Université de Montréal

Simulateur compilé d'une description multi-langage des systèmes hétérogènes

par
Mathieu Dubois

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Thèse présentée à la Faculté des arts et des sciences
en vue de l'obtention du grade de Philosophiæ Doctor (Ph.D.)
en informatique

Mars, 2011

Université de Montréal
Faculté des arts et des sciences

Cette thèse intitulée:

Simulateur compilé d'une description multi-langage des systèmes hétérogènes

présentée par:

Mathieu Dubois

La thèse a été évaluée par un jury composé des personnes suivantes:

Guy Lapalme,	président-rapporteur
Michel Boyer,	membre du jury
Houari Sahraoui,	membre du jury
Otmane Ait Mohamed,	examinateur externe
Laurent Lewis,	représentant du doyen F.A.S.

Thèse acceptée le: 7 Juin 2011

RÉSUMÉ

La conception de systèmes hétérogènes exige deux étapes importantes, à savoir : la modélisation et la simulation. Habituellement, des simulateurs sont reliés et synchronisés en employant un bus de co-simulation. Les approches courantes ont beaucoup d'inconvénients : elles ne sont pas toujours adaptées aux environnements distribués, le temps d'exécution de simulation peut être très décevant, et chaque simulateur a son propre noyau de simulation. Nous proposons une nouvelle approche qui consiste au développement d'un simulateur compilé multi-langage où chaque modèle peut être décrit en employant différents langages de modélisation tel que SystemC, ESyS.Net ou autres. Chaque modèle contient généralement des modules et des moyens de communications entre eux. Les modules décrivent des fonctionnalités propres à un système souhaité. Leur description est réalisée en utilisant la programmation orientée objet et peut être décrite en utilisant une syntaxe que l'utilisateur aura choisie. Nous proposons ainsi une séparation entre le langage de modélisation et la simulation. Les modèles sont transformés en une même représentation interne qui pourrait être vue comme ensemble d'objets. Notre environnement compile les objets internes en produisant un code unifié au lieu d'utiliser plusieurs langages de modélisation qui ajoutent beaucoup de mécanismes de communications et des informations supplémentaires. Les optimisations peuvent inclure différents mécanismes tels que le regroupement des processus en un seul processus séquentiel tout en respectant la sémantique des modèles. Nous utiliserons deux niveaux d'abstraction soit le « register transfer level » (RTL) et le « transaction level modeling » (TLM). Le RTL permet une modélisation à bas niveau d'abstraction et la communication entre les modules se fait à l'aide de signaux et des signalisations. Le TLM est une modélisation d'une communication transactionnelle à un plus haut niveau d'abstraction. Notre objectif est de supporter ces deux types de simulation, mais en laissant à l'usager le choix du langage de modélisation. De même, nous proposons d'utiliser un seul noyau au lieu de plusieurs et d'enlever le bus de co-simulation pour accélérer le temps de simulation.

Mots clés: Simulateur, SystemC, co-simulation, multi-langage, systèmes.

ABSTRACT

The design of heterogeneous systems requires two main steps, modeling and simulation. Usually, simulators are connected and synchronized by using a cosimulation bus. These current approaches have many disadvantages: they are not always adapted to the distributed environments, the execution time can be very disappointing, and each simulator has its own core of simulation. We propose a new approach which consists in developing a multi-language compiled simulator where each model can be described by employing various modeling languages such as SystemC, ESyS.Net or others. Each model contains modules and communication links between them. These modules describe functionalities for a desired system. Their description is realized by using the programming object and can be described by using a syntax that a user will have chosen.

We thus propose a separation between the language of modeling and simulation. The models are transformed into the same internal representation which could be seen like unique objects. Our environment compiles these internal objects by producing a unified code instead of using several languages of modeling which add many mechanisms of communications and extra informations. Optimizations can include various mechanisms such as merging processes into only one sequential process while respecting the semantics of the models. We will use two abstraction levels, the "register transfer level"(RTL) and the "transaction-level modeling"(TLM). RTL allows a low level abstraction for modeling and the communication between the modules is done with signals. The TLM is a modeling for transactional communication with a higher abstraction level than RTL. Our aim is to support these two types of simulation, but the user can choose the language of modeling. In the same way, we propose to use a single core and to remove the cosimulation bus to accelerate the simulation time.

Keywords: Simulator, SystemC, cosimulation, multi-laguages, systems.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

NOTATION

TLM	Transaction Level Modeling
RTL	Register Transfer Level
FLI	Foreign Language Interface
PLI	Programming Language Interface
VHDL	High-Level Synthesis Design
CDG	Control Dependence Graph
FDG	Flow Dependence Graph
PDG	Program Dependence Graph
SDG	System Dependence Graph
DLL	Dynamic Link Library
FIFO	First In First OUT
DFS	Depth First Search
XML	Extensible Markup Language
DTD	Document Type Definitions
XSL	Extensible Stylesheet Language
XSLT	Extensible Stylesheet Language Transformations
LALR	Look-Ahead Left Recursive
AFD	Automate Fini Déterministe
ASLD	Abstract Syntax Language Definition
API	Application Programming Interface
CLI	Common Language Infrastructure
CIL	Langage Intermédiaire Commun
BIM	Bus Interface Model
PIM	Pin Interface Model
COM	Component Object Model
CCW	COM Callable Wrapper
RCW	Runtime Callable Wrapper

REMERCIEMENTS

Je tiens tout d'abord à remercier le codirecteur de cette thèse, El Mostapha Aboulhamid, pour son soutien d'encadrement et financier. Il m'a laissé une grande liberté d'action dans cette recherche tout en me guidant vers les bons choix. De plus, j'apprécie grandement l'aide de mon directeur Stefan Monnier. J'aimerais aussi dire merci à Frederic Rousseau du groupe TIMA en France pour ses conseils et son support.

Également, je tiens à remercier tous les étudiants du laboratoire d'analyse et de synthèse des systèmes ordinés (LASSO) pour les échanges d'idées qu'on a eues pendant les réunions.

Enfin, je remercie mes parents et mon frère Martin Dubois pour leur soutien dans ce travail.

CHAPITRE 1

INTRODUCTION

Dans cette dissertation, nous vous montrerons que nous pouvons créer un simulateur supportant efficacement plusieurs langages pour concevoir plus rapidement des systèmes avec une exécution qui est aussi rapide que si nous avions utilisé un seul langage.

La simulation est une technologie pour valider le comportement et la performance des puces ainsi que pour pouvoir rapidement comparer diverses options avant la réalisation physique qui peut prendre plusieurs mois. [Bombana and Bruschi, 2003; Yoo and Jerraya, 2005] et bien d'autres ont développé diverses techniques pour simuler des systèmes composés d'éléments modélisés dans des langages différents. Ces techniques fonctionnent en utilisant un simulateur pour chaque langage, et en connectant les simulateurs. Nous appelons une simulation avec plusieurs simulateurs une co-simulation. Cette co-simulation utilise des mécanismes de synchronisation pouvant augmenter le temps nécessaire pour obtenir un prototype.

Le concepteur de puce utilise des modèles qui sont des descriptions des comportements des composants pour pouvoir simuler un système. Un système peut avoir plusieurs composants et chacun d'eux peut avoir plusieurs modèles différents selon un niveau de détail à simuler. Plus le niveau est élevé, plus le temps de simulation est long.

Dans cette recherche, nous accélérons le simulateur en optimisant et précompilant certaines parties soit la co-simulation et la simulation. Nous éliminerons les mécanismes entre plusieurs langages et précompilerons des modèles plus efficacement.

1.1 Motivation

La simulation permet de valider, de comparer différentes solutions possibles sans avoir la nécessité de construire un prototype. Par contre, certaines applications peuvent nécessiter des jours et des semaines pour être simulées. Par exemple, lorsqu'on veut changer des paramètres d'un système comme le nombre de CPUs et la quantité de

caches, il faut re-simuler et perdre plusieurs mois pour obtenir des nouveaux prototypes.

L'accélération de la simulation est un sujet de recherche très actif, la diminution de la durée d'une simulation d'un système est la clé du succès pour obtenir des prototypes rapidement tout en minimisant leur coût de conception et de fabrication. La mise en marché peut rapidement être effectuée et ainsi avoir des retombées économiques pour la société. Cependant, la simulation homogène ou hétérogène (un ou plusieurs simulateurs) comporte des mécanismes pouvant ralentir la simulation. De plus, concevoir un système simple d'utilisation peut entraîner une dégradation de la vitesse d'exécution. Pour effectuer une simulation, l'industrie offre plusieurs langages pour la conception des logiciels et la modélisation des systèmes. Malheureusement, il n'existe pas un langage unique qui couvre efficacement tout le chemin de la conception partant de l'idée jusqu'à son implémentation [Grant, 2002].

La simulation permet aussi de valider un concept à différents niveaux de réalisation. Par exemple, une équipe de développeurs collabore pour la validation du même système [Dubois et al., 2004]. Au début de la conception, une application logicielle complète pourra être réalisée pour simuler la fonctionnalité voulue. Par la suite, le logiciel se divisera en sous logiciels pour être réparti sur l'ensemble des programmeurs disponibles dans l'équipe. Tout en conservant l'application initiale, les parties logicielles seront de plus en plus raffinées et pourront éventuellement faire partie d'une application finale. Mais, les parties logicielles doivent être simulées avec l'application initiale pour valider l'ensemble de la fonctionnalité [Dubois, 2004; Jerraya and Nicolescu, 2004]. Chaque concepteur aura une partie de la fonctionnalité globale du système à réaliser.

Il faut aussi considérer que 70% du cycle de développement est consacré à la vérification [Kokrady et al., 2006]. Il peut être impossible de vérifier tout le système avec des vecteurs de test dans un délai acceptable. Ce dernier peut être très crucial pour une mise en marché et une mauvaise vérification peut entraîner des coûts de rappel d'un produit. La vérification d'un système avec ses paramètres est réalisée par la simulation. Cette dernière permet de valider les entrées et les sorties d'un système par l'utilisation de vecteurs. L'usager peut vérifier le bon fonctionnement de son application selon ses exigences.

Il existe différents niveaux d'abstraction utilisés pour une simulation d'un système [Bois et al., 2003]. La simulation complète au niveau transfert de registre (RTL) d'un système est trop lente pour une exploration efficace [Kao et al., 2001; Pasricha et al., 2004]. RTL peut représenter un circuit intégré au niveau des registres. Dans le cadre de notre recherche, nous nous concentrons pour diminuer la durée d'une simulation des modèles au niveau de transfert de registre et au niveau transactionnel, permettant ainsi une conception plus rapide. Le niveau transactionnel (TLM : transaction level modeling) est utilisé pour décrire un système à un plus haut niveau d'abstraction que le RTL [Bernstein et al., 2004]. Nous aimerions diminuer les durées de simulations lorsque nous avons plusieurs simulateurs pour la co-simulation d'un niveau transactionnel et RTL.

Typiquement, une plateforme de simulation pour les systèmes est composée de modèles, d'un moteur de simulation et d'outils d'analyses. Elle permet de valider avec un moteur de simulation les choix qu'un concepteur a choisis pour son système. L'exploration architecturale peut être réalisée avec cette plateforme afin de décider les composantes du système à construire. Les modèles représentent les composants du système et les outils d'analyses fournissent des informations sur ceux-ci. Par exemple, un concepteur peut vouloir analyser la bande passante d'un lien (bus de communication) entre des modules. Au début, le bus peut être simulé sans avoir de contention et, au cours du raffinement de celui-ci, on peut ajouter l'information de la latence et de la bande passante au niveau détaillé.

La recherche sur l'accélération de la simulation a permis des solutions pour la conception des systèmes. Elle a réduit la durée de simulation en réalisant des optimisations sur des graphes et en réduisant des appels de fonctions. Cette réduction est possible lorsque nous avons une diminution du détail à simuler. Les coûts de simulation peuvent venir des mécanismes de communication entre des simulateurs par l'utilisation d'entêtes et de protocoles non nécessaires à la simulation. Nous avons aussi le nombre de changements de contexte entre l'exécution des processus et le temps perdu pour recalculer des valeurs plusieurs fois lors de la simulation.

L'interopérabilité des langages comprend trois principaux aspects pour la modélisation des systèmes soit :

- Le niveau d'abstraction :

  Chaque langage a sa force et sa faiblesse, dépendant du niveau d'abstraction [Grant, 2002]. Le concepteur utilisera le meilleur langage pour le niveau de modélisation souhaité. Les systèmes sont décrits en utilisant des langages de modélisation entre les composants d'un même système permettant d'exprimer la fonctionnalité souhaitée. L'interopérabilité entre les langages pour co-simuler ces modèles est réalisée composant-par-composant avec une capacité d'échanger des données.

- La sémantique des langages hôtes comme C# et C++ :

  La durée de simulation est affectée par l'ajout des mécanismes d'échanges et de synchronisations entre les langages. La capacité d'un langage hôte à communiquer avec d'autres est cruciale pour une modélisation hétérogène. Dans les langages qui nous intéressent, nous avons toujours un point commun d'échange pour deux langages choisis.

  Lors du raffinement du système de haut à bas niveau en utilisant plusieurs hôtes, il est très important de pouvoir simuler un système au complet où certains éléments sont modélisés à très bas niveau et d'autres à très haut niveau. Le haut niveau permet d'avoir une simplicité pour l'usager en décrivant son système avec moins de détails, tandis que le bas niveau possède le plus grand nombre de détails. La simulation de celui-ci est souvent hors de prix lorsqu'on simule l'ensemble et exige peut-être des jours de simulation. Certains composants ne seront jamais modélisés à bas niveau comme les bancs d'essai, les stimulus externes au système, car ceux-ci permettent uniquement de valider la simulation.

- Les noyaux (simulateurs) :

  Les noyaux de simulation sont des simulateurs. Ils décrivent des méthodes de calcul décrivant le comportement d'une simulation. Ils doivent communiquer et s'échanger de l'information pour se synchroniser. Les mécanismes doivent être

communs entre eux et permettre l'échange de données.

La co-simulation est une technique permettant à deux simulateurs ou plus d'exécuter une modélisation conjointe d'un système. Son coût est lié aux mécanismes d'interopérabilité. Elle permet une plus grande flexibilité pour la réutilisation des modèles hétérogènes. Un système peut être simulé avec différents modèles décrits dans plusieurs niveaux d'abstraction. Chaque modèle peut être raffiné individuellement et simulé avec tout son système. La simulation et la co-simulation peuvent être accélérées par l'utilisation des grappes, mais la parallélisation est bien sûr non-triviale et est accompagnée de ses propres coûts.

1.2 Méthodologie

Dans notre méthodologie de recherche, nous commencerons à étudier l'impact des mécanismes de co-simulation. Nous étudierons les méthodes de compilations pour les systèmes et d'interopérabilité entre les langages. De plus, nous examinerons la possibilité d'avoir un noyau de simulation qui fonctionne avec plusieurs langages. La séparation entre la vue de l'utilisateur et le simulateur doit être examinée. La vue est ce que l'utilisateur décrit avec les langages hôtes et la vue du simulateur est celle qui permet d'exécuter la simulation.

L'analyse de différents mécanismes de co-simulation pour enlever les mécanismes entre des simulateurs fournira l'information nécessaire pour construire une plateforme de simulation commune. Cette dernière devrait être plus rapide que les environnements standards, car il y a une élimination des mécanismes de co-simulation. Il sera possible de traduire des modèles hétérogènes vers un langage commun pour réduire les appels de fonctions entre plusieurs langages. Nous évaluerons les simulateurs compilés avec divers niveaux d'abstractions telles que transactionnelles et RTL. La simulation peut se faire en compilant les modèles des langages sources de l'utilisateur vers le langage de prédilection du simulateur.

1.3 Techniques principales proposées

Les techniques proposées sont soit nouvelles ou bien modifiées des techniques habituelles. Nous discuterons des mécanismes d'interopérabilité, la représentation homogène des langages hôtes, la génération de code et la compilation par zone.

- Mécanismes d'interopérabilité :

  Nous allons considérer quatre principaux mécanismes d'interopérabilité soit TCP/IP, la mémoire partagée, le « Component Object Model » COM et Pinvoke. Ces techniques permettent une communication entre divers langages. La première et la deuxième sont déjà utilisées pour la simulation des systèmes.

  La technique Pinvoke est utilisée pour appeler des bibliothèques dynamiques lors de l'exécution du code et elle est compilée dans un même binaire. Elle est le nom donné à une méthode d'appel de bibliothèque dynamique en .NET que nous avons choisi d'utiliser pour ce travail. Le COM est utilisé comme intermédiaire pour échanger des données entre deux langages hôtes. Nous utiliserons ces techniques dans une plateforme de simulation hétérogène.

  Nous proposons aussi deux autres techniques que nous nommons dans le cadre de cette recherche : l'adaptateur géré et la fonction statique. Ce sont des techniques pour être compilées dans un même binaire comme le Pinvoke.

- Représentation homogène des langages hôtes :

  Nous voulons optimiser la co-simulation pour la modélisation des systèmes. Notre représentation interne correspond à un sous-ensemble commun de tous les langages hôtes supportés. Par exemple, une classe qui représente un module d'un système sera représentée de manière interne par un noeud de même type « module » quel que soit le langage d'entrée. Nous compilons un sous-ensemble de chaque langage d'entrée qui représente les points communs pour la modélisation d'un système. Par exemple, nous avons des modules, des canaux de communications, des

interfaces et d'autres éléments. Nous pouvons aussi conserver les fonctionnalités spécifiques à un langage pourvu qu'elles n'empêchent pas l'interopérabilité.

- Génération de code :

  Normalement, la simulation s'effectue dans un seul langage hôte. Nous proposons une exécution de la simulation dans plusieurs langages hôtes. Le choix de ce langage dépend de la vitesse d'exécution du pseudo-code compilé que procure ce dernier. Notre compilateur peut générer du code dans n'importe quel langage supporté permettant une exécution qui est la plus rapide.

- Compilation par zone :

  Chaque modèle qui a des composants comprend des processus pour décrire un système. Dans un système complet, les processus doivent être exécutés pour simuler un comportement et peuvent nécessiter plusieurs langages hôtes. Nous proposons de regrouper chaque processus dans un seul pour pouvoir les simuler. Nous voulons ainsi optimiser les interactions entre plusieurs langages hôtes, car il n'y aura plus de mécanismes de co-simulation. Les processus sont mis dans des zones que nous aurons prédéfinies. De plus, nous ajouterons un ordonnanceur dans le code compilé pour définir l'ordre d'exécution des zones lors de la simulation.

1.4 Contributions

L'objectif de ce travail est la création d'un simulateur compilé pour des systèmes hétérogènes. Nous voulons modéliser des systèmes décrits dans plusieurs langages. Les langages hôtes considérés pour cette recherche sont SystemC et ESyS.Net [Lapalme et al., 2004; Schlebusch, 2000]. Les langages générés sont C++, C#, VHDL et Java.

- Adaptation des mécanismes de co-simulation pour une modélisation plus flexible :

Trois principaux mécanismes d'interopérabilité sont utilisés dans la littérature soit la mémoire partagée, l'intégration dans le même binaire et le passage de message. Nous adaptons les mécanismes des fonctions statiques, du pinvoke, du COM et de l'adaptateur géré pour effectuer une co-simulation plus flexible pour l'usager avec SystemC et ESyS.Net. Nous montrerons qu'il est possible avec l'adaptateur géré de relier directement différents types de données (nécessaires à la simulation) provenant de différents simulateurs ensemble sans l'utilisation de fonctions intermédiaires qui sont normalement utilisées.

- Exécution hétérogène sans l'utilisation de mécanismes d'interopérabilité tout en ayant une rapidité comme si nous avions utilisé un seul langage :

  Dans la simulation hétérogène, des simulateurs sont utilisés sans être capables de supporter plusieurs langages hôtes pour la conception des systèmes. Nous montrerons comment simuler un système composé d'éléments décrits dans des langages différents, en les compilant dans une représentation unifiée, donc sans recourir aux techniques traditionnelles de co-simulaont inefficaces. Par exemple, la technique de la mémoire partagée est au moins 5 fois plus lente qu'une simulation utilisant qu'un seul langage [Dubois et al., 2007]. D'autres exemples sont aussi montrés à la section 5.3 avec un même facteur.

  Cette méthode permet de regrouper des processus et des méthodes des modèles. Nous créerons une représentation interne pour l'accélération de la simulation. En utilisant un noyau de simulation unique, il est possible d'éliminer les mécanismes de co-simulation entre deux simulateurs qui augmentent la durée de la simulation. Nous proposerons des techniques de minimisation des communications avec la possibilité d'utiliser plusieurs langages. Normalement, les méthodes de compilation se limitent à leurs langages respectifs sans permettre une minimisation entre plusieurs langages. Le coût et le nombre des appels des fonctions sont minimisés dans leur langage respectif incluant leurs interactions avec les autres langages né-

cessaires à la simulation complète d'un système. Les appels de fonctions peuvent être des interfaces utilisées par des mécanismes d'interopérabilité entre deux langages. De plus, les appels nécessaires à la modélisation pour la gestion des évènements et des processus sont minimisés.

