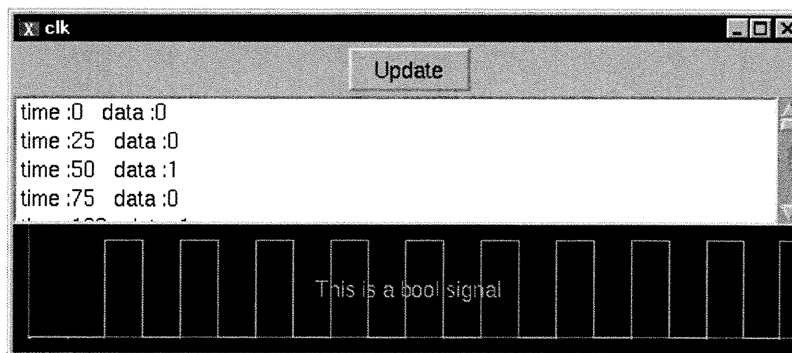


Dans cette classe, nous avons un objet *map*, `data`, qui contient la liste des temps de simulation et des valeurs du signal. Une instance de cet objet est créée pour chaque signal au début de la simulation. Par contre, elle ne contient que la valeur initiale du signal tant que la méthode `start_trace()` n'est pas appelée. Dans notre application, ceci arrive lorsque l'usager choisit le signal dans le menu.

Le autres members sont:

`SignalWindow *window` est un pointeur vers la fenêtre d'affichage de la trace du signal.

Le membre `const sc_signal_based *signal` est un pointeur vers le signal lui-même dont le type est transformé (type cast) en sc_signal_base *.



Un autre membre, `bool tracing` est vrai si l'usager a sélectionné le signal dans le menu et faux sinon.

Nous avons vu précédemment la séquence d'événements qui nous amène à l'appel de la méthode `signal_info<T>::update(const sc_signal_base * signal, T &signal_value)`. Cette méthode est statique, donc, aucune instance de la classe est nécessaire pour pouvoir l'utiliser. Elle recherche un signal dans la *map* `info_list` de la classe `signal_info_base`, si la recherche est vaine, un nouvel objet `signal_info` est créé. Quelque soit le résultat de la recherche, il y a ensuite un appel à `signal_info<T>::update(T &signal_value)`. Cette dernière va initialiser la valeur lors du premier appel et placer chaque nouvelle valeurs du signal dans la *map* `data`, après que le membre `tracing` soit devenu vrai.

Dans le constructeur, nous plaçons l'appel nécessaire pour insérer le signal dans `info_list` ce qui appel `mainwindow->new_signal(...)` qui va lier le signal avec le menu `signals`.

Finalement, la méthode `update_window(...)` appelle la méthode `signal_window::update(...)` pour afficher la valeur dans une fenêtre.

Classe signal_window

Maintenant que l'information est disponible, nous devons l'afficher.

Cette classe contient des *widgets* de QT, un `QVBoxLayout` qui contient un `QlistBox` pour l'affichage textuel du temps et de la valeur du signal ainsi qu'un `Qframe` comme zone de dessin pour le signal

Changements nécessaires dans le modèle

Ici, nous désirons garder les changements simples et peu nombreux.

L'idée est de replacer l'appel à `sc_start( )` par une instance de `mainwindow` dans `main.cpp`.

Premièrement, il faut inclure mainwindow.h.

Ensuite, déclarons un pointeur sur un objet `MainWindow` et sur un Qapplication.

```
MainWindow *mainWindow;

QApplication *application;
```

Finalement, nous replaçons l'appel de `sc_start ( )` par des appels à `CreateMainWindow()` et `application->exec()` :

```
CreateMainWindow(&application, &mainWindow, ac, av);

bool success = application->exec();
```

Ce qui créera et affichera la fenêtre du GUI et démarrera l'application.

Remerciements

Je tiens à remercier mon directeur El Mostapha Aboulhamid et mon co-directeur Guy Bois pour les conseils et le support qu'ils m'ont apportés.

Je remercie également Luc Charest pour l'aide et les explications précieuses fournies sur l'approche orientée objet et les méthodologies de génie logiciel.

Finalement, merci à ma famille ; Josée et mes enfants William et Mylène pour leur patience et leur support.