- Modification d'un simulateur pour qu'il supporte un autre langage :

  Nous avons modifié le simulateur ESyS.Net pour qu'il supporte une variante de SystemC. L'objectif est d'avoir un noyau de simulation commun pour les langages les plus répandus.

- Une séparation entre le langage et le modèle permettant une accélération de la simulation :

  En prenant avantage du fait que les modèles que nous compilons n'utilisent qu'un sous ensemble de C#/Java nous pouvons les compiler de manière plus efficace que les compilateurs C#/Java actuels. Plutôt que d'écrire un compilateur complet, nous traduisons le code C# vers C++ de manière à laisser le gros du travail aux compilateurs C++ existants. La compilation ciblée permet de compiler le code dans un format qui s'exécute le plus rapidement que possible tout en conservant le langage de modélisation choisi par l'usager.

- Traduction d'un langage à un autre :

  Une fonctionnalité supplémentaire de ce travail est la capacité du compilateur de traduire les langages supportés d'un langage à un autre. Par exemple, cette dernière permet d'utiliser des composants ESyS.Net avec des composants SystemC.

1.5 Organisation de la présentation

Le chapitre 2 fait l'état de l'art de la modélisation et de la simulation. Les méthodes d'interconnexions des langages, ainsi qu'un survol des moyens de synchronisation entre

deux noyaux sont présentés. Par la suite, dans le chapitre 3 nous discuterons de quatre techniques de co-simulation basé sur des mécanismes d'interaction entre les langages. Nous allons comparer nos méthodes avec ceux qui sont largement utilisées pour une co-simulation soit le TCP/IP et la mémoire partagée. De plus, nous présenterons une technique pour éliminer un noyau de simulation dans une co-simulation pour obtenir un seul simulateur supportant plus qu'un langage. Le chapitre 4 présente une nouvelle méthodologie de compilation permettant une simulation compilée. La méthode consistera en trois étapes principales soit la génération XML, la génération de code, l'analyse et le traitement. Nous présenterons la méthode à deux niveaux d'abstraction soit transactionnelle et registre. Pour l'illustration de la nouvelle méthode, nous utiliserons un exemple de premier arrivé, premier servi (FIFO). De plus, un exemple avec un producteur et plusieurs consommateurs sera décrit. Nous comparerons ainsi différents aspects de simulation. Le chapitre 5 présente les résultats ainsi que des exemples du RTL et du TLM compilé. Nous terminerons par une conclusion dans le dernier chapitre.

CHAPITRE 2

ÉTAT DE L'ART : MODÉLISATION ET SIMULATION

Dans ce chapitre, nous allons présenter les aspects d'une modélisation, d'une simulation, d'un survol des techniques existantes pour accélérer une simulation et une co-simulation. Nous regarderons les langages de modélisation les plus populaires avec leur paradigme, ainsi que des techniques qui ont été utilisées pour diminuer la durée de simulation. Les mécanismes de co-simulation comme la compilation dans le même binaire, la méthode de connexion par mémoire partagée et la méthode de connexion par lien TCP/IP seront présentés. La comparaison des langages de modélisation existants dans ce chapitre permet une revue sur les méthodologies pour simuler des systèmes. Cette analyse montre aussi l'interaction entre les noyaux et leurs mécanismes de synchronisation. De plus, certaines approches ont permis de diminuer la durée d'une simulation en utilisant un ordonnancement statique.

2.1 Paradigmes de modélisation

Dans cette section, nous discuterons principalement de trois paradigmes de modélisation : les langages fonctionnels, les langages synchrones et l'électronique numérique.

Les langages fonctionnels peuvent être Erlang, IIRC et Lava basé sur Haskell [Bjesse et al., 1998; Robert et al., 1996]. Le paradigme fonctionnel est basé sur l'usage de fonctions sans effets de bord. Un de ses représentants les plus important dans notre domaine est Erlang. Les fonctions sont vues comme des acteurs échangeant des messages entre eux. Chaque acteur est une machine à états effectuant un calcul concurrent. À chaque message reçu, un acteur effectue un traitement et envoie d'autres messages.

Une des techniques optimisations utilisée sur les processus (acteurs) permet de réduire les coût de communication et créer plus d'opportunités d'optimisation, au même titre que l'inlining [Stenman and Sagonas, 2002]. Le processus effectuant le traitement d'un message reçu est copié dans celui qui l'a appelé. Par contre, l'analyse statique re-

quise peut être difficile à effectuer pour établir le lien entre les processus appelants et appelés. Une autre méthode est d'avoir une mémoire partagée entre les processus évitant ainsi de copier les messages en mémoire.

Les langages synchrones ont été créés pour des applications « réactives ». Pour illustrer son fonctionnement, nous pouvons parler d'Esterel [Zaffalon, 2005]. Ces langages ont des interfaces, des communications et des évènements permettant une modélisation des systèmes réactifs réagissent en fonction de leur environnement. Par exemple, un système peut avoir des actionneurs, une sortie et des capteurs en entrée. Un actionneur peut être le contrôle d'une valve d'un réservoir et le capteur peut mesurer le niveau de liquide. Les interfaces d'entrées et de sorties sont les liens entre le système et son environnement. Dans notre exemple, le système peut être modélisé par un processus qui vérifie le remplissage ou le vidange du réservoir. Selon le niveau du réservoir, un message est envoyé pour le vider ou le remplir. L'ordonnancement des processus est synchronisé avec une horloge globale qui avance d'un instant à l'autre selon les réactions. À chaque instant, les processus peuvent être créés, détruits ou exécutés.

Les applications ciblées initialement étaient des systèmes d'instrumentation et contrôle, mais de nos jours Esterel est utilisé pour concevoir des systèmes [Berry et al., 2002]. Une simulation avec Esterel peut être effectuée à différents niveaux d'abstraction comme le TLM et le RTL. Le paradigme de simulation pour l'instrumentation et de contrôle qui utilise des automates peut aussi modéliser des portes logiques [Zaffalon, 2005].

Les langages pour l'électronique numérique comme Verilog, SystemC et VHDL permettent aussi une description matérielle d'un circuit logique électronique [Martin, 2003; Palnitkar, 2003; Perry, 1998]. Ils permettent une description fonctionnelle avec des schémas logiques pouvant être mis dans une puce. Ce sont des langages pour la conception d'ASICs « Application Specific Integrated Circuits » et de FPGAs « Field Programmable Gate Arrays » [Curtin, 1990; Knack, 1994; Olsen and McDermith, 1993]. L'utilisateur peut modéliser des processus concurrents et avoir des signaux entre eux. De plus, nous avons principalement des ports de communications et des entités. Les entités sont des modules qui permettent une communication avec le monde extérieur. Elles consistent

en une description structurelle précise de l'architecture matérielle à utiliser et leurs interconnexions. À chaque entité, nous déclarons une implémentation sous forme d'une architecture. Des ports de communications sont décrits pour chaque entité (module) permettant une communication entre eux. Chaque architecture possède des processus qui décrivent une fonctionnalité.

Les langages pour l'électronique numérique permettent de modéliser un système aux niveaux détaillés (RTL) en fonction des horloges, des entrées et des sorties. Ils peuvent aussi supporter d'autres niveaux d'abstraction. Ils sont basés principalement sur des techniques de simulation événementielle, tandis qu'Esterel est basé sur des machines à états finis. De plus, Esterel ne permet pas d'avoir des effets de bord. Les communications entre les sous-systèmes sont diffusées dans Esterel, contrairement aux langages VHDL, Verilog et SystemC.

Dans un système, une modélisation de haut niveau entre deux modules fait que les deux peuvent très bien communiquer en utilisant des messages comme les langages fonctionnels. Le langage synchrone est mieux adapté pour le niveau RTL que le niveau TLM, tandis que les langages fonctionnels sont plus appropriés pour une modélisation transactionnelle que RTL. La durée de simulation d'une modélisation TLM est plus courte que le RTL, mais comprend moins de détails.

RTL est une simulation lente, car elle est au niveau des portes logiques. Elle permet un raffinement aux dernières étapes de conception dans un circuit intégré. Au niveau TLM, nous pouvons modéliser des interfaces abstraites entre des modules pour créer des canaux de communication [Hatami et al., 2009]. Les ports des modules sont décrits à l'aide d'interfaces. Une interface est simplement une déclaration des fonctions qui pourront être utilisées à travers des ports d'un module. L'implémentation se fait à l'intérieur des canaux. Ces derniers sont les moyens de communication entre les modules et sont branchés par les ports des modules. Les processus peuvent aussi communiquer directement avec les canaux en appelant directement les interfaces du canal.

2.2 Détails et coûts de la simulation

Certaines recherches proposent d'augmenter le niveau d'abstraction de la communication entre les modules et de permettre à l'usager d'éviter le détail des communications réduisant ainsi le coût de la simulation détaillée [Xinping and Malik, 2002]. Cependant, à ce niveau, l'usager n'a pas toutes les informations pour sa réalisation finale et il peut les ignorer.

Les langages fonctionnels permettent des communications entre les modules en utilisant uniquement des messages. Par contre, ceux-ci ne donnent pas à l'usager l'implémentation des canaux, ce qui est primordial pour modéliser un canal physique tel qu'un bus de communication. Dans les langages fonctionnels, [Robert et al., 1996] présente différentes techniques pour diminuer la durée de simulation pour le paradigme d'échange de messages entre processus. Dans le cadre de l'exploration d'une architecture d'un système, il est important d'avoir un compromis entre la durée de conception et le niveau de détails simulés. Dans le domaine RTL, des recherches ont aussi permis d'accélérer la simulation. La modélisation du circuit est séparée en deux parties principales : circuits combinatoires et circuits avec horloge [Breuer et al., 1994]. Des optimisations sur la première partie ont été faites en utilisant une compilation statique [Kupriyanov et al., 2004]. C'est-à-dire qu'il y a un ordonnancement statique qui est effectué pendant la compilation. Les éléments combinatoires sont ordonnancés de manière à être exécutés selon un ordre prédéfini. Les techniques de simulation peuvent être optimisées pour augmenter l'efficacité, mais la flexibilité a toujours un coût pour l'usager.

2.3 Modélisation et co-simulation

Une exploration rapide de plusieurs architectures est réalisée grâce aux langages de modélisation illustrés à la figure 2.1. Cette exploration permet d'avoir une description d'un système grâce aux canaux et aux modules communicants avec des interfaces de haut niveau sans se soucier des détails d'implémentations.

Une plateforme de simulation comprend quatre aspects principaux soit : un modèle, un noyau de simulation, un langage de programmation et des niveaux d'abstraction.

À partir du langage de programmation, un modèle peut être écrit et simulé par un noyau de simulation. Le modèle est une description d'une fonctionnalité que l'usager veut décrire comme un circuit électrique. Celui-ci peut être décrit en utilisant des modules, des communications, des canaux, des signaux, des évènements, des primitives et autres.

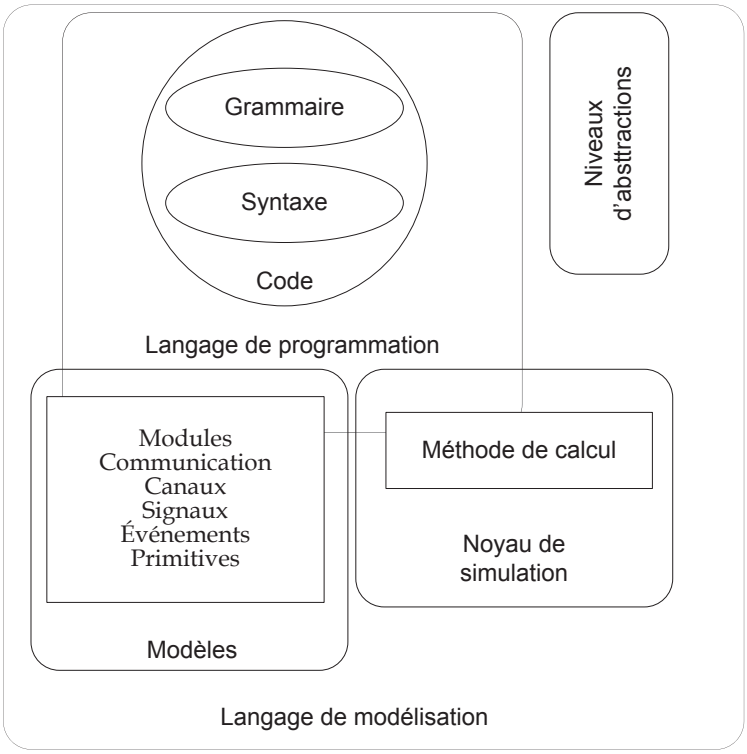


Figure 2.1 – Langage de modélisation

La co-simulation a deux approches [Nicolescu and Gabriela, 2002] soit l'approche à langage unique [Calvez et al., 1996; Pasquier and Calvez, 1999] et l'approche distribuée. La première approche est celle permettant de traduire une spécification multi-langage dans un langage unique pour pouvoir le simuler et l'autre approche est une simulation conjointe entre deux simulateurs.

Une co-simulation [Nicolescu and Gabriela, 2002] est composée principalement de trois éléments principaux : des simulateurs, des interfaces de co-simulation et un bus de co-simulation. Les simulateurs permet l'exécution des modèles par différents simula-

teurs. Les interfaces sont le lien de communication entre deux ou plusieurs simulateurs. Le bus permet une synchronisation entre les simulateurs.

2.3.1 Réutilisation des modèles

La réalisation d'un système nécessite une méthodologie pour éviter de réécrire les mêmes fonctionnalités à chaque conception d'une architecture. Chaque module matériel ou logiciel peut être encapsulé sous la forme d'un module réutilisable qui sont souvent appelés « propriété intellectuelle (IP) ». Ce dernier est un composant créé par une personne ou une compagnie avec ses propres caractéristiques.

Il peut être un module permettant une communication avec un périphérique de mémoire ou bien un sous-système. La figure 2.2 présente un schéma bloc d'un environnement de conception des systèmes basé sur la réutilisation des modèles [Lennard et al., 2006].

Une bibliothèque IP-XACT a été créé par l'industrie pour faciliter l'intégration des modules provenant de différents fournisseurs. La bibliothèque est composée de trois éléments. Le premier est une métadonnée pouvant être représentée en XML [Elliotte Rusty and W. Scott, 2000]. Une métadonnée est tout simplement une donnée servant à définir une autre donnée. Deuxièmement, nous avons deux types de générateurs soit pour les interconnexions et la génération de la configuration. Et pour terminer, nous avons la vue des modèles.

Dans l'exemple du code 2.1 , « MaLibrairie » a un module avec un bus d'interface ayant un signal « irq ». La spécification IP-XACT est un standard pour décrire les modèles [Kwanghyun et al., 2008]. Elle permet de décrire les caractéristiques d'un modèle

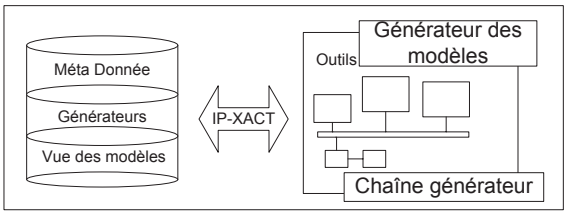


Figure 2.2 – Environnement de conception SoC

Listing 2.1 – Spirit Exemple

```
1  <spirit:component xmlns:spirit="http://www.spiritconsortium.org/
       XMLSchema/SPIRIT/1.0"xmlns:xsi="http://www.w3.org/2001/
       XMLSchema-instance"xsi:schemaLocation="http://www.
       spiritconsortium.org/XMLSchema/SPIRIT/1.0">
2      <spirit:library> MaLibrairie </spirit:library>
3      <spirit:name> Module </spirit:name>
4      <spirit:version>1.0</spirit:version>
5      <spirit:busInterface>
6          <spirit:name> Interrupt </spirit:name>
7          <spirit:busTypespirit:vendor="Spirit"spirit:library=" MaLibrarie "
               spirit:name="Int"/>
8          <spirit:master/>
9          <spirit:signalMap>
10             <spirit:signalNamespirit:busSignal="IRQ"> irq </spirit:signalName>
11         </spirit:signalMap>
12     </spirit:busInterface>
13 </spirit:component>
```

selon un guide prédéterminé. Par exemple, l'interface, les entrées et les sorties d'un modèle peuvent être définies. Cette manière de procéder permet d'échanger des modèles entre des personnes ou des compagnies et d'avoir un langage neutre. De plus, les outils supportant la norme IP-XACT sont compatibles.

Par contre, cette bibliothèque ne contient pas toute l'information du modèle dans un format commun à tous. Par exemple, nous pouvons définir un modèle en VHDL et un autre en SystemC, mais il n'y a pas une représentation interne au complet avec le code source. Nous avons simplement un pointeur vers un fichier SystemC et un fichier VHDL. Le point commun est la documentation du modèle en XML. C'est-à-dire une représentation commune de la spécification d'un module pour documenter sa spécification pouvant être utilisé par plusieurs outils différents. Cette spécification comprend des informations de conception permettant aux outils logiciels de configurer et d'intégrer automatiquement un modèle dans un design. IP-XACT décrit un module au niveau des entrées et des sorties avec sa caractérisation.

2.3.2 Présentation des modèles de calcul

La clé du succès pour construire un noyau de simulation est de définir des modèles de calcul. Les modèles de calcul sont un des trois domaines de types primaires selon le standard Rosetta [Alexander, 2009]. Chaque modèle de calcul permet une simulation pour une plage d'applications. Il est utile de connaître les différentes catégories pour être en mesure d'avoir la bonne sémantique de simulation pour une application donnée. La figure 2.3 présente les domaines de Rosetta. Nous avons l'état continu, l'état fini et l'état infini pour le modèle de calcul.

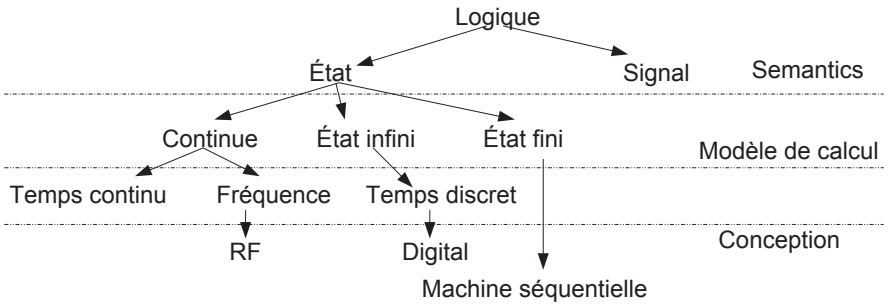


Figure 2.3 – Les domaines de Rosetta

Les domaines permettent de définir le type de simulation qu'il faut exécuter selon l'application qui a été modélisée. Par exemple, un dispositif de télécommunication peut avoir différents modèles. Un signal peut être reçu en utilisant une antenne et être amplifié pour être converti d'analogique à numérique. Il doit être modélisé avec au moins trois domaines : la fréquence, le temps discret et le domaine de puissance. L'interaction entre les niveaux d'abstractions (domaines) est réalisée en employant le traducteur et les fonctions blocs de Rosetta. Les traducteurs sont employés pour déplacer des données entre les domaines en employant des interfaces. Les fonctions définissent des mécanismes pour les modèles mobiles entre les domaines de spécifications. Les définitions des fonctions jouent deux rôles importants dans la modélisation de Rosetta. D'abord, elles sont employées pour déplacer l'information entre les domaines pour exécuter l'analyse. En second lieu, des fonctions sont employées pour déplacer l'information entre les domaines

pour changer la modélisation à un niveau d'abstraction différent. Chaque modèle dans un domaine spécifique doit être décrit pour être exécuté par un modèle de calcul spécifique.

La section suivante présente des modèles de calcul avec leur langage de modélisation permettant d'établir des comparaisons entre des noyaux.

2.4 Descriptions des modèles et des simulateurs

Dans cette section, nous discuterons de VHDL, Verilog, SystemVerilog, SystemC et ESyS.Net.

2.4.1 VHDL

VHDL est utilisé pour la synthèse des systèmes sur FPGA ou ASIC. Il est très souvent utilisé pour le niveau RTL. À ce niveau, le concepteur peut obtenir des informations de vitesse et de puissance par rapport à une architecture ciblée. Également, VHDL peut être utilisé pour un niveau d'abstraction plus élevé.

Une simulation d'un circuit numérique implique un modèle de temps discret, tel qu'illustré à la figure 2.4, pour simuler son comportement.

Le temps discret [Cataldo et al., 2005; Semeria and Ghosh, 2000] permet un avancement d'une simulation au niveau du temps. Par exemple, T = 0 ns augmentera à T = 1 ns, 2 ns et ainsi de suite. L'axe cause à effet (causal) n'influence pas le temps discret et permet un ordonnancement (relation de préséance) pour des évènements qui sont cau-

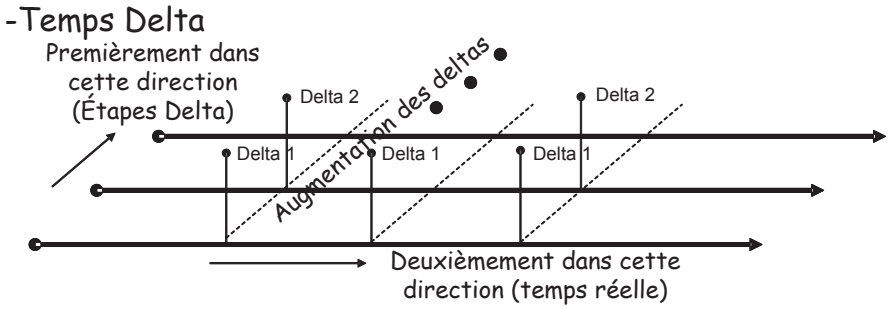


Figure 2.4 – Modèle de Temps

sals, mais paraissant simultanés. À chaque temps T, l'axe causal permet d'exécuter tous les signaux qui doivent changer de valeur jusqu'à ce qu'il n'y ait aucun autre changement. À chaque fois que l'axe causal avance, les signaux sont recalculés s'il y a eu des évènements qui les modifient au temps T.

Nous pouvons dire que la simulation est cause à effet. C'est-à-dire qu'à chaque fois qu'un signal change (cause), le signal émet un événement (effet) permettant d'exécuter les signaux qui sont dépendants de lui. L'ordre des processus est défini dynamiquement lors de l'exécution. Il existe des méthodes d'ordonnancements statiques pour des applications ayant un bas niveau d'abstraction [Baresi et al., 1996].

La description de l'interface d'un module en VHDL est illustrée au code 2.2.

Listing 2.2 – Module en VHDL

```
1  entity Module is
2  port(
3      Output1 : out std_logic;
4      Output2 : out std_logic;
5      Input1 : in std_logic;
6      Input2 : in std_logic
7  );
8  end entity Module;
```

Chaque description se compose d'au moins une entité/architecture, ou une entité avec des architectures multiples. La section « entity » est utilisée pour déclarer les ports d'entrée-sortie du circuit, alors que le code de description réside dans la partie architecture. Dans notre exemple, une entité nommée « module » a deux entrées et deux sorties. Des bibliothèques normalisées de conception sont typiquement employées et sont incluses avant la déclaration d'entité. L'utilisation du nom « port » permet de définir les entrées et les sorties d'un circuit.

Nous avons des signaux et des variables dans la conception des circuits. Un signal est utilisé pour connecter les composants du circuit ensemble et transporter l'information entre les expressions du circuit. Les variables sont utilisées à l'intérieur des processus pour le calcul d'une certaine valeur.

En VHDL, il y a deux styles de comportement : concurrent et séquentiel. Les expressions concurrentes sont décrites dans le corps de l'architecture. Celles-ci manipulent des

signaux, des processus concurrents et des instances des composantes (ports). Les expressions séquentielles incluent les expressions cases, si alors, les boucles, affectation de variables et des signaux. Les processus séquentiels et combinatoires sont illustrés au code 2.3.

Listing 2.3 – Processus en VHDL

```
1  bascule : PROCESS (clk, reset)
2  BEGIN
3   IF reset = '1' THEN
4       q <= 0;
5   ELSE
6       IF clk'event AND clk = '1' THEN
7           q <= d;
8       END IF;
9   END IF;
10 END bascule;
11 c <= a and b;
```

Le processus bascule (code 2.3) contient les signaux « clk » et « reset ». Le processus est séquentiel et il sera exécuté à chaque fois que « reset » et « clk » changeront. Dans ce cas-ci, « reset » est asynchrone, car il ne dépend pas de l'horloge. Le processus combinatoire tel que « c <= a and b » change à chaque fois que « b » ou « a » change.

Lors de la conception des circuits séquentiels, un important modèle est utilisé : les machines à états finis (FSM). Un modèle FSM contient de la logique séquentielle et combinatoire telle que les états des registres dans lesquels on enregistre l'état du circuit et qu'on fait des mises à jour sur les fronts de l'horloge. Les états des sorties dépendent de la fonction qui est calculée dans le FSM.

Lors de la conception RTL, nous nous intéressons au VHDL synthétisable permettant d'obtenir un circuit au niveau des portes logiques. Le circuit est décrit comme un ensemble de registres et de fonctions permettant le transfert entre le chemin de données et les registres.

2.4.2 Verilog

Verilog est utilisé pour la synthèse des systèmes comme VHDL sur FPGA ou ASIC [Palnitkar, 2003]. La méthode simplifiée de sa simulation est illustrée à la figure 2.5.

VHDL avance son temps avec une liste, tandis que Verilog a une liste par temps T0, T1 et TN.

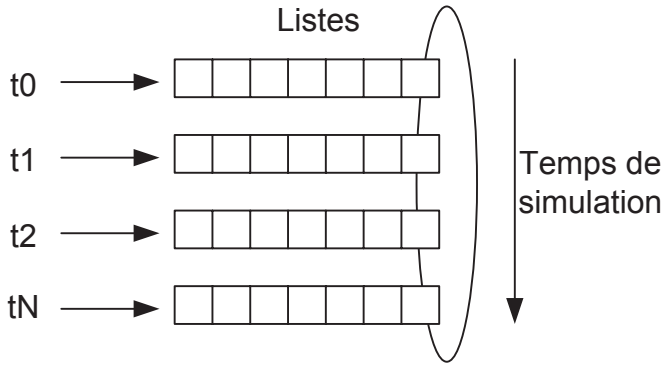


Figure 2.5 – Modèle simplifié d'une simulation Verilog

Chaque évènement est placé dans des listes correspondant à un temps donné. Un évènement est émis par un signal Verilog et ce dernier est placé dans la liste TN selon le temps qui sera exécuté. Par exemple, un évènement émis au T0 pour une exécution à T2 sera placé dans la liste T2. Lorsque le simulateur Verilog avance son temps discret, il prend la liste correspondante à un instant T pour l'exécuter. Dans l'exemple du code 2.4, nous avons un module « add » avec 3 entrées et 2 sorties.

Listing 2.4 – Module en Verilog

```
1  module add (
2        a , // Premième entrée
3        b , // Seconde entrée
4        ci , // L'entrée de retenue
5        sum ,// Le résultat de la somme
6        co // Le reste de la somme
7  );
8  // Déclaration d'entrée
9  input a; input b; input ci;
10 // Déclaration de sortie
11 output sum; output co;
12 // Types de ports
13 wire a; wire b; wire ci; wire sum; wire co;
14 ...
15 endmodule // Fin du module
```

L'opération addition peut se faire avec les entrées « a »,« b », « ci », la retenue « ci » précédente et les sorties « sum » et « co » sont respectivement la somme et la propagation de la retenue vers un étage ultérieur. C'est un exemple combinatoire, mais Verilog permet aussi d'avoir des processus séquentiels.

Listing 2.5 – Processus en Verilog

```
1 Processus séquentiel
2       always @ ( posedge clk )
3       q <= d;
4 Processus combinatoire
5       always @ ( a or b )
6       c = a and b;
```

Les processus séquentiels sont exécutés sur un évènement (clk) et la logique combinatoire sur un changement des entrées (a, b). Verilog permet une modélisation de bas niveau. La prochaine section traitera du langage SystemVerilog.

2.4.3 SystemVerilog

SystemVerilog est un niveau supérieur de verilog-2005 pour la vérification et la modélisation [Chris, 2008]. C'est un langage de vérification de haut niveau permettant l'augmentation de la productivité de la conception des circuits intégrés à large échelle. SystemVerilog possède des types tels que les sémaphores et les mécanismes de synchronisation. La simulation de celui-ci est divisée en cinq régions. Chaque région est exécutée dans un ordre précis tel qu'illustré à la figure 2.6.

La simulation se divise en cinq régions : active, inactive, non bloquante, observée et réactive. La région active gère des évènements tels que des assignations bloquantes et continues, des primitives d'entrées et de sorties et des événements d'évaluation de la sensibilité des processus. La sensibilité des processus est une liste d'évènements pouvant faire exécuter un processus. Les assignations bloquantes sont utilisées pour la modélisation des parties combinatoires et les assignations continues permettent la modélisation des parties séquentielles. Les assignations permettent de contrôler l'exécution des processus selon leur évènement qu'ils reçoivent. Dans le cas d'une assignation bloquante, le processus continue de s'exécuter lorsqu'il est débloqué par un évènement qu'il attendait.

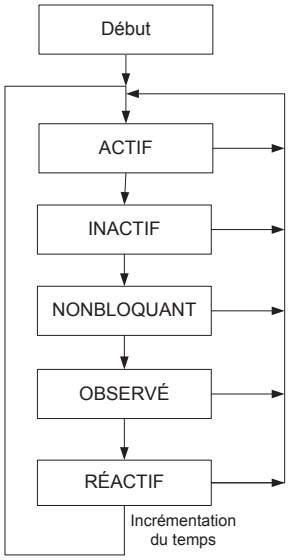


Figure 2.6 – Simulation de SystemVerilog

SystemVerilog permet des évènements d'évaluations, de mise à jour et d'évaluations de la sensibilité lors d'une simulation. Tant qu'il y aura des changements sur les expressions à exécuter, il y aura un recalcul des expressions jusqu'à ce qu'il n'y reste aucun évènement. La région inactive permet un ordonnancement des évènements devant être exécutés après ceux de la région action. La région non bloquante est utilisée pour les affectations non bloquantes.

La région observée permet la vérification avec des évènements d'affirmations. Cette région est très pratique pour déboguer un système. Elle permet une visualisation des évaluations de test à l'usager et elle est exécutée dans la queue observée. La dernière région est la région réactive ajoutée pour SystemVerilog par rapport à Verilog pour obtenir une compatibilité.

2.4.4 SystemC

SystemC est un noyau de simulation très répandu écrit en C++ . Nous pouvons l'utiliser pour une modélisation à haut niveau TLM et à bas niveau RTL. La figure 2.7 représente une structure typique d'une modélisation d'un système à haut niveau.

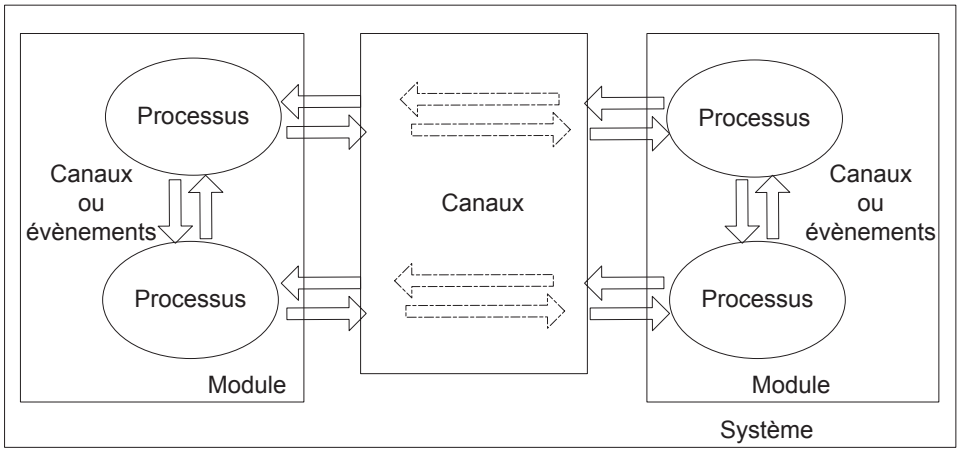


Figure 2.7 – Un système en SystemC [Vermeersch et al., 2002]

SystemC peut modéliser des processus séquentiels. La conception en SystemC est hiérarchique et modulaire. Chaque module peut contenir des sous-modules. Le plus haut niveau sera le module principal. La hiérarchie complète est définie à travers les modules avec leurs différents niveaux hiérarchiques. Des modules sont connectés en utilisant des canaux primitifs. Les ports sont utilisés pour relier les types de canaux entre les modules. Les types de ports peuvent être port types « sc_in », « sc_out » et « sc_inout ». Le premier est un port d'entrée, le deuxième est un port de sortie et le dernier est un port bidirectionnel. Dans l'exemple du code 2.6, nous avons un module « MonModule » contenant deux entrées et une sortie avec une méthode « Fairetraitement ».

Listing 2.6 – C++

```
1  #include "systemc.h"
2  SC_MODULE(MonModule) {
3      sc_in <bool> din; // Donnée d'entrée
4      sc_in <bool> clk; // L'horloge
5      sc_out <bool> dout; // Sortie
6      void Fairetraitement() { // Processus réalisant une fonction
7      dout = din;
8  }
9  SC_CTOR(MonModule) { // Constructeur
10     SC_METHOD(Fairetraitement); // Méthode appelée Fairetraitement
11     sensitive_pos << clk; // Exécute Fairetraitement() à chaque cycle d'horloge
12 }};
```

SC_METHOD est une fonction sans notion de temps. Nous avons aussi SC_THREAD permettant une exécution concurrente et peut être arrêtée pour attendre certains évènements et continuer à nouveau son traitement. Les évènements et les listes de sensibilité permettent à SystemC de simuler un comportement matériel. Les évènements sont décrits par la classe « sc_event ».

Dans un processus SC_THREAD, la méthode « wait_until » peut être utilisée pour contrôler son exécution. Cette méthode permet d'arrêter l'exécution du processus jusqu'à ce qu'un évènement spécifique arrive et qui est décrit dans « wait_until ». Par exemple, « wait_until (A) » permet de continuer l'exécution uniquement quand l'évènement A se produira. De plus, il est possible de mettre du temps tel que « wait_until (10ns) ». Le simulateur va débloquer le processus après un avancement de 10ns dans le temps.

Six étapes [Bois et al., 2003; Martin, 2003] sont requises pour effectuer une simulation dans SystemC.

1. Incrémentation du temps de simulation.
2. Exécution des méthodes et de processus qui ont un changement à leur entrée.
3. Mise à jour des sorties de certains processus.
4. Si des changements surviennent sur une ou plusieurs sorties, recommencer à partir de 2.
5. Exécution des processus. Les sorties seront propagées à l'étape 3 du prochain top d'horloge.
6. Incrémentation du temps de simulation et redémarrage à l'étape initiale.

Il exécute les processus selon les changements à leurs entrées permettant une simulation événementielle.

2.4.5 ESyS.Net

ESyS.Net est un langage de modélisation comparable à SystemC pour décrire des modules, des communications, des canaux, des signaux et des évènements illustrés à la figure 2.8.

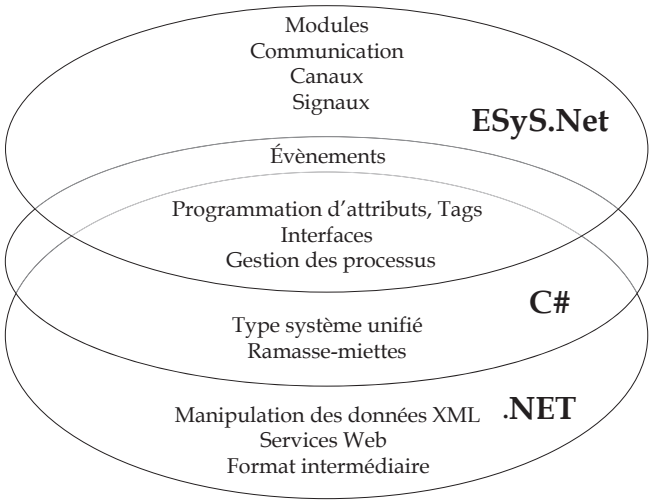


Figure 2.8 – ESyS.Net [Metzger et al., 2006]

Chaque signal et canal peut générer des évènements qui seront traités dans un autre ordre propre au simulateur comme illustré par l'axe causal dans la figure 2.4.

Cette modélisation est décrite en utilisant le langage C# et simulée par le code ESyS.Net qui, lui-même, est écrit dans le même langage. ESyS.Net utilise les caractéristiques C# (.NET) pour exécuter le modèle incluant des processus séquentiels. Il possède aussi l'introspection permettant d'avoir des informations sur les types et les méthodes dans le code compilé.

2.4.6 Comparaison des simulateurs

La majorité des langages présentés permet une simulation à plusieurs niveaux d'abstraction. Certains d'entre eux offrent des bibliothèques pour faire de la vérification. Le niveau RTL est largement répandu pour les langages en VHDL et Verilog, tandis que SystemC et SystemVerilog permettent bien le TLM. ESyS.Net, SystemC et SystemVerilog offrent la modélisation TLM ainsi que RTL.

Les langages de modélisation offrent des durées de simulations différentes selon l'hôte du langage et leur méthode de calcul. Dans tous les cas, une modélisation RTL nécessite un temps discret pour effectuer une simulation. VHDL, ESyS.Net et SystemC utilisent les deltas délais tandis que Verilog et SystemVerilog ont des instructions d'af-

fectation bloquante et non bloquante. Dans le cas de VHDL, les processus sont exécutés selon les évènements qui les affectent, chaque assignation peut être recalculé plusieurs fois d'où le delta délais. Verilog utilise plusieurs régions et catégorise chaque évènement selon sa région et chaque région est exécutée dans un ordre précis.

L'utilisateur de ces langages peut principalement modéliser des processus séquentiels pour représenter un circuit. De plus, certains proposent des ports de communication, des canaux et autres pour la conception d'un système.

2.5 Techniques d'accélération

Certaines techniques d'accélération ont été proposées pour diminuer la durée d'une simulation. La première technique est la simulation compilée du RTL. La deuxième est le « levelized compiled code », la troisième est la simulation logicielle architecturale et la dernière technique est SystemCASS par les machines à états finis.

2.5.1 Simulation compilée RTL

Un circuit électrique est composé d'éléments séquentiels et d'éléments logiques. Les éléments séquentiels réagissent à une horloge et les éléments logiques sont des éléments combinatoires. Par exemple, un registre est un élément séquentiel, car il dépend d'une horloge pour fonctionner. Il change de valeur uniquement avec une transition d'horloge. Par contre, un additionneur combinatoire de base est un élément combinatoire et sa valeur résultante change en fonction de ses entrées. Il existe trois descriptions pour réaliser une compilation d'un simulateur au niveau RTL [Kupriyanov et al., 2004].

Description 1 (la liste de sensibilité). Une liste de sensibilité est définie comme étant des éléments séquentiels de $\{ur_1 , \ldots , ur_n\}$ d'un circuit qui est calculé à chaque fois qu'au moins un élément appartenant à $\{vr_1,\ldots , vr_m\}$ a changé depuis le cycle précédent de simulation. La notation est représentée sous la notation suivante :

$\{vr_1 , \ldots , vr_m\} \rightarrow \{ur_1 , \ldots , ur_n\}$.

« Ur_1 » à « ur_n » correspond aux sorties des registres et « vr_1 » à « vr_n » sont les entrées. Chaque élément (registre) doit uniquement changer de valeur si son entrée (vr)

a été transformée. Nous avons un recalcul lors d'une transition d'horloge de la nouvelle valeur de sortie du registre en fonction de ses entrées.

Description 2 (Netlist). Un netlist N = (V, F) est un ensemble V des éléments logiques et un ensemble F pour l'interconnexion des éléments v ∈ V les uns aux autres. Le netlist représente les termes en éléments logiques de base. Un noeud unidirectionnel f ∈ F qui connecte n+m éléments sera représenté par f = ($\{v_1, \ldots, v_n\}$, $\{u_1, \ldots, u_m\}$), où $v_1, \ldots, v_n \in$ V sont les noeuds sources et $u_1, \ldots, u_m \in$ V sont les noeuds cibles de f.

Description 3 (Netgraph). Un netgraph G = (V,E). G est construit à partir du netlist N. Un ensemble de noeuds V = $V_r \cup V_c$, représentant des registres V_r et des éléments combinatoires V_c sont donnés par le netlist N = (V, F). L'interconnexion des netlists est représentée par un arc dirigé e = (v_1, v_2) ∈ E.

Les trois descriptions servent à bien cadrer ce qu'on désire supporter dans l'architecture pour le RTL.

Essentiellement, nous avons les éléments combinatoires et synchrones (horloge) pour un circuit électronique. Précédemment, nous avons vu les descriptions des éléments séquentiels dans la modélisation des langages. L'objectif d'une simulation compilée est de créer, à partir de ces éléments, une représentation sous forme de graphe pour avoir un ordonnancement statique avant l'exécution. À partir des trois définitions, on peut dire qu'un netlist est créé à partir des listes de sensibilité pour créer un graphe.

Par exemple, la figure 2.9 montre 4 signaux « sig1 » à « sig4 ». « Sig1 » et « sig2 » feront partie de la liste « vr » et « sig3 », « sig4 » de la liste « ur ». La liste « ur » représente les éléments séquentiels combinatoires et la liste « vr » dépend de l'horloge. « Sig1 » et « sig2 » n'ont pas changé de valeur après un changement d'horloge et il est inutile de recalculer « sig3 » et « sig4 ». La figure 2.10 représente le graphe de l'exemple.

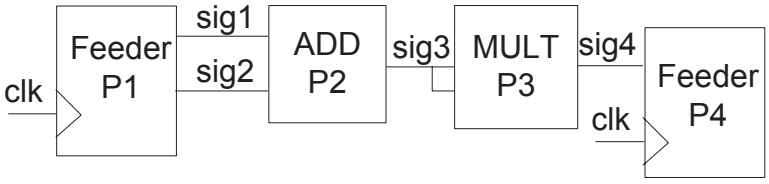


Figure 2.9 – Exemple RTL

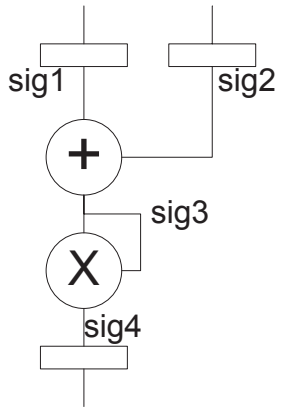


Figure 2.10 – Exemple RTL graphique

Les rectangles de la figure 2.10 représentent les bascules où les registres et les cercles sont des opérations combinatoires à exécuter. Dans notre exemple, Vr = {sig1 , sig2} et Vc = {sig3 , sig4}. Vr est le carré et Vc est le cercle. Nous aurons V = {sig1 , sig2, sig3 , sig4}. Il faut déterminer l'ordre d'exécution du calcul des variables.
Tous les éléments combinatoires Vc peuvent être ordonnancés statiquement lorsqu'il n'y a pas de cycles. Tous les registres Vr peuvent être mis à jour à tout changement d'horloge. Des techniques de décomposition et de partitionnement de calcul de graphes peuvent être utilisées [Bhasker, 1988]. Un partitionnement est réalisé pour optimiser des graphes dans le but de faire une simulation. L'ordonnancement statique à partir des graphes créés lors d'une compilation d'un modèle permet d'avoir une simulation plus rapide qu'un ordonnancement dynamique qui peut calculer plusieurs fois la même assignation.

2.5.2 Levelized compiled-code (LCC)

Le levelized compiled code (LCC) [Laung-Terng et al., 1987] permet une simulation au niveau registre. Un algorithme général est défini en 2.7 pour LCC.

Listing 2.7 – Un algorithme de LCC [Laung-Terng et al., 1987]

```
1 Répéter n cycles d'horloge
2    Prendre les patrons d'entrées correspondants à un
        cycle d'horloge
3    Exécuter la simulation LCC une seule fois
4    (code généré préalablement par un algorithme de logic
        levelization)
5    Produire les résultats de sortie et
6    Comparer avec les résultats prévus si nécessaire
```

L'algorithme consiste à exécuter un code avec une durée de simulation correspondant à n cycles d'horloge. Cette technique est l'émulation du chemin de données à partir des entrées et des sorties des bascules. Par contre, s'il y a une boucle dans le chemin de données, il n'est pas supporté. L'algorithme exécute sous forme d'une boucle l'horloge et aussi l'algorithme prend les noeuds dépendant de l'horloge pour exécuter une simulation LCC. Un algorithme de logic levelization au code 2.8 permet de voir plus en détail le fonctionnement d'un LCC.

Listing 2.8 – Un algorithme de Logic Levelization [Laung-Terng et al., 1987]

```
1 Marquer tous les noeuds se reliant aux ports d'entrées
     et aux sorties des bascules comme disponibles
2 Mettre les portes logiques se reliant aux ports d'
     entrées et les sorties des bascules dans une file d'
     attente
3 Tant que la file d'attente n'est pas vide
4    Prendre une porte logique de la file d'attente dont
        toutes les entrées sont marquées disponibles
5    Marquer tous les noeuds de sorties de la porte
        logique comme disponibles
6    Produire du code source en C (une instruction ou une
        expression)
7    Mettre dans la file d'attente les portes logiques qui
        sont reliées au noeud de sortie.
8 Si le nombre de noeuds disponibles n'est pas égal à tout
     le nombre de noeuds, alors rapportez l'erreur. Le
     circuit contient des boucles de rétroaction qui
     génèrent des noeuds qui ne sont pas disponibles.
```

Tout d'abord, nous identifions les noeuds d'entrées et de sorties par rapport à la

bascule. Ensuite, nous inscrirons disponibles si et seulement si les noeuds ne sont pas dans une boucle de rétroaction pour les mettre dans une liste d'attente. Pour terminer, nous génèrerons un code source à partir des noeuds disponibles jusqu'à ce que la liste d'attente soit vide. Nous aurons une erreur si les noeuds disponibles ne sont pas égaux au nombre de noeuds.

Cette technique ne prend pas en compte le temps de propagation (délai combinatoire), car on a un ordonnancement statique de l'ordre d'exécution des parties combinatoires (portes logiques). Une fois le code ordonnancé, le code est exécuté une seule fois par cycle d'horloge.

On obtient avec cette technique une simulation plus rapide, car le code est ordonnancé statiquement et exécuté séquentiellement dans une boucle représentant un cycle d'horloge par tour de boucle.

Une autre approche est une simulation logicielle architecturale que nous expliquerons dans la prochaine section.

2.5.3 Simulation logicielle architecturale

L'approche d'une simulation architecturale est d'incorporer les caractéristiques telles que la puissance dans des simulateurs de microarchitecture [Lee et al., 2004]. Une microarchitecture peut être un processeur. La figure 2.11 présente la simulation logicielle architecturale. Elle consiste en un simulateur de circuit, une architecture et des métriques telles que la puissance et la surface.

Le simulateur architectural prend deux entrées principales : un fichier de configuration qui définit la microarchitecture (un processeur) qui a été modélisée, et un programme à exécuter. La configuration définit les étapes du pipeline, plus toutes les unités spéciales qui résident dans ces étapes, telles que des facteurs prédictifs de branchement, des antémémoires, des unités fonctionnelles et des interfaces de bus. Un pipeline peut être celui d'un processeur à cinq étages.

L'exécution du simulateur de circuit est basée sur des événements. Par contre, il est important dans ce type de simulation d'avoir une évaluation des métriques d'un circuit. Une cellule de caractérisation a été créée pour chaque élément qu'on veut analyser. L'al-

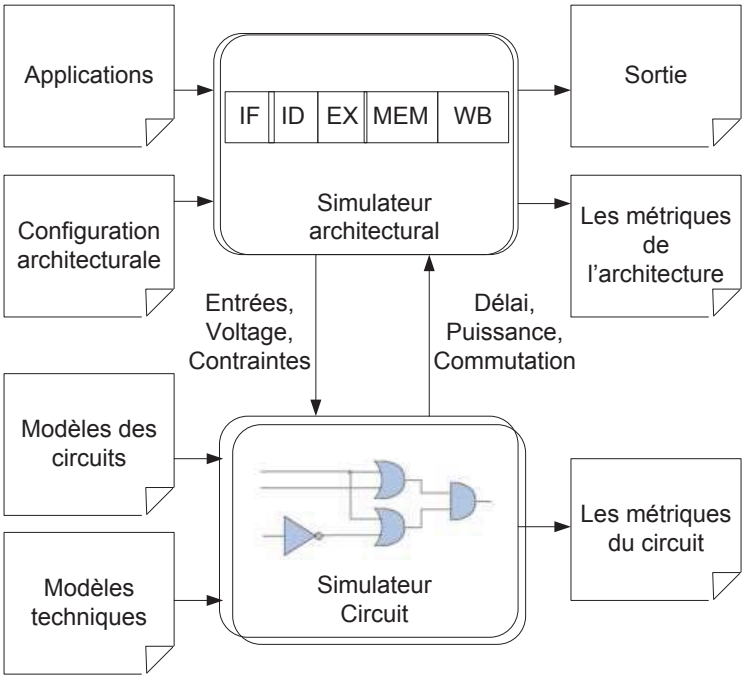


Figure 2.11 – Simulation logicielle architecturale [Lee et al., 2004]

gorithme de simulation circuit est décrit au pseudo-code 2.9. C'est une simulation à partir d'un vecteur d'entrée. Chaque vecteur d'entrée est composé de plusieurs bits.

Listing 2.9 – Un algorithme de simulation circuit [Lee et al., 2004]

```
1 Simulation (vecteur d'entrée)
2 Pour chaque bit qui a changé du vecteur d'entrée
3    Génère une transition sur le bit du vecteur
4    Met cette transition dans une liste d'attente
5 Tant que la liste d'attente n'est pas vide
6    Enlève la transition la plus récente
7    Pour chaque cellule, si la transition appartient à la
        cellule d'entrée
8       Évaluation de la transition et ajouter la
           caractéristique voulue.
```

À chaque fois qu'on a une transition sur un bit, on génère une transition dans une liste d'attente ordonnée. On vide la liste d'attente en partant de la transition la plus récente. Chaque transition qui a été faite sur un bit possède une cellule avec ses caractéristiques et correspondant à un bit d'entrée. L'algorithme évalue ensuite la caractérisation vou-

lue, telle que la puissance et met à jour cette dernière, par rapport à l'autre élément. Par exemple, un ensemble de portes logiques sera évalué séparément quand il y a un changement à leur porte. Nous pouvons aussi obtenir la puissance qu'elle consomme séparément. Nous obtiendrons la puissance totale du circuit en additionnant les puissances de chaque porte.

L'accélération de la simulation est réalisée en réduisant le nombre d'instructions à simuler pour obtenir la caractérisation voulue.

2.5.4 Simulation au niveau cycle de SystemCASS

SystemCASS est une approche de simulation utilisant une description SystemC [Buchmann and Greiner, 2007]. Ce simulateur construit un graphe de dépendance entre les signaux durant la phase d'initialisation. Avant la compilation, il est possible de connaître l'interdépendance entre les signaux pour le niveau cycle (RTL). La figure 2.12 présente la technique utilisée pour accélérer la simulation. Ce modèle utilise des machines communicantes synchrones à états finis qui permet d'effectuer un ordonnancement statique correspondant au circuit à simuler.

La création d'une machine à état permet d'exécuter plus rapidement un modèle. Cependant, il ne supporte pas les interfaces et une modélisation TLM. Principalement, on utilise deux types de machine à état fini soit Mealy et Moore [Bensalem et al., 2008; Solovjev and Chyzy, 1999]. Dans un FSM de Mealy, les valeurs des variables de sortie dépendent à la fois de l'état courant et des variables d'entrées. Le diagramme d'états et de transitions contiendra un signal d'entrée et de sortie pour chaque transition. Dans le cas d'une machine de Moore, les valeurs en sortie ne dépendent que de l'état courant. SystemCASS utilise les deux pour créer un FSM d'une simulation qui sera compilé pour être exécutée. Selon l'auteur, un gain de performance jusqu'à 15 fois par rapport à SystemC 2.1 peut être obtenu selon les applications utilisées.

Les techniques d'accélération sont principalement pour le niveau RTL. Celles-ci ordonnancent statiquement un circuit pour pouvoir l'exécuter. Ce simulateur est plus rapide que les simulateurs qui ne font pas d'ordonnancement statique avec des graphes.

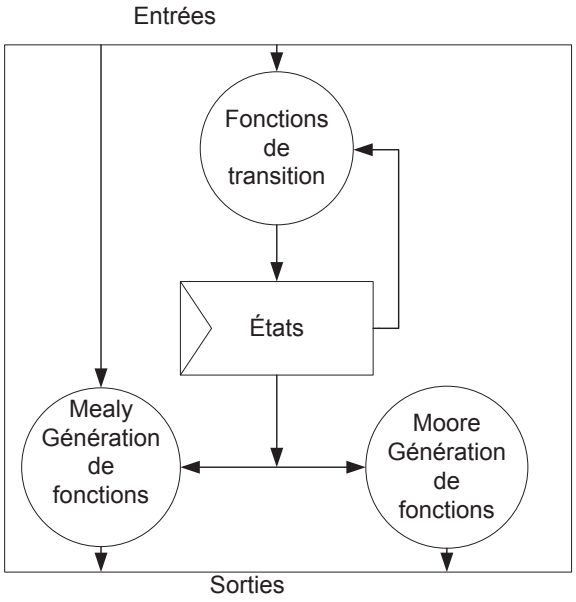


Figure 2.12 – Simulation par machine à états finis

2.6 Co-simulation

La co-simulation exécute concurremment au minimum deux noyaux de simulation en utilisant quelques mécanismes de synchronisation. Le but est de lier deux modèles dans des langages hôtes différents (figure 2.13). Trois méthodes principales sont employées couramment : le même fichier binaire, la mémoire partagée et le TCP/IP.

L'interopérabilité des langages de modélisation telle qu'illustrée à la figure 2.14 permet d'identifier les principaux éléments pour obtenir une co-simulation à travers des méthodes de communication standard. La figure présente deux langages différents de modélisation et leur interaction qui sont importants pour effectuer une co-simulation. Il faut connecter, au besoin, quatre éléments ensemble ; comme l'interaction des niveaux d'abstraction, l'interaction des langages, la synchronisation des noyaux et les modèles d'interactions. Les modèles doivent être capables de s'échanger des informations entre eux. Au besoin, des langages de programmation peuvent être interconnectés. Si les niveaux d'abstraction sont différents, il est primordial de créer des adaptateurs.

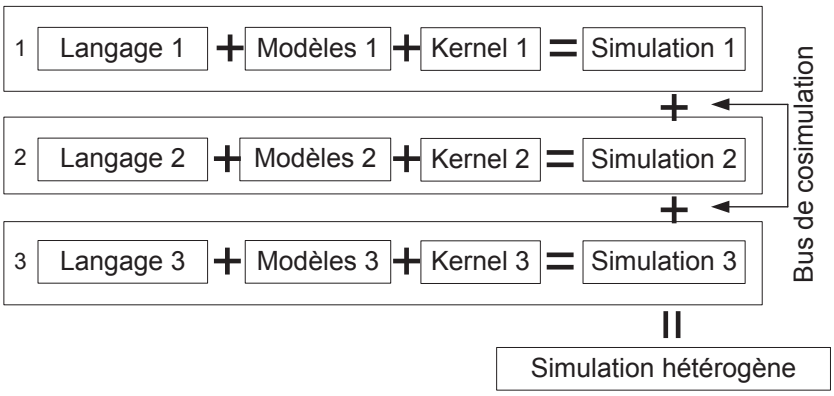


Figure 2.13 – Co-simulation standard avec plusieurs noyaux [Dubois et al., 2006]

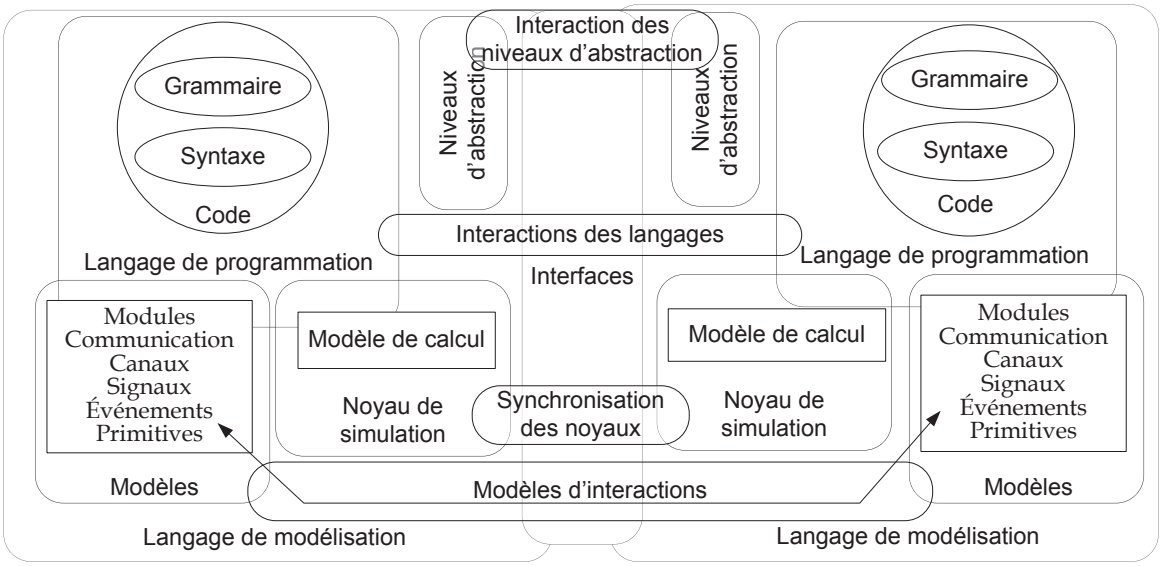


Figure 2.14 – Langages de modélisation [Dubois et al., 2009]

De même lors de la synchronisation entre deux noyaux de simulation, il faut mettre en phase les deux noyaux.

Une co-simulation matérielle logicielle est importante dans le processus de conception des systèmes. Plusieurs méthodes ont été présentées pour améliorer la vitesse de co-simulation. Un environnement de co-simulation est proposé en [Kim et al., 1996], consistant à générer des interfaces d'interconnexions automatiques. Par exemple, un langage C et un langage de modélisation matériel (VHDL) sont connectés par un lien TCP/IP. Nous avons une génération automatique de code pour l'interaction des langages et des modèles par des fonctions d'écriture et de lecture pour le lien TCP/IP. Nous discuterons du TCP/IP ultérieurement.
De manière générale, une co-simulation est possible lorsqu'un point commun entre deux langages de modélisation est trouvé et qu'il y a une synchronisation entre eux. En [Bishop and Loucks, 1997], la difficulté associée à la conception et à l'interconnexion est présentée avec des stratégies réduisant des contraintes de tailles et de performances. La gestion du temps et de la synchronisation sont aussi discutées comme une méthode d'accusé de réception (figure 2.15).

C'est une méthode de synchronisation des noyaux permettant d'avoir une simulation concurrente. Les deux noyaux avancent selon des plages de temps et ils conservent une synchronisation par des accusés de réceptions.

2.6.1 Mécanisme standard

Les mécanismes standards que nous expliquons dans cette section sont la compilation dans le même binaire, la méthode de connexion par mémoire partagée et la méthode de connexion par TCP/IP.

2.6.1.1 Compilation dans le même binaire

La compilation dans le même binaire (un seul exécutable) est employée par deux noyaux de simulation ayant des parties complémentaires d'un modèle. Des approches dynamiques et statiques sont employées pour la co-simulation. « Mentor Seamless CVE »

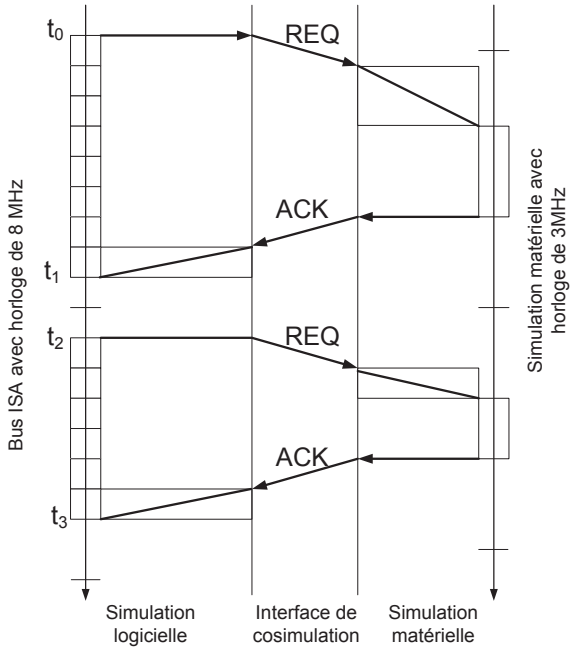


Figure 2.15 – Synchronisation avec la méthode accusé de réception

utilise une approche dynamique. Il peut appeler une bibliothèque pendant son temps d'exécution, sans connaître les informations sur celle-ci. SystemC emploie une approche statique. Il connaît toutes les bibliothèques avant son exécution. Les bibliothèques statiques sont compilées avec le code source principal avant l'exécution. Celles qui sont dynamiques sont précompilées et liées lors de l'exécution du programme. Le lien entre des outils peut être fait en déclarant quelques prototypes des fonctions qui sont employées en commun. La compilation dans le même binaire permet d'avoir un seul code à exécuter et peut être plus rapide. L'inconvénient c'est qu'il n'est pas possible de l'utiliser dans un environnement distribué. Il est aussi limité au langage hôte pouvant être incorporé dans un même fichier. La difficulté d'avoir un même fichier binaire est liée à la réalisation d'un compilateur supportant plusieurs langages, car il faut qu'il soit capable de mettre en commun les informations à exécuter.

2.6.1.2 Méthode de connexion par mémoire partagée

Un même espace mémoire est partagé par deux noyaux de simulation. La mémoire partagée est divisée en deux parties principales : les données échangées et les commandes nécessaires pour synchroniser les simulateurs. Ceci peut être utile pour une co-simulation asynchrone d'une modélisation au niveau transactionnel. Par exemple, la figure 2.16 illustre trois consommateurs et un producteur. Nous avons trois FIFOs pour le modèle et trois autres pour réaliser la communication avec la mémoire partagée. Les FIFOs entre A et B permettent une communication entre les deux langages hôtes (ESyS.Net et SystemC). Le lien entre C et B est la liaison entre la mémoire partagée et le modèle. Chaque processus, dans les deux langages, peut être exécuté concurremment sans réduire sa vitesse de simulation, car les FIFOs sont employés pour protéger des données et pour régulariser le débit de communication.

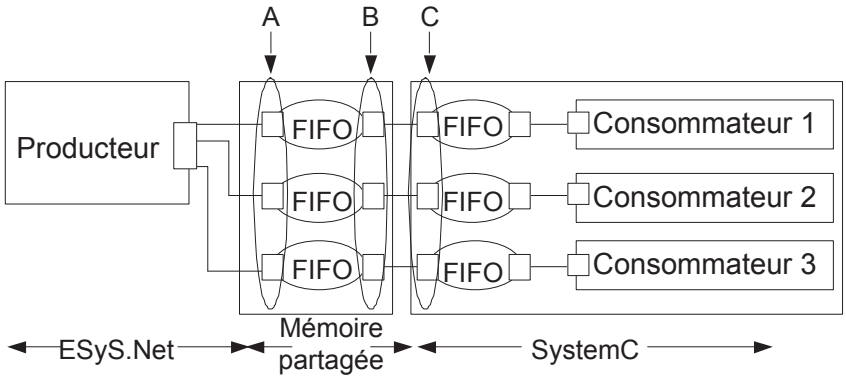


Figure 2.16 – Exemple de mémoire partagée

2.6.1.3 Méthode de connexion par lien TCP/IP

La méthode TCP/IP consiste à avoir un serveur et un client. À l'initialisation, le serveur écoute le port de communication et le client doit demander une connexion. Dans chaque niveau d'abstraction, les données entre des simulateurs doivent être encapsulées dans le format TCP/IP. Il exige une structure de communication que le serveur et le client peuvent comprendre. Au temps d'exécution, un événement devrait être connecté

à un lien synchrone ou asynchrone et également la synchronisation devrait être transmise par le TCP/IP. À un instant spécifique, le client peut lire des données du canal et reprendre son exécution en avançant son temps discret et, si nécessaire, il envoie une réponse au serveur. Un client peut également attendre un paquet TCP/IP pour reprendre son exécution. Une autre possibilité est que le serveur et le client peuvent avancer leur temps discret pour une période spécifique.

Dans ce chapitre, nous avons vu plusieurs langages de modélisation pour simuler un système et des techniques de compilations. Chaque langage offre un environnement de conception pour que l'utilisateur puisse concevoir son système. Nous avons principalement discuté des niveaux RTL et TLM ainsi que les méthodes de calculs à utiliser pour simuler leur comportement. De plus, l'interaction entre plusieurs simulateurs a été abordée.

Le temps de simulation est un critère important pour pouvoir valider rapidement un système et choisir les bonnes configurations qu'un concepteur souhaite. La co-simulation permet d'avoir plusieurs descriptions de modèles, mais ajoute des mécanismes de co-simulation qui augmentent le temps de conception. Les plus utilisés sont la mémoire partagée et le TCP/IP. Ceux-ci peuvent avoir des limitations lors de la conception d'un système en entier ayant beaucoup de mécanismes et demandant une synchronisation adéquate. Le prochain chapitre discute de nouvelle manière de co-simuler des modèles hétérogènes.

CHAPITRE 3

ADAPTATIONS DES TECHNIQUES DE CO-SIMULATION ET DE SIMULATION

Nous proposons dans cette section des mécanismes pour la co-simulation soit : le Component Object Model (COM), la fonction statique, le pinvoke et l'adaptateur géré. Ceux-ci sont bien adaptés à l'environnement .NET et se concentrent sur la simulation discrète. .NET permet l'exécution de plusieurs langages de conception généraliste et leur interopérabilité.

Par contre, le COM n'est pas adapté pour la co-simulation entre des noyaux de simulation. L'échange entre les noyaux demande une synchronisation qui n'est pas directement fournie par .NET. Les données pour la modélisation des entrées et sorties entre des modèles décrits dans deux environnements différents doivent être sérialisées, transmises et désérialisées pour effectuer une co-simulation. De plus, .NET ne supporte pas des langages largement répandus pour la conception des systèmes comme VHDL.

La synchronisation entre ces langages est réalisée sur des frontières de temps discret. Celles-ci peuvent être un top d'horloge commun pour des systèmes synchrones, un événement commun, ou des sauts d'un temps discret au prochain par un nombre de temps spécifique. Sauf indication contraire, nous supposons qu'un noyau spécifique de simulation est défini comme serveur et l'autre en tant que client. Le serveur a des points d'entrées et le client les emploie pour commander le serveur. Nous présenterons également une simulation hétérogène en utilisant l'interopérabilité et quelques changements syntaxiques. Nous avons publié les résultats de ce chapitre dans [Dubois and Aboulhamid, 2005], sauf la section 3.3.

3.1 Adaptations des techniques de co-simulation

Certaines techniques présentées sont des compilations dans le même binaire et leurs mécanismes d'interactions pour connecter deux langages hôtes sont différents. Dans tous

les cas, sauf pour l'adaptateur géré, les données communiquant entre les modèles d'un système doivent passer par une structure prédéfinie.

Trois étapes sont nécessaires pour effectuer un co-simulation. Le premier hôte met ses données dans une structure et la communique au deuxième hôte par la technique proposée. Ensuite, l'hôte 2 avance son temps discret et récupère ses données pour l'envoyé à l'hôte 1. Et pour terminer, l'hôte 1 avance son temps discret. La fonction statique, le COM et Pinvoke utilise des structures fixes pour échanger leur donnée. L'adaptateur géré n'utilise pas une structure, car les modèles peuvent s'interconnecter directement.

3.1.1 Fonction statique

Nous voulons relier un simulateur ESyS.Net et SystemC ensemble pour une modélisation d'un système. Cependant, SystemC est écrit en utilisant le langage hôte C++ et ESyS.Net utilise le C#. Normalement, la communication entre ces deux langages peut se réaliser avec la mémoire partagée ou bien le TCP/IP. L'objectif avec cette technique est d'avoir les simulateurs en C++ et en C# dans un même binaire.

Cette méthode porte le nom de fonction statique, car elle utilise une fonction statique pour lier deux langages hôtes. Elle s'utilise comme des fonctions classiques. Elle est utile quand un code non-géré doit appeler un code géré, contrairement à d'autres techniques qui réalisent l'inverse. Le code géré est un programme informatique qui s'exécute sous la gestion d'une machine virtuelle, à la différence du code non-géré, qui est exécutée directement par le processeur de l'ordinateur [Gough, 2005].

La figure 3.1 illustre un simulateur non-géré (client de SystemC) avec un simulateur géré (serveur d'ESyS.Net). Dans cette approche, ESyS.Net est encapsulé à l'aide de C++ géré (C++ avec une extension gérée), et des fonctions intermédiaires statiques sont créées pour commander et échanger des données entre les simulateurs. Un client peut appeler une fonction statique avec un choix d'actions spécifiques. Les choix 1 à 3 contrôlent la progression de simulation d'ESyS.Net en synchronisation avec SystemC et les choix 4 à N sont utilisés pour l'échange des données entre les simulateurs. L'échange des données peut être fait en ajoutant des variables globales pour lier SystemC avec ESyS.Net.

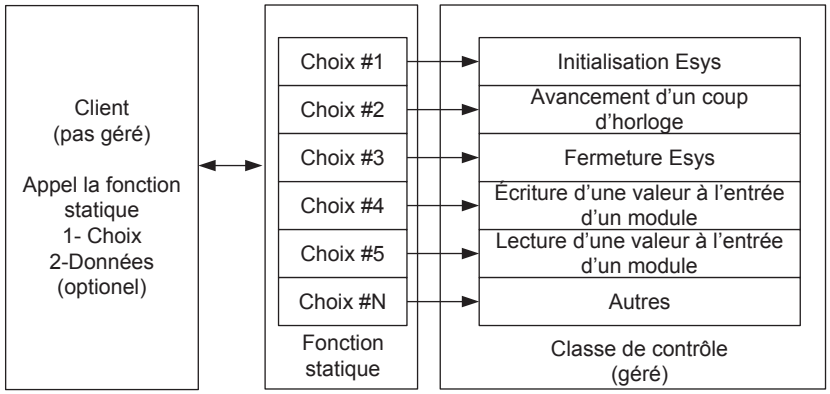


Figure 3.1 – Illustration du fonctionnement pour la fonction statique [Dubois and Aboulhamid, 2005]

Pour réaliser ce type de co-simulation, il est nécessaire d'amener le noyau d'ESyS.Net en C# dans du C++ géré. Nous utilisons dans le fichier C++ la DLL d'ESyS.Net avec la DLL contenant l'adaptateur qui permettra le code géré. Ensuite, nous appellerons les noms des fonctions auxquelles nous accédons. Pour utiliser cette technique, il faut créer une classe pour déclarer et utiliser le noyau d'ESyS.Net avec du C++ géré. La classe « classesys » au code 3.2 permet la communication entre ESyS.Net et la fonction statique (3.1).

Listing 3.1 – Fonction statique

```
1  static int fctstatique(int choix)
2      {
3      classesys *esys;
4      switch (choix)
5          {
6          case 1: {
7              esys->step();
8              return 10;
9              break; }
10         case 2: {
11             return esys->Output0();
12             break;
13         }
14         case 3: {
15             esys = new classesys();
16             esys->init();
17             return 10;
18             break;
19           }
20 }
```

Nous avons les fonctions « init » et « step » pour initialiser et avancer le temps discret d'ESyS.Net. L'échange des données entre les modèles SystemC et d'ESyS.Net passe également par cette classe. La fonction statique « fctstatique » reçoit un paramètre qui est « choix ». Il faut noter que l'échange des données passe à travers des variables générales comme « data0 » et « data1 ». La fonction « fctstatique » est appelée directement par SystemC. Cette technique requiert deux appels de fonction. La première est celle de la fonction statique et la deuxième est celle de la classe « classesys » pour ESyS.Net.

Dans ce cas, nous avons amené les modèles d'ESyS.Net dans ceux de SystemC, la prochaine méthode avec Pinvoke permet d'avoir l'inverse.

Listing 3.2 – ESyS.Net dans du C++ géré

```
1  __gc class classesys
2  {
3  public :
4      static ESysNet::Simulator *sim;
5      static simplebus::simple_bus_test *sym;
6      int matOutput0(void)
7      {
8          data0=(int)sym->data_0->Value;
9          data1=(int)sym->data_1->Value;
10         return 1;
11     }
12     void Step()
13     {
14         sim->Step(1);
15     }
16     void init()
17     {
18         sim = new ESysNet::Simulator();
19         sym = new simplebus::simple_bus_test(sim,"top");
20         sim->AssembleModel();
21     }
22 };
```

3.1.2 Pinvoke

Par opposition à la méthode précédente, nous explorons un arrangement où SystemC agit en tant que serveur et ESyS.Net en tant que client. Pour réaliser cela, la bibliothèque de SystemC est encapsulée dans une bibliothèque de lien dynamique (DLL) [Weihua et al., 2007] avec les points d'entrées qui peuvent commander son noyau de simulation. La figure 3.2 illustre cette co-simulation. Les modèles SystemC et ESyS.Net peuvent communiquer entre eux par un lien dynamique.

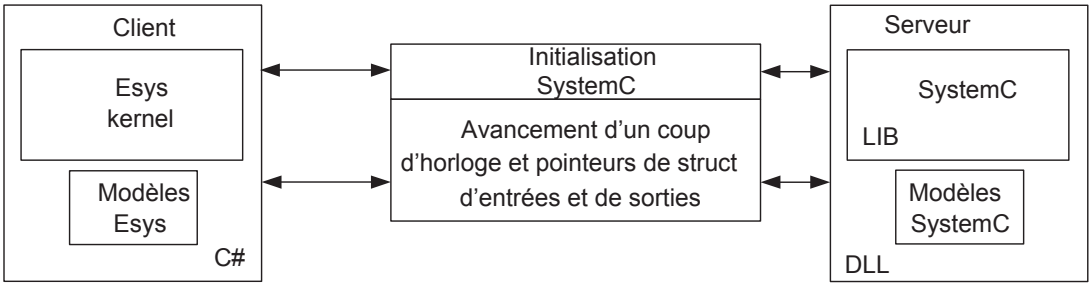


Figure 3.2 – Illustration du fonctionnement pour le Pinvoke [Dubois and Aboulhamid, 2005]

Dans C#, nous déclarons le DLL de SystemC illustré au code 3.3. Nous avons deux fonctions « init » et « run ». La première permet d'initialiser SystemC et la deuxième d'exécuter SystemC au cours d'une co-simulation. Nous passerons à la fonction « run » les vecteurs d'entrées et de sorties ainsi que la plage de temps que l'on désire avancer dans le simulateur SystemC. Cette fonction possède trois sous-fonctions : PutStim, Stimulate et GetOutput illustrées au code 3.4. Les listings 3.3 à 3.5 et 3.7 ont été publiés dans [Dubois et al., 2009]

Listing 3.3 – Appel du DLL de SystemC dans C#

```
1 public class "SystemCDllWrapper
2     {
3     [DllImport("SystemCDll.dll")]
4     public static extern void Init();
5     [DllImport("SystemCDll.dll")]
6     public static extern void Run(invector invect, outvector
          outvect, int simtime);
7     }
```

Listing 3.4 – Contrôle de SystemC

```
1 extern "C" void Run(invector * invector_int,
2 outvector * outvector_int, int simtime)
3 {
4       PutStim(invector_int); // Mettre les entrées
5       Stimulate(simtime); // Avance le simulateur
6       GetOutput(outvector_int); // Donne la réponse
7       _flushall();
8 }
```

D'abord, les entrées d'un module SystemC sont alimentées avec les sorties d'un module du client (ESyS.Net). En second lieu, le simulateur de SystemC avance son temps discret. Et pour finir, la dernière fonction récupère l'information du module de SystemC. Les codes 3.5, 3.6 et 3.7 montrent les fonctions (« putstim »,« stimulation » et « getouput ») en détails.

Listing 3.5 – Fonction de simulation

```
1 extern "C" void PutStim(void *Invector)
2 {
3       INVECTOR *pInvector = (INVECTOR *)Invector;
4       data_in = (sc_uint<32>)pInvector->data[1];
5 }
```

Nous utilisons un pointeur « void * » pour permettre à l'usager de créer sa propre structure. PutStim reçoit le vecteur qui contient des entrées pour être liées aux signaux des modèles en SystemC. Nous récupérons la structure « pInvector » avec un signal comme « data_in » et nous utilisons un type « sc_uint<32> » pour le rendre compatible avec SystemC.

Listing 3.6 – Fonction d'avancement du temps

```
1 extern "C" void Stimulate(int simtime)
2 {
3       sc_cycle((unsigned long)simtime);
4 }
```

Stimulate utilise la fonction « sc_cycle » pour avancer le temps discret du simulateur SystemC. L'utilisation d'un « unsigned long » permet de préciser la plage d'avancement.

Listing 3.7 – Fonction pour récupérer les résultats

```
1  extern "C" void GetOutput(void *Outvector)
2  {
3      OUTVECTOR *pOutvector = (OUTVECTOR *)Outvector;
4      pOutvector->data[0] = data_out1.read();
5      pOutvector->data[1] = data_out2.read();
6  }
```

Une fois que le simulateur a avancé son temps discret, nous allons lire les signaux dans une structure définie. En conclusion, le code 3.8 permet une co-simulation d'un modèle ESyS.Net avec un modèle SystemC.

Listing 3.8 – Simulation d'ESyS.Net et un modèle SystemC avec Pinvoke

```
1  simple_bus_test sm = new simple_bus_test(sim, "DUT");
2  invector myin = new invector();
3  outvector myout = new outvector();
4  sim.AssembleModel();
5  SystemCDllWrapper DLLWrapper = new SystemCDllWrapper();
6  SystemCDllWrapper.Init();
7
8  for (int i = 0;i <= 100;i++)
9  {
10     sim.Step(1);
11     myin.data_in0 = sm.data_0.Value;
12     DLLWrapper.Run(myin,myout,10);
13     sm.data_out0.Value = myout.data_out0;
14 }
```

Dans un modèle d'ESyS.Net, nous créons une boucle pour simuler l'avancement d'une horloge. Avant la boucle, nous initialisons les noyaux d'ESyS.Net et de SystemC. Pour ESyS.Net, il faut assembler le modèle par la commande « sim.AssembleModel ». Nous créons également des vecteurs pour des entrées et des sorties des modèles. L'appel « DLLWrapper.Run » permet au noyau d'ESyS.Net de communiquer avec celui de SystemC.

3.1.3 COM

Le « Component Object Model (COM) » est une norme d'interface présentée par Microsoft en 1993 [Box, 1997; Gani and Picuri, 1995]. Il est employé pour permettre

la communication inter processus et la création d'objets dynamiques dans une gamme étendue de langages de programmation. C'est une façon d'utiliser un langage neutre pour réaliser des objets qui peuvent être utilisés pour l'interopération entre des langages.

Un utilisateur n'a pas besoin de connaître l'implémentation interne des objets. La figure 3.3 illustre le fonctionnement entre le code géré et le code non-géré. Le COM agit comme un intermédiaire entre deux clients soit l'application gérée et non-gérée.

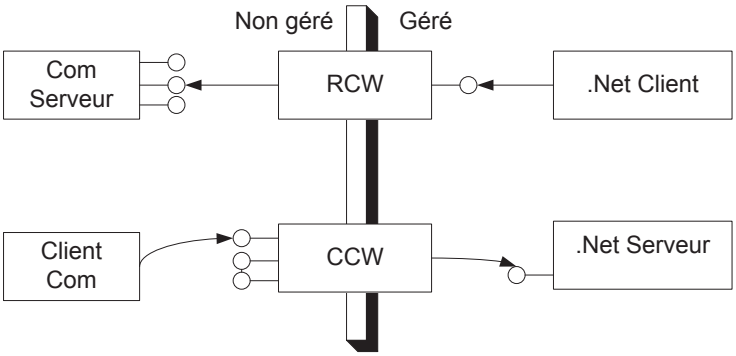


Figure 3.3 – Fonctionnement du COM [Dubois and Aboulhamid, 2005]

Les objets COM sont accessibles par l'utilisation du « common language runtime (CLR) » au travers d'un proxy appelé adaptateur d'appel au temps d'exécution ou « run-time callable wrapper (RCW) ».

Le RCW apparaît au client .NET comme des objets ordinaires. La fonction principale du RCW est de faire le lien entre les clients utilisant .NET et un objet COM. On a un RCW pour chaque objet COM. L'adaptateur d'appel COM ou « COM Callable Wrappers (CCW) » est utilisé pour la communication avec l'objet .NET à l'exécution (CLR). Le CCW agit comme un intermédiaire pour permettre à un client COM d'appeler un objet .NET. En résumé le CCW permet d'avoir du .NET vers un client COM dans un code géré, tandis que le RCW est fait pour du code non-géré.

Une co-simulation avec la méthode COM permet une simulation hétérogène entre deux simulateurs. Elle est possible en créant quelques fonctions pour commander un simulateur visé. Par exemple, nous créons un objet COM avec un ou deux modules et toutes les fonctions pour permettre le contrôle d'un simulateur. Deux fonctions princi-

pales sont utilisées soit l'initialisation et la fermeture. Des fonctions sont aussi nécessaires pour transmettre l'entrée et les données des modèles inclus dans un objet COM. Un dernier élément est nécessaire pour une bonne synchronisation est une fonction qui laisse avancer le temps discret d'un simulateur dans le COM.

La figure 3.4 illustre la relation entre un client et un serveur. Dans ce cas-ci, ESyS.Net est compilé avec ses modèles dans un objet.

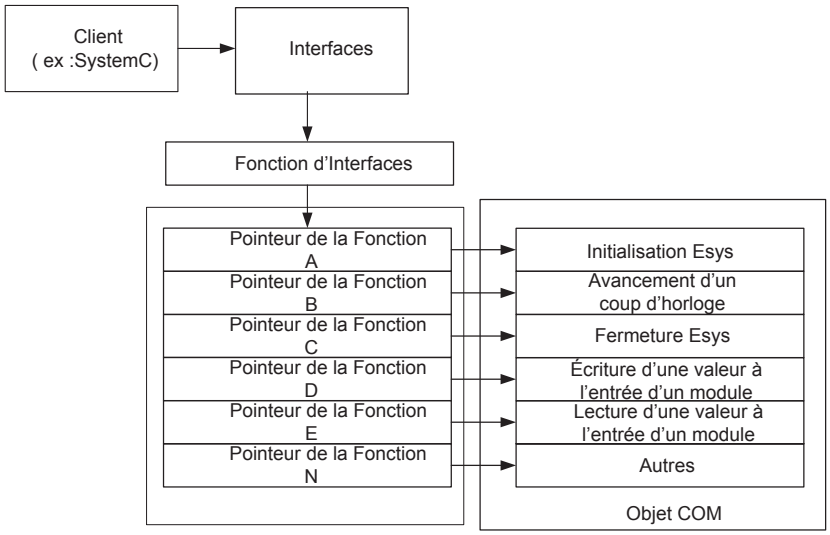


Figure 3.4 – Exemple du fonctionnement de COM avec SystemC et ESyS.Net

Les fonctions A à C servent aux contrôles du simulateur. Les fonctions D et E sont pour l'écriture ou la lecture des modules dans l'objet. La fonction N représente d'autres fonctions définies pour l'utilisateur.

Le code 3.9 illustre l'interaction de SystemC avec l'objet. Les lignes 1 à 28 initialisent l'objet, accomplissent une simulation et libèrent l'objet.

Listing 3.9 – Co-simulation avec un COM d'ESyS.Net et SystemC

```
1  //Initialisation de SystemC
2  CoInitialize(NULL);
3  hr = spmatesys2.CreateInstance(__uuidof(matesys));
4  if (SUCCEEDED(hr))
5       {
6            spmatesys2->init();
7       }
8  else{
9            printf("Echec\n");
10      }
11 for(i = 0; i < 100000;i++)
12      {
13           //Relie les signaux ensemble
14           spmatesys2->matInput1 = data_out1;
15           //Avance un pas de simulation
16           spmatesys2->Step();
17           clk = true;
18           sc_cycle(50);
19           clk = false;
20           sc_cycle(50);
21           //Met les signaux de sortie
22           data_in = spmatesys2->matOutput1;
23      }
24 //Ferme le simulateur
25 spmatesys2->EndEsys();
26 //Libère l'objet COM
27 CoUninitialize();
28 }
```

Par exemple, la fonction A de la figure 3.4 permet d'initialiser ESyS.Net. Une fonction « init » est déclarée dans l'interface « Imatesys2 » illustré au code 3.10 et elle est appelée en utilisant spmatesys2 -> init().

Listing 3.10 – Interface COM dans ESyS.Net

```
13 Interface accessible (Dans le COM ESyS.Net)
14 public interface Imatesys2 {
15      double matInput1 { set; }
16      double matInput2 { set; }
17      double matOutput1 { get; }
18 void Step(); void init(); void EndEsys();}
```

De la même manière, on a des fonctions déclarées pour chaque élément du simulateur

comme « EndEsys » et « Step ». Chacun d'eux permet soit d'incrémenter le temps discret du simulateur d'ESyS.Net ou de le fermer.

L'adaptateur COM permet une interaction entre les modèles de SystemC et d'ESyS.Net comme la fonction statique et le Pinvoke. La technique avec le COM permet d'appeler directement les fonctions d'ESyS.Net sans passer par une fonction intermédiaire comme la fonction statique. L'utilisateur doit pour toutes les méthodes construire une structure pour échanger des données entre les simulateurs. Au niveau de la compilation, il faut créer des adaptateurs pour lier les simulateurs.

3.1.4 Adaptateur géré (MW)

La dernière méthode est basée sur une classe gérée qui consiste à encapsuler la bibliothèque SystemC dans une classe gérée avec des points d'entrées qui peuvent commander son noyau. Cette méthode permet des applications intéressantes, parce qu'elle peut directement relier un module ESyS.Net à un module SystemC sans intermédiaire, tel qu'illustré a la figure 3.5.

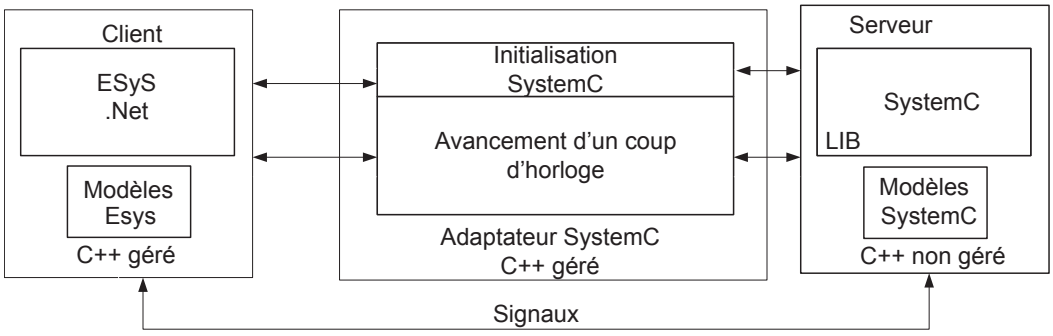


Figure 3.5 – Adaptateur géré (MW)

Dans cette méthode, le code géré appelle le code non-géré. Des vecteurs d'entrées et de sorties n'exigent pas une structure intermédiaire. Les signaux des modèles de SystemC peuvent être reliés directement à ceux d'ESyS.Net. Il est nécessaire d'initialiser les deux simulateurs et d'incrémenter leur temps discret d'une manière synchrone. Le noyau d'ESyS.Net exécute T0 et demande à celui de SystemC d'exécuter T0 et après

le noyau d'Esys.Net va au temps T1 et celui de SystemC va aussi au temps T1 et ainsi de suite. Pour créer un adaptateur géré en SystemC, il suffit d'écrire une classe gérée en C++. La classe « SysyemCWrapper » du code 3.11 illustre un adaptateur pour SystemC. L'étiquette « __gc » indique au compilateur que c'est une classe gérée.

Listing 3.11 – Adaptateur SystemC pour du C++ géré

```
1  #pragma managed
2  public __gc class SystemCWrapper
3  {
4  private:
5  SystemCClass* ac;
6  public:
7  SystemCWrapper() {ac = new SystemCClass();}
8  void Init(void)
9  {ac->Init();}
10 void Stimulate(unsigned long simtime)
11 {ac->Stimulate(simtime);}
12 ~SystemCWrapper() {delete ac;}
13 };
```

La classe « SystemCClass » du code 3.12 donne accès à SystemC parce que le terme « __nogc » définit une classe non-gérée.

Listing 3.12 – Classe pour simuler et initialiser SystemC

```
1  __nogc class SystemCClass
2  {
3  public:
4  void Init(void);
5  void Stimulate(unsigned long simtime);
6  };
7  void SystemCClass::Init()
8  {
9      the_module.clk(clk);
10     the_module.data_out1(data_out1);
11     the_module.data_out2(data_out2);
12     the_module.data_in(data_in);
13     sc_initialize();
14 }
15 void SystemCClass::Stimulate(unsigned long simtime)
16 {
17     sc_cycle(simtime);
18 }
```

Le contrôle et les données de SystemC sont encapsulés dans une classe. Le code 3.13 illustre les déclarations des signaux et d'un module d'ESyS.Net.

Listing 3.13 – Classe pour connecter les signaux ESyS.Net et SystemC

```
1 #pragma unmanaged
2 sc_signal<bool> clk;
3 sc_signal<int> data_out1;
4 sc_signal<int> data_out2;
5 sc_signal<int> data_in;
6 module the_module("the_module");
```

La ligne du code 3.14 présente la manière de lier les signaux de deux modules de différents simulateurs.

Listing 3.14 – Liaison des signaux ESyS.Net et SystemC

```
1 sym->adder->porta->Value=dataout1;
```

L'expression de gauche est un signal d'un module d'ESyS.Net et « dataout1 » est un signal de SystemC. La dernière étape pour co-simuler deux modèles est montrée par le code 3.15.

Cette technique utilise comme les autres des fonctions pour l'initialisation et l'incrémentation du temps discret. Elle ne nécessite aucune fonction pour relier ensemble des entrées et des sorties des modèles provenant de SystemC et d'ESyS.Net. L'avantage de cette technique est de permettre à l'usager de lier des modèles entre deux simulateurs sans avoir à créer une structure. Le principe de fonctionnement est semblable aux autres techniques. Dans le cas de la technique du COM et de la fonction statique, nous adaptons ESyS.Net, tandis qu'avec le Pinvoke et l'adaptateur géré nous adaptons SystemC.

Listing 3.15 – Simulation d'ESyS.Net et de SystemC avec l'adaptateur géré

```
1  #pragma managed
2  int _tmain(int argc, _TCHAR* argv[])
3  {
4  int i=0;
5  int count=0;
6  Simulator *sim;
7  MySystem *sym;
8  SystemCWrapper *wsystemc;
9  sim = new Simulator();
10 sym = new MySystem(sim,"top");
11 wsystemc = new SystemCWrapper;
12 sim->AssembleModel(); //Construction du modèle ESyS.Net
13 wsystemc->Init();//Initialisation de SystemC
14 for(i=0;i<=100;i++)
15 {
16     sym->adder->porta->Value=data_out1;
17     sym->adder->portb->Value=data_out2;
18     sim->Step(1);
19     clk=1;
20     wsystemc->SC_DUT_Stimulate(50);
21     clk=0;
22     wsystemc->SC_DUT_Stimulate(50);
23     data_in=sym->adder->portc->Value;
24 };
```

3.2 Comparaison des méthodes de co-simulation

Pour notre expérimentation, nous avons employé une version synchronisée d'un simple bus [Hilderink and Grötker, 2002] comme illustré par la figure 3.6, avec des modules maîtres et esclaves.

Chaque maître peut échanger des données en utilisant un canal. Des maîtres sont modélisés avec ESyS.Net sauf un qui est modélisé avec SystemC. Le maître 1 est un maître bloquant pouvant verrouiller le bus. Le maître 2 a un accès non bloquant au bus, alors que le maître 3 a un accès direct. Nous avons deux esclaves qui sont des mémoires, le premier n'a aucun état d'attente et le second est programmable avec un état d'attente. Un arbitre contrôle l'accès au bus. Un adaptateur ESyS.Net traduit un canal de commu-

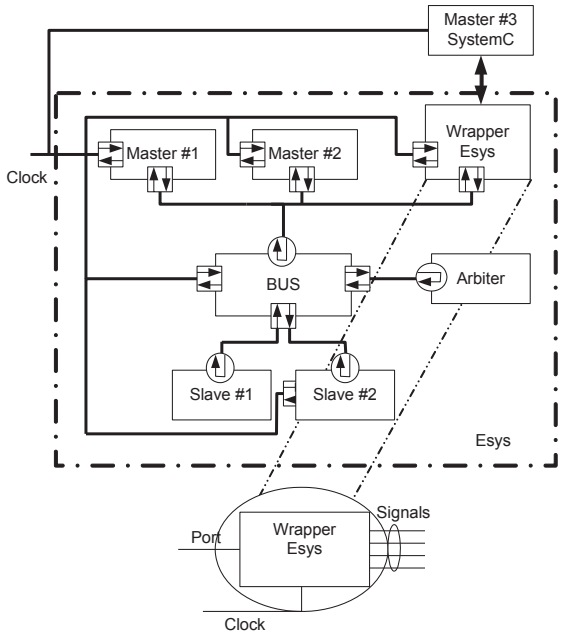


Figure 3.6 – Simple bus pour une co-simulation entre SystemC et ESyS.Net [Dubois and Aboulhamid, 2005]

nication en un ensemble de transactions pour les cycles d'horloge. L'adaptateur récupère les données du bus en utilisant une fonction bus_port.diect_read et le donne au maître modélisé en SystemC.

Nous avons utilisé les différents mécanismes présentés antérieurement soit la mémoire partagée et TCP/IP qui sont utilisés normalement pour effectuer une co-simulation. Nos techniques basées sur le COM et le Pinvoke et MW avec la FCT sont utilisés pour co-simuler ces modules simple bus [Dubois and Aboulhamid, 2005] en comparaison avec les méthodes traditionnelles. L'expérimentation a été exécutée sur un ACER Aspire 5715Z dual cpu 1.86GHz avec 2.0Go de mémoire. Le temps de co-simulation est illustré au tableau 3.1. Nous utilisons uniquement le critère du temps pour effectuer nos mesures permettant d'avoir la co-simulation la plus rapide.

Les meilleures approches sont la mémoire partagée, COM, TCP/IP et Pinvoke. Les pires cas sont MW et la fonction statique (FCT). Leurs vitesses dépendantes de l'environnement d'exécution. Dans tous les cas, les deux simulateurs incrémentent leur temps discret l'un après l'autre. La mémoire partagée utilise les évènements du système d'exploitation pour se synchroniser entre les simulateurs et s'initialise en utilisant le Pinvoke. La vitesse de simulation de la mémoire partagée et celle du COM sont plus rapides que celle de Pinvoke. La durée d'exécution de la technique du COM et celle de la mémoire partagée sont relativement semblables. Le TCP/IP est exécuté sur la même machine en mode local. Il est généralement un peu plus lent que la mémoire partagée, le COM et Pinvoke. Les simulations dans les mêmes fichiers comme le MW et le FCT ne semblent pas être la manière la plus rapide de simuler un modèle hétérogène. MW est relative-

Tableau 3.1 – Temps de co-simulation

Techniques	Temps (ms)
Mémoire partagée	150
TCP/IP	230
COM	100
Pinvoke	350
Adaptateur géré (MW)	1 030
Fonction statique	1 300

ment plus rapide que FCT. La FCT requiert plus d'appels de fonction que les autres techniques. Il semble aussi que du code géré appelant du code non-géré et l'inverse soit plus lent que les techniques d'interopérabilité du système d'exploitation. La mémoire partagée, le COM, le TCP/IP et Pinvoke sont optimisés au niveau du système exploitation, tandis que la compilation mixte entre du code géré et non-géré pour la FCT et MW sont optimisés pour l'environnement de compilation et non au niveau du système exploitation. Le MW et FCT sont comparables pour une compilation mixte entre du code géré et non-géré.

3.3 L'hybride syntaxique entre ESyS.Net et SystemC

Cette section discute de la possibilité de modifier du code SystemC (C++) dans un environnement ESyS.Net. Les codes 3.16 et 3.17 illustrent le propos avec du code C++ géré et non-géré. Nous retrouvons SC_MODULE, SC_CTOR et SC_METHOD représentant des macros SystemC. Nous utilisons des macros avec du code C++ géré. « EventList » est la même que la syntaxe d'ESyS.Net et aussi la manière d'interconnexion des signaux. Le « signal<int> » est un objet d'ESyS.Net remplaçant l'objet de SystemC. Il faut cependant ajouter le symbole accent circonflexe pour utiliser le code C++ géré. La liste de sensibilité est décrite par le mot « sensitive » pour SystemC, tandis que c'est « EventList » pour ESyS.Net. Ce code peut être exécuté en utilisant une variante d'ESyS.Net permettant l'enregistrement des processus de SystemC dans le noyau d'ESyS.Net par l'utilisation des dictionnaires tout en conservant les processus des modèles d'ESyS.Net.

Listing 3.16 – Exemple d'un module additionneur écrit dans du C++ géré

```
1
2  SC_MODULE(Adder)
3  {
4  public :
5  Signal<int>^ porta;
6  Signal<int>^ portb;
7  Signal<int>^ portc;
8  SC_CTOR(Adder){}
9  private:
10 SC_METHOD(run);
11 [EventList("sensitive",{"porta","portb"})]
12 void run(void)
13 {
14     portc->Value = porta->Value+portb->Value;
15 }};
```

Listing 3.17 – Exemple d'un module additionneur écrit dans du C++ non-géré

```
1
2  SC_MODULE(Adder)
3  {
4  Signal<int> porta;
5  Signal<int> portb;
6  Signal<int> portc;
7
8  void run(void)
9  {
10     portc = porta+portb;
11 }
12
13 SC_CTOR(Adder){
14     SC_METHOD(run);
15     sensitive << porta << portb;
16     }
17 };
```

La figure 3.7 montre l'interaction entre des modèles ESyS.Net et SystemC pouvant être exécutée avec ESyS.Net.

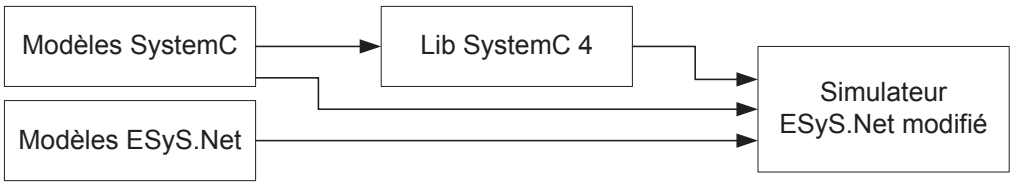


Figure 3.7 – ESyS.Net et SystemC

Il est impossible avec le C++ géré de cacher toute la syntaxe C++ non-gérée en utilisant une classe entre SystemC et ESyS.Net. Nous avons réalisé une variante de SystemC et d'ESyS.Net avec une modification de son noyau. Les modules SystemC hériteront d'une classe en C++ géré permettant son intégration dans ESyS.Net.

Par exemple, nous avons pris un consommateur, un producteur, et un FIFO. Le consommateur est écrit avec une variante de SystemC au code 3.18 et le producteur avec le FIFO est écrit en ESyS.Net au code 3.19.

Listing 3.18 – Exemple d'un module consommateur en C++ géré

```
1  ref class consumer : public sc_module
2  {
3  public:
4  read_if ^in;
5  SC_HAS_PROCESS(consumer);
6  consumer(sc_module_name name) : sc_module(name)
7  {SC_THREAD("main");}
8  void main()
9        { char c;
10 while (true) {
11       c = in->read();
12       cout << c << flush;}}};
```

Nous pouvons constater que les modules ESyS.Net n'ont aucun changement de syntaxe et que SystemC a quelques changements de syntaxe. La différence réside dans la déclaration du code C++ géré. Nous ajoutons des accents circonflexes pour l'interface « read_if », la déclaration « SC_THREAD » avec les guillemets et l'ajout du mot « ref » devant la classe.

Listing 3.19 – Exemple d'un module producteur avec ESyS.Net

```
1 public class producer : BaseModule
2 {public write_if ecriture;
3 [ThreadProcess]
4 private void run()
5 {
6 char c; int cnt = 0;
7 string str = "Visit www.systemc.org and see what SystemC can do
      for you today!";
8 while (cnt < str.Length)
9 {ecriture.write(str[cnt]);cnt++;}}}
```

L'objectif n'est pas de mesurer la vitesse de co-simulation entre SystemC et ESyS.Net, mais plutôt d'éliminer un noyau de simulation. Dans le cas d'ESyS.Net, sa vitesse d'exécution est plus lente que celle de SystemC. Donc, un noyau qui regroupe deux langages hôtes (C++ et C#) a une durée de simulation plus longue que s'il avait été dans le même hôte (C++).

Dans le cas de la variante d'ESyS.Net, il est possible d'avoir une co-simulation avec SystemC en modifiant légèrement le code C++ non-géré. Un avantage de la variante de SystemC est la possibilité d'utiliser la réflexion et l'introspection disponible à l'aide du C++ géré. Nous avons deux noyaux SystemC et ESyS.Net qui ont été remplacés par un nouveau noyau de simulation d'ESyS.Net. La modification se fait au niveau de la déclaration des signaux dans SystemC. Le compromis fut de réécrire un peu ESyS.Net et d'avoir une syntaxe différente de SystemC, mais la syntaxe d'ESyS.Net reste identique.

Dans ce chapitre, nous avons vu qu'il est possible d'utiliser d'autres techniques interactions entre les langages dans le but de modéliser un système hétérogène. Nous nous sommes concentrés à utiliser des techniques existantes pour l'interconnexion entre les processus de divers langages. Notre objectif était de savoir comment adapter celles-ci dans le cadre d'une co-simulation d'un système en reliant plusieurs noyaux de simulation. Nous avons aussi montré qu'il est possible d'enlever un noyau de simulation avec un changement syntaxique.

Le prochain chapitre présente une méthodologie pour co-simuler sans avoir une limitation syntaxique. De plus, il présentera une accélération de la simulation.

CHAPITRE 4

DESCRIPTION ET COMPILATION HÉTÉROGÈNE

Nous proposons une nouvelle approche pour simuler des modèles décrits dans divers langages. Notre objectif est de développer un environnement multi-langage où il y a une séparation entre le modèle et la syntaxe utilisée par l'usager. L'utilisateur peut utiliser des langages comme SystemC, ou l'architecture .NET basée sur les langages (C#, J#, ...) en utilisant l'environnement de conception ESyS.Net. Le choix d'ESyS.Net permet d'avoir un simulateur avec du code géré, mais, pour notre approche, nous pouvons utiliser d'autres simulateurs numériques. L'usager choisi ses langages hôtes favoris pour décrire sa modélisation d'un système dans les niveaux d'abstractions supportés comme le RTL et le TLM. De plus, les langages utilisés lors d'une co-simulation seront compilés de manière uniforme sans être obligé de créer des structures et des mécanismes pour lier les hôtes par l'usager.

L'idée principale de compilation est de décomposer les modèles en fonctions, processus et attributs décrivant une fonctionnalité dans une représentation interne homogène. Nous proposons deux principaux types de simulations compilées : la première est le RTL et le second, TLM. Le RTL compilé est déjà connu dans la littérature, mais le TLM compilé est plus difficile à réaliser.

L'environnement de compilation utilise des objets internes pour créer un nouveau code unifié, permettant d'utiliser un seul noyau au lieu de plusieurs. L'ajout de mécanismes de communication entre les langages hôtes augmente la durée d'une co-simulation. Dans un même temps, il est possible d'effacer l'interface entre les modules en fusionnant les canaux et les modules. Les processus et méthodes d'un objet interne peuvent être groupés et s'exécuter en un seul processus au lieu de plusieurs. Les nouveaux processus peuvent aussi être divisés pour être exécutés sur plusieurs ressources informatiques, dans le but de diminuer la durée d'une simulation.

Chaque simulateur a son propre noyau et peut être décrit en utilisant différents langages. Un noyau est créé dans le même langage que le modèle à simuler comme

ESyS.Net pour C# et SystemC pour C++. Une étroite synchronisation doit être réalisée par un bus de co-simulation comme interface entre les différents noyaux [Bois et al., 2003].

La figure 4.1 présente cette nouvelle approche de simulation : les modèles sont transformés en utilisant notre compilateur pour une simulation distribué ou centralisé.

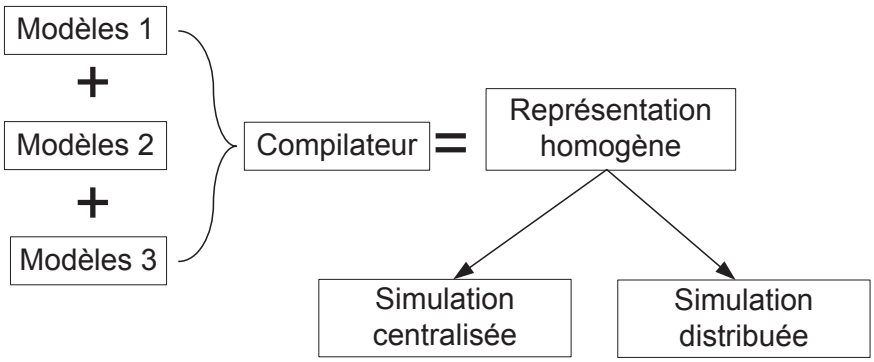


Figure 4.1 – Une nouvelle approche pour simuler un modèle hétérogène.

Les objectifs pour le concepteur peuvent être la modularité et la réutilisation. D'un autre côté, les buts de simulation sont particulièrement la performance, la distribution, l'enlèvement des détails qui ne sont pas nécessaires pour un niveau d'abstraction donnée, l'exactitude et l'indépendance de la syntaxe d'un multi-langage pour une description initiale. La clef de la réussite pour une simulation rapide et efficace est la séparation entre les modèles et la simulation. Nous avons publié ce chapitre dans [Dubois et al., 2006, 2007, 2009].

4.1 Chemin de Simulation

Un modèle décrivant un système contient principalement des processus, des signaux, des ports, des canaux et des interfaces. Deux niveaux principaux d'abstraction sont généralement considérés : une modélisation au niveau des transactions (TLM) à un haut niveau d'abstraction et une modélisation au niveau de transfert de registre (RTL). Dans le TLM, la communication entre les modules est principalement réalisée par des canaux modélisés à un haut niveau d'abstraction, en particulier des FIFOs, des interconnexions

spécifiques ou un protocole de communication. Au niveau RTL, des interconnexions sont décrites au niveau des signaux et le comportement du système est au niveau cycle. La simulation est naturellement beaucoup plus coûteuse au niveau RTL comparé au TLM. On a principalement deux approches pour simuler des systèmes de RTL : simulation entraînée par les événements et compilée [Dubois and Aboulhamid, 2005][Jennings, 1991].

4.2 Les défis et les difficultés

Il n'est pas facile de fusionner, partitionner et recomposer des processus, ainsi que préserver la sémantique pour diminuer la durée d'une simulation. Tandis que la simulation compilée des descriptions de RTL a été entreprise dans le passé, nous savons qu'il n'y a pas eu beaucoup de tentatives pour obtenir une co-simulation compilée des multi-descriptions du TLM. Le compilateur doit déterminer quels processus peuvent être ensemble et comment élaborer automatiquement des interfaces entre des modules. Une interface peut servir de lien entre un module et un canal. Elle définit d'une manière abstraite les opérations de l'échange de données qu'un canal devrait mettre en application.

Par exemple, 50 processus doivent être séquentialisés dans un seul processus. Chacun d'eux émet des événements et aussi peuvent attendre d'autres événements. La synchronisation des processus est importante. Il faut éviter d'avoir des interblocages entre eux. Chaque processus doit être exécuté pour qu'il n'y ait pas de blocage entre deux processus. C'est-à-dire qu'un processus attend après un autre et ainsi de suite. La simulation va être bloquée quand nous avons un blocage.

4.3 Première version d'un modèle de compilation basé sur AST

Le modèle de compilation et de simulation proposé illustré à la figure 4.2 comporte trois principales phases : génération d'arbres, ordonnancement et analyses de dépendance et la génération de code.

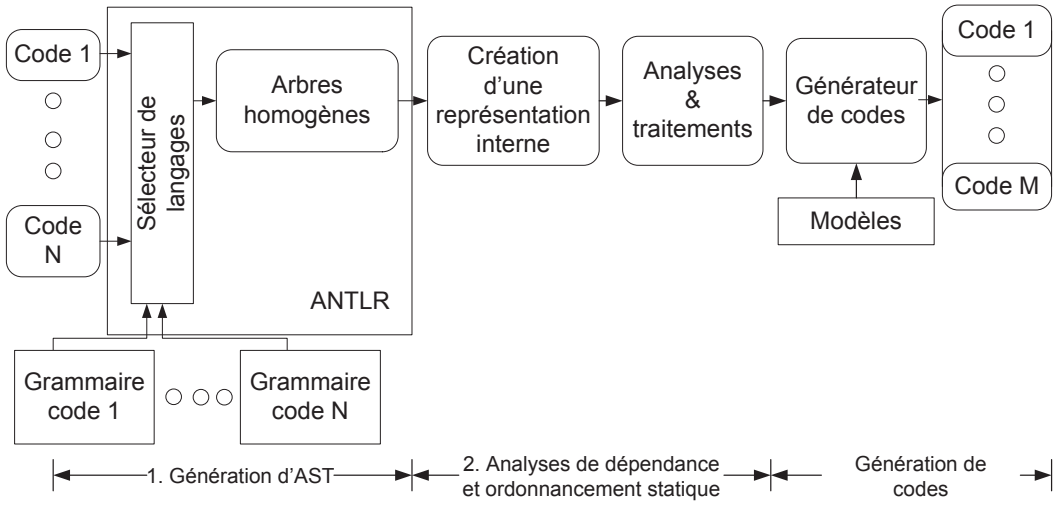


Figure 4.2 – Architecture proposée

4.3.1 Génération des arbres syntaxiques

Dans la phase 1, à partir des modèles hétérogènes initiaux et les grammaires des différents langages, des « abstract syntax trees (AST) » sont créés en utilisant ANTLR [Fischer et al., 2007; Parr and Quong, 1995]. Nous pouvons avoir la même représentation d'arbre pour chaque langage dans des spécifications initiales. La syntaxe est différente selon le langage hôte utilisé et constituent les défis principaux de cette phase. Une fois ces difficultés surmontées, nous obtenons une vue homogène à travers les ASTs. ANTLR permet la construction d'arbre hétérogène défini par l'usager. Nous définissons la construction d'un arbre commun. L'objectif est de transformer chaque arbre hétérogène en un arbre homogène. Ensuite, il suffit de mettre tous les arbres homogènes ensemblent. Cette transformation peut être laborieuse lorsqu'on ajoute plusieurs langages. Pour simplifier notre, tâche nous supportons un sous-ensemble syntaxique des langages hôtes pour RTL et TLM.

4.3.2 Analyse de dépendance et ordonnancement

Dans la phase 2, une représentation interne est produite des ASTs et des graphes de dépendance de contrôles et de données entre les lignes de code sont créés, au besoin.

Quelques codes peuvent être statiquement programmés selon le graphe de dépendance et chaque ligne de code est prête pour passer à la phase 3 pour la génération de code. D'autres codes peuvent être ajoutés dynamiquement et sont traités dans la phase 3.

La représentation interne est une nouvelle classe avec des champs comme le nom d'une méthode, du module, du type et des listes de sensibilité. Elle contient également des dictionnaires au sujet d'interconnexions entre les modules et autres. Elle nous isole complètement de la description initiale de plusieurs langages. L'analyse crée un certain nombre de graphes de la représentation interne comme l'interdépendance de graphes pour l'interface, les logiques combinatoires et les informations sur des types. L'analyse se décompose en trois étapes. (a) Tous les modules et signaux correspondant au niveau RT sont recherchés (Base Module, Signal<int>, ...). (b) Des événements et les interfaces utilisés dans une représentation de TLM sont extraits. (c) Les relations entrent les signaux, les interfaces, les modules et ainsi de suite sont établis. Après analyse, toutes les informations sur des modèles et leur interaction sont connues ; nous pouvons maintenant les traiter. Le traitement se compose du partitionnement, du regroupement ou de la division des processus et de la réorganisation des lignes de codes, avant de les envoyer à la phase 3 pour la génération de code.

4.3.3 Génération de codes avec templates

La phase 3 produit un code vers la simulation efficace. Pour quelques cas spécifiques, par exemple, une simulation au niveau cycle, le noyau est complètement vide et le modèle est transformé entièrement dans un programme séquentiel plat qui contient une boucle. Une itération de la boucle représente un cycle d'horloge. Cette phase utilise le noyau String Template (ST) [Parr, 2006]. C'est un générateur de code qui emploie une séparation du modèle-vue [Parr, 2004]. Il permet de créer des espaces récurrents ou pas, pour que l'utilisateur soit libre de les remplir selon son application en utilisant des attributs. Nous le verrons dans une section ultérieure, nous avons un modèle pour chaque niveau d'abstraction : TLM, ou RTL. Des modèles spécifiques pour le regroupement, la séparation et la distribution des processus sont employés. À notre connaissance, c'est une manière originale de transformer automatiquement une première description

afin d'effectuer une simulation efficace.

4.4 Deuxième version proposée d'un modèle de compilation basé sur le XML

La première version du modèle de compilation basée sur les ASTs est limitée à la création des ASTs et il est plus difficile de traiter les ASTs venant de plusieurs langages. De plus, des outils gratuits de transformation de code dans une structure AST sont très limités. Pour pousser la limite d'interopérabilité et de portabilité, nous présentons une version d'un modèle de compilation basé sur le XML et modifions aussi les parties de traitements qui étaient basées sur les ASTs. Le problème est que la séparation du code et des ASTs sont assez limités. La figure 4.3 présente une nouvelle version avec une plus grande flexibilité.

Nous avons encore trois phases. La première est la génération XML, la deuxième est l'analyse et le traitement et la dernière phase est la génération de code.

4.4.1 XML génération

La première phase consiste à générer du XML à partir des langages. Il existe plusieurs outils gratuits pour en générer. Nous pouvons trouver des convertisseurs gratuits XML pour le VHDL, C++, C et Java avec des limitations qui sont normalement dues aux créateurs. C'est-à-dire que les créateurs réalisent des outils selon leurs applications et leurs besoins. Pour notre recherche, nous avons réalisé un C# XML à partir d'une structure en C#. En utilisant le code générateur de la troisième partie (ST), nous avons traversé l'arbre du parseur C# pour avoir un format XML à l'aide des patrons visiteurs. Contrairement au AST, nous voulons totalement dissocier la syntaxe du modèle. Un langage peut être représenté en forme d'arbre. Un arbre AST est contraignant et non portable. Évidemment, nous pouvons mettre un arbre AST sous forme XML. L'architecture proposée se limitera à deux modèles RTL et TLM dans le cadre de cette recherche.

Nous pouvons représenter un système hétérogène par différents langages de modélisation pour le niveau RTL et TLM. Pour chaque langage supporté, nous pouvons représenter un XML uniforme à partir des XML hétérogènes. Des outils de transfor-

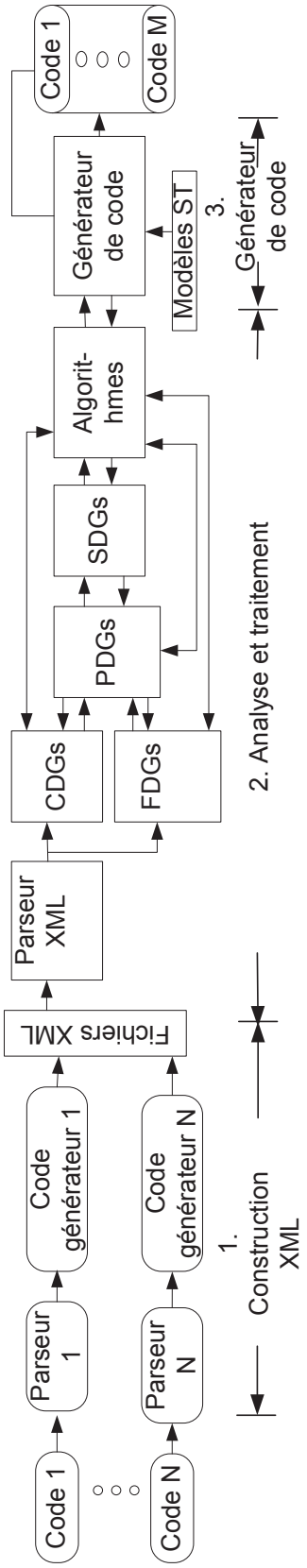


Figure 4.3 – Architecture modifiée proposée

mations XML à XML sont requis. Un « Document Type Definition » (DTD) est créé pour connaitre la compatibilité entre le XML généré par d'autres outils et un XML de référence [Abdel-aziz and Oakasha, 2005].

La plupart des générateurs XML, à partir du code comme C# et C++, contiennent un DTD. Il faut définir un DTD unique et créer une transformation XML à partir du DTD des générateurs de codes pour avoir un XML qui est défini avec un DTD uniforme.

4.4.2 Analyse et traitement

La meilleure façon de représenter un code source demeure sous forme de graphes. Il est possible d'effectuer des transformations lorsqu'on parcourt un AST, mais cette méthode est plutôt limitée. Nous allons introduire 4 types de graphes permettant une plus grande flexibilité soit control dependence graph (CDG), flow dependence graph (FDG), program dependence graph (PDG) et system dependence graph (SDG). Ces graphes sont très utiles pour plusieurs applications. Par exemple, ils peuvent être utilisés pour l'extraction d'un code (slice) en VHDL [Clarke et al., 2002; Russell, 2002]. Le slice permet d'obtenir une partie de code qui peut être analysée et transformée au besoin.

Les graphes peuvent être divisés et groupés. Pour la suite de l'explication, nous utiliserons une application producteur et consommateur pour illustrer le fonctionnement des graphes tel qu'illustré à la figure 4.4. Les codes 4.1,4.2 et 4.3 sont respectivement le FIFO, le producteur et le consommateur. Nous avons deux modules B et C échangeant des données en utilisant des interfaces à travers des canaux FIFO. B est un producteur et C est un consommateur. C'est un exemple typique pour une modélisation transactionnelle.

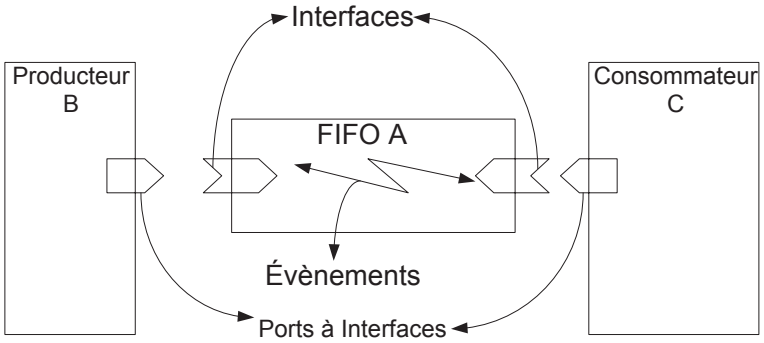


Figure 4.4 – Producteur et consommateur utilisant un FIFO

Listing 4.1 – Classe FIFO

```
23 public class fifo : BaseChannel, MyInOut
24 {
25 private int max = 10;
26 private char[] data = new char[10];
27 private int num_elements;
28 private int first;
29 private Event write_event = new Event();
30 private Event read_event = new Event();
31
32 public void write(char c)
33 {
34 if (num_elements == max)
35       Wait(read_event);
36 data[(first + num_elements) % max] = c;
37 ++num_elements;
38 write_event.Notify(0);
39 }
40
41 public void read(ref char c)
42 {
43 if (num_elements <= 0)
44       {Wait(write_event);}
45 c = data[first];
46 −−num_elements;
47 first = (first + 1) % max;
48 read_event.Notify(0);
49 }
```

Listing 4.2 – Classe producteur

```
59 public class producer : BaseModule
60 {
61 public write_if ecriture;
62
63 [ThreadProcess]
64 private void run()
65 {
66 int cnt = 0;
67 string str = "Visit www.systemc.org and see what SystemC can do
       for you today!";
68
69 while (cnt < str.Length)
70 {
71 ecriture.write(str[cnt]);
72 cnt++;}}}
```

Listing 4.3 – Classe consommateur

```
79 public class consumer : BaseModule
80 {
81 public read_if lecture;
82
83 [ThreadProcess]
84 private void run()
85 {
86 char c = 'r';
87 while (true)
88 {
89 lecture.read(ref c);
90 Console.Write(c);
91 if (lecture.num_available() == 1)
92 Console.Write("<1>");
93 if (lecture.num_available() == 9)
94 Console.Write("<9>");}}}
```

Dans les figures 4.5a à 4.7, les graphes contiennent des noeuds représentant des expressions du code. Ces expressions sont : if(diamond), while(double circle), function(3d box), process(parallelogram), return(pentagon), class fields (box), call(triple octagon), other(ellipse). Par exemple, le code « if » est représenté par la forme du diament en utilisant le terme « diamond ».

4.4.2.1 Graphe de dépendances de contrôles

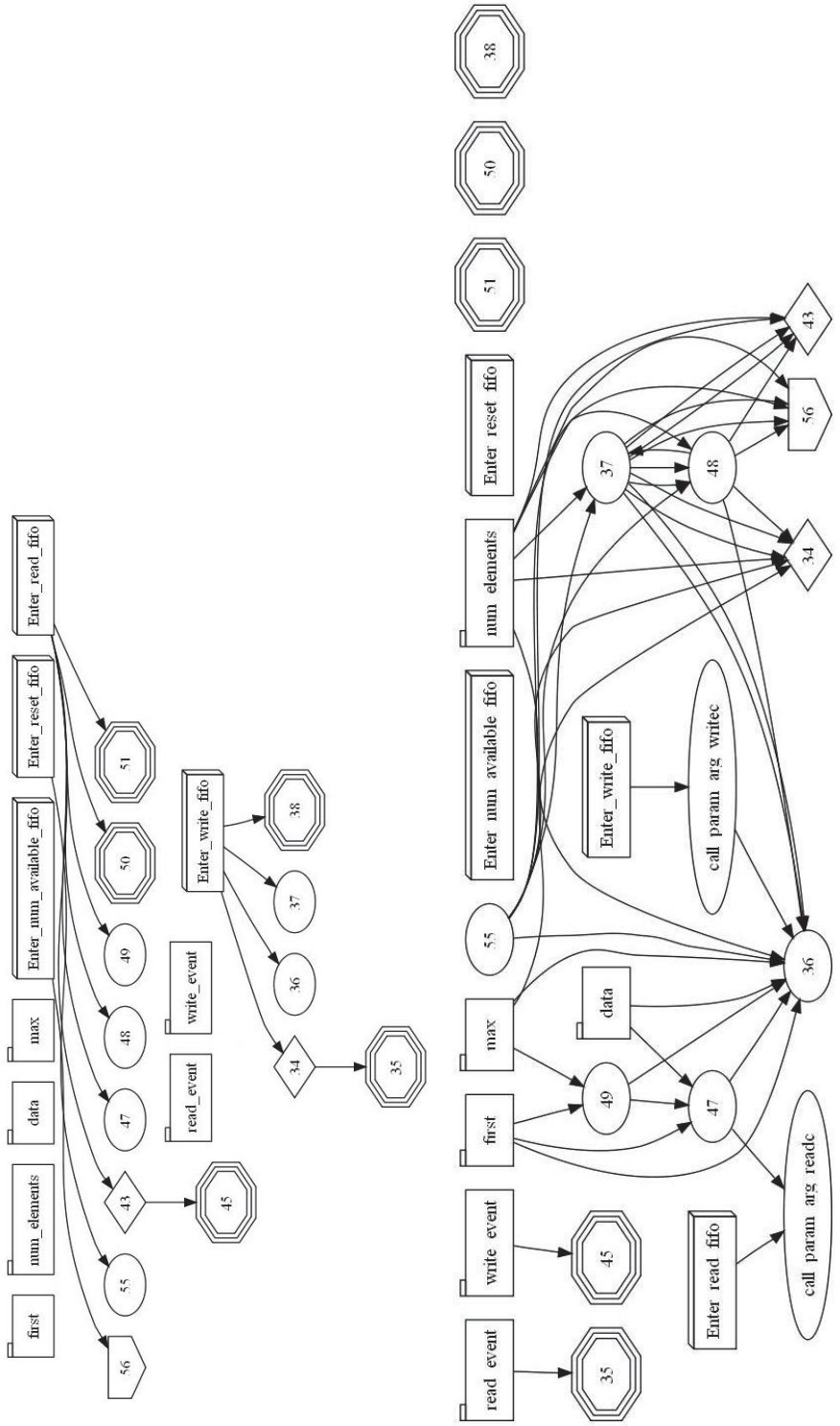
La première étape est de construire un graphe de dépendances de contrôles (CDG) pour chaque méthode ou processus dans le modèle [Han and Chen, 2010]. Les noeuds dans un CDG correspondent aux expressions du code. Un arc à partir d'une expression p à une expression q existe, si l'exécution de q dépend directement de l'exécution de p. Par exemple, dans le code de la méthode « write » de la classe « fifo », la ligne 35 dépend de l'expression conditionnelle de la ligne 34 et toutes les expressions (34, 36, 37 et 38) de cette méthode dépendent de l'entrée de la méthode. Notez que les noeuds 35 et 38 sont des appels aux méthodes d'un simulateur (figure 4.5a).

4.4.2.2 Graphe de dépendances de données

Le graphe de dépendances de données (FDG) décrit le chemin de données dans une classe [Kavi et al., 1986]. La figure 4.5b montre le FDG de la classe FIFO [Dubois et al., 2009]. Par exemple, nous avons une flèche du noeud 37 au noeud 34 parce que 37 modifie la variable « numelements » et cette variable est employée par le noeud 34. Dans le FDG, un noeud d'argument est créé pour chaque argument d'une méthode.

4.4.2.3 Graphe de dépendances de programmes

Le graphe de dépendances de programmes (PDG) représente l'union des FDGs et des CDGs (figure 4.6) [Canfora and Cimitile, 1995]. Un graphe contenant la dépendance du code sur les données et sur les contrôles est très utile. Par exemple, nous pouvons connaitre les autres noeuds contrôlant et affectant un noeud.



(a) Le CDG (b) Le FDG

Figure 4.5 – Graphes pour un FIFO [Dubois et al., 2009]

Dans notre cas, nous voulons regrouper les noeuds de contrôles pour regrouper les processus en un seul et connaitre les données qui seront nécessaires pour initialiser notre code séquentiel.

4.4.2.4 Graphe de dépendances de systèmes

Le graphe de dépendances de systèmes (SDG) peut être vu comme une union de tous les PDGs des classes d'un modèle incluant les interactions d'inter classe et d'intra classe [Livadas and Croll, 1993]. À la compilation, le SDG aide à identifier l'interdépendance entre les processus et les fonctions. Après la lecture de la déclaration d'un système tel que le FIFO, le producteur et le consommateur, il est nécessaire de les lier. Dans ce graphe, des arcs sont ajoutés entre les fonctions appelées et appelantes. Il permet de représenter toutes les relations entre les classes, les fonctions et les interfaces. Par exemple, l'appel de la fonction « write » dans le processus du producteur est lié à la fonction d'écriture du FIFO. Les codes 4.2 et 4.3 montrent le producteur et le consommateur correspondant au SDG dans la figure 4.7. Il faut noter que nous n'avons pas inclus dans le code présenté tous les noeuds. Notre compilateur ajoute au début du noeud « Enter_ » et à la fin le nom de la classe pour chaque fonction. Par exemple, la fonction « write » de la classe FIFO est écrite dans le noeud « Enter_write_fifo ». Nous pouvons voir trois sous graphes avec trois noeuds commençant par « Enter_ ». Nous avons le producteur, le consommateur et la FIFO qui sont reliés. De plus, nous pouvons voir directement les appels du producteur dans la FIFO par une interface. De la même manière, nous avons une interface pour le consommateur avec « call_interface ». Le SDG permet d'avoir la relation entre les modules (producteur et consommateur) qui interagit avec une interface FIFO. Les prochaines sections traitent des modèles RTL et TLM.

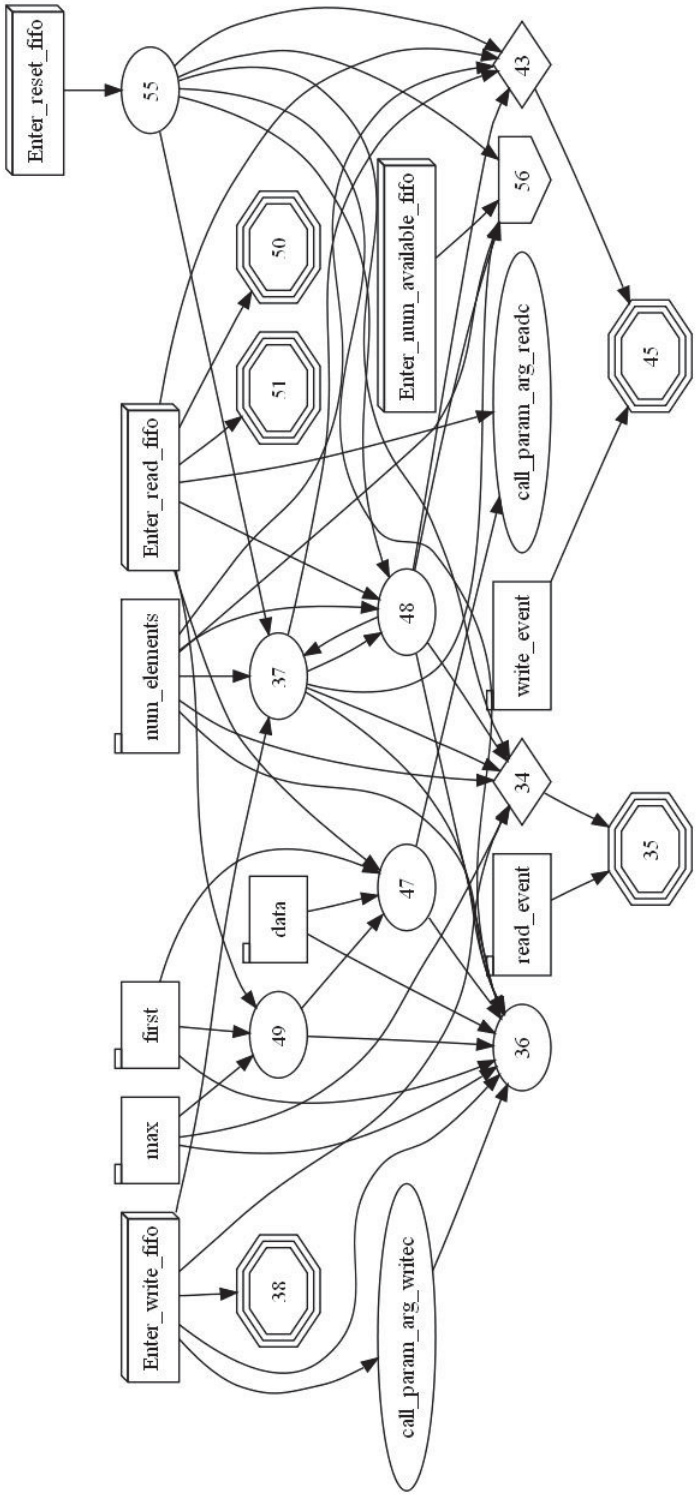


Figure 4.6 – Graphe d'un PDG [Dubois et al., 2009]

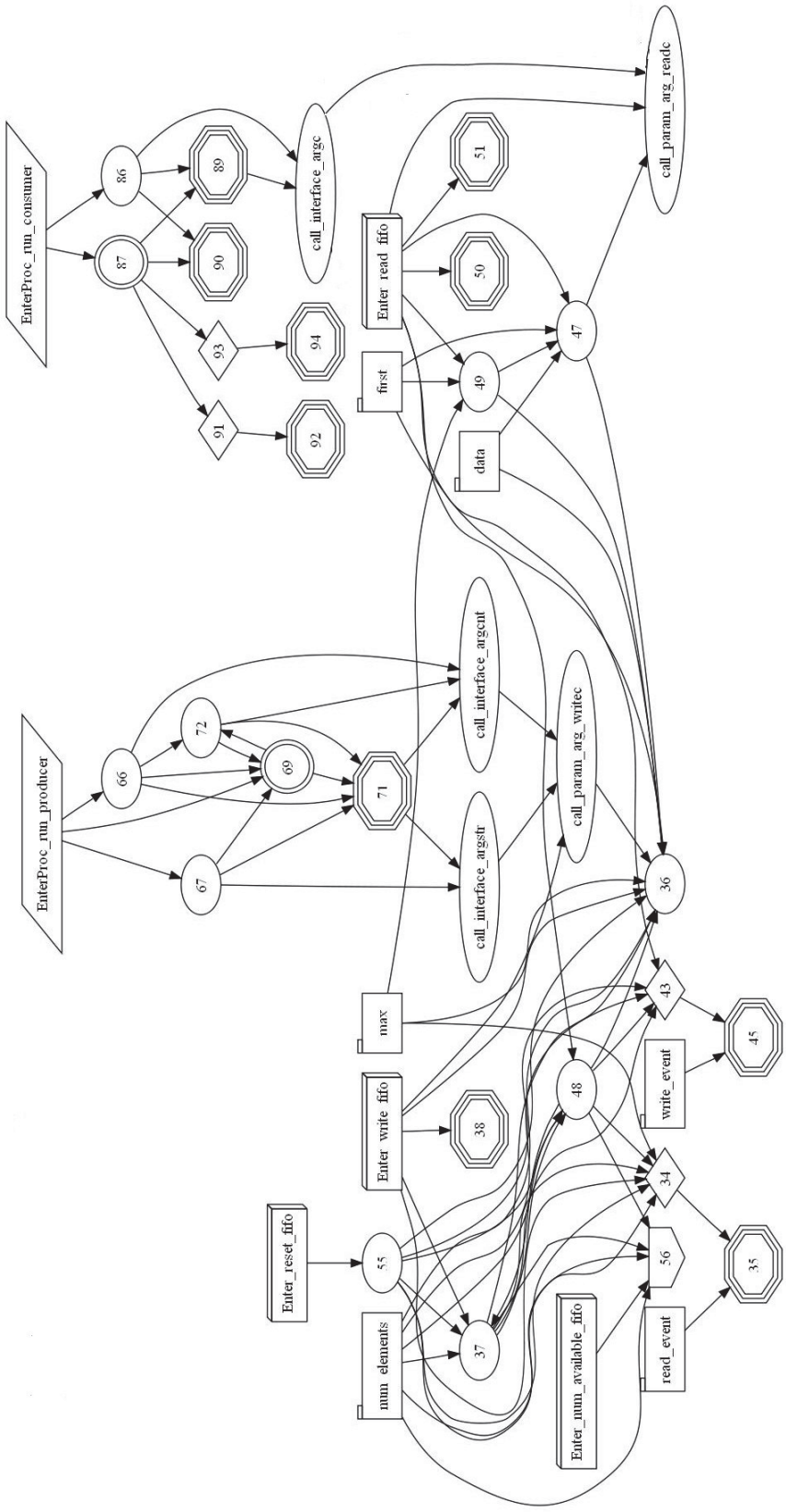


Figure 4.7 – Graphe d'un SDG [Dubois et al., 2009]

4.5 RTL

La figure 4.8 présente un exemple simple d'un modèle RTL. Un compteur d'horloge est représenté par « Feeder ». Les deux autres modules « ADD » et « MULT » sont combinatoires.

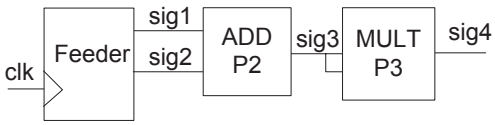


Figure 4.8 – Exemple d'un modèle RTL

Tous les signaux contrôlés par l'horloge sont considérés comme des registres et ils sont représentés dans le code par deux variables. À chaque top d'horloge, les deux variables sont incrémentées. « Sig1 » dans la figure 4.8 est représenté par « sig1.last » et « sig1.now ». Tous les autres signaux sont représentés par une simple variable (i.e., sig3).

La méthode de la figure 4.9 est utilisée par la phase de génération de code pour générer le code de la partie RTL pour n'importe quel modèle. Cette méthode est définie dans [Jennings, 1991].

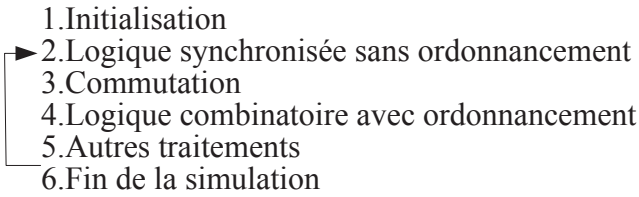


Figure 4.9 – Méthode pour simuler une description RTL

Dans le champ 1, le code est généré pour initialiser toutes les variables représentant tous les signaux. Les champs 2 à 5 correspondent à un cycle d'horloge. En 2, toutes les valeurs futures des registres sont calculées (i.e., sig1.now). En 3, les sorties des registres sont mises à jour (sig1.last = sig1.now, etc.). En 4, les logiques combinatoires sont mises à jour. Pendant qu'on remplit les champs, l'ordre est important, sig3 doit être mis à jour avant sig4. C'est un ordre qui est produit avec une politique d'ordonnancement statique

dans le compilateur selon le graphe de dépendance de données. Le champ 5 est réservé pour d'autres traitements comme le traçage. Le champ 6 contiendra le code en fin de simulation.

Le code 4.4 présente la compilation du modèle de la figure 4.8. Il est important de noter que tous les évènements ont été supprimés. L'ordonnancement statique permet de disposer le code en fonction des évènements. Une boucle émule le comportement de l'horloge et les signaux sont exécutés selon leur dépendance. Il en résulte d'une amélioration de la durée de la simulation. Il faut noter également que la vision du concepteur perçoit encore son modèle comme une hiérarchie de modules et une description d'évènements.

Listing 4.4 – Code C# créé par la compilation du modèle de 4.8

```
//Initialisation
feedi.last =1; sig1.last =0; sig2.last =0; sig3.last =0; sig4.last =0;
for (clk = 0; clk <= TimeLimit; clk++ ) {
//Logique synchrone sans ordonnancement
feedi.now = feedi.last+1; sig1.now = feedi.last; sig2.now = feedi.last;
//Commutation
feedi.last = feedi.now; sig1.last = sig1.now; sig2.last = sig2.now;
//Logique combinatoire avec ordonnancement
sig3 = sig1.last+sig2.last; sig4 = sig3*sig3;
//Usager
} Fin de la simulation
```

4.6 TLM

La génération de code automatique d'une description d'un modèle TLM est beaucoup plus complexe comparée au RTL. Une modélisation TLM utilise des interfaces et des canaux de communication à haut niveau qui nécessite plus d'analyse que le RTL. Un noyau de simulation pour une modélisation RTL utilise un graphe où il n'y a pas d'interface, mais des signaux. Il est plus facile de créer des graphes à partir des signaux et des interfaces génériques. L'idée principale concernant la compilation TLM est de décomposer le code en plusieurs zones qui peuvent être facilement réorganisées comme un patron d'un modèle prédéfini. Une liste de priorités est utilisée pour synchroniser les

zones. Pour expliquer le modèle de simulation, nous utilisons un producteur, un consommateur et un FIFO comme illustré à la figure 4.10. Le code 4.1 montre la FIFO, tandis que le producteur et le consommateur sont respectivement au code 4.2 et 4.3.

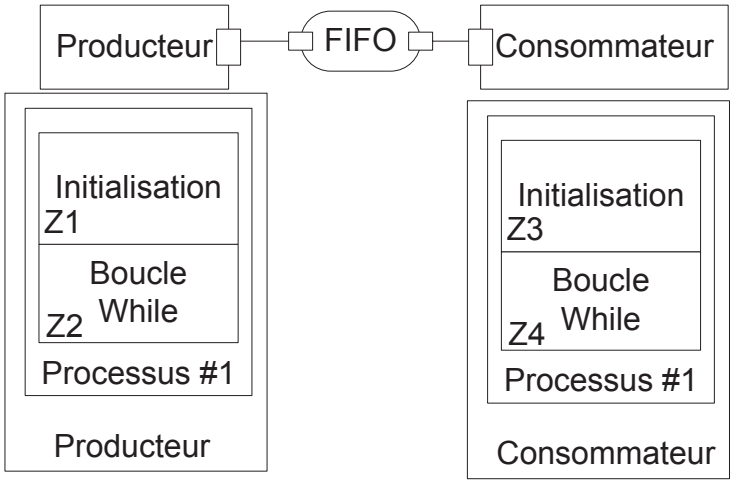


Figure 4.10 – Exemple pour un modèle TLM

Le producteur envoie des éléments aux consommateurs à travers un FIFO. Dans cet exemple, nous définissons 4 zones Z1 à Z4. Z1 et Z3 sont exécutées une fois parce qu'elles sont avant la boucle dans le processus. Chaque boucle sera assignée à une zone (Z2 et Z4). Chaque méthode dans le FIFO va être déplacée dans le processus appelant. L'objectif est de séquentialiser les processus en éliminant les interfaces entre eux. Une liste de priorité est utilisée pour ordonnancer dynamiquement les différentes zones à la ligne 2 du code 4.5. Au commencement de la simulation, une valeur spéciale correspondant à la plus grande priorité doit être insérée dans la liste pour indiquer la fin de la simulation. À chaque retrait à la ligne 10, la prochaine zone à être exécutée est déterminée. Si toutes les phases sont exécutées alors, le dernier élément dans la liste indiquera la fin de la simulation (cette valeur est utilisée à la ligne 15 en fin de simulation). La valeur à être insérée est définie par la relation entre un événement notifié et la zone appelant. Z2 et Z4 sont numérotées dans un « switch case » et peuvent être appelées en ajoutant leur numéro respectif dans la liste.

Le code compilé pour le TLM est complètement séquentiel. Tous les processus ont été regroupés et les événements, remplacés par un code conditionnel.

Listing 4.5 – Simulation compilé pour un modèle TLM

```
1  Déclaration de la logique interne
2  Création d'une liste de priorité
3  Copie des zones initiales dans la liste c.−à−d. Z2 et Z4 avec priorité 1, 2 (les priorités
       maximums).
4  Insertion d'un indicateur de fin de simulation avec la plus grande priorité
5  Z1
6  Z3
7  switch (boucle)
8  {
9        case 0: Insertion des évènements dans la liste
10             Enlève l'évènement de la liste ayant la plus grande priorité
11             switch (Zone à exécuter)
12             {
13             case 1: Z2
14             case 2: Z4
15             case 3: Fin de simulation ;
16             }
17             if (Fin de simulation) goto case 1;
18             goto case 0;
19       case 1: Sortir de la simulation;
20 }
```

Nous espérons une importante augmentation de la performance de simulation parce que le noyau d'ESyS.Net a été complètement enlevé. Il faut aussi noter que la précision de la simulation est restée la même et l'utilisateur voit encore son modèle comme deux modules connectés par un canal FIFO. Dans ce modèle, il n'y a pas de nécessité pour détecter un blocage. L'ajout des « breaks » que nous verrons en détail dans le prochain chapitre permet de ne pas être bloqué dans un processus et les évènements à exécuter sont dans la liste de priorité. Nous aurons toujours un processus à exécuter. Mais si plusieurs processus n'ont jamais de fin, ils s'exécuteront dynamiquement selon leurs niveaux de priorités. Dans cette version, les « waits » avec notion de temps ne sont pas supportés tel que « wait (10 ns) ».

Nous avons montré que nous pouvons utiliser une représentation interne basée sur le XML pour éliminer un bus de co-simulation. Nous avons également présenté une manière originale pour co-simuler un modèle TLM. Le modèle TLM utilise une transformation par zones, tandis qu'un modèle RTL a une transformation en programme séquentiel où une boucle correspond à un cycle d'horloge. Nous avons également montré qu'il est possible d'avoir une dissociation entre la vue de l'utilisateur pour décrire son système et l'exécution de notre compilateur.

CHAPITRE 5

RÉALISATION ET IMPLÉMENTATION

La réalisation de notre compilateur nécessite plusieurs modèles. Le premier modèle est la description d'un modèle interne et la deuxième est un modèle pour la génération de code. Nous supportons en entrée du compilateur les langages : SystemC (C++) et ESyS.Net (C#). Les codes générés par le compilateur sont : C++, C#, VHDL et Java. Tout d'abord, nous allons discuter des méthodes pour avoir une représentation homogène. Ensuite, nous expliquerons la réalisation et l'implémentation des modèles TLM et RTL avec des exemples. Pour terminer, nous comparerons les durées des simulations de plusieurs hôtes avec le TLM. La section 5.3.2 a été publié dans [Dubois et al., 2007] sauf l'exemple des routeurs.

5.1 La génération du modèle interne à partir d'une description hétérogène

La description interne est réalisée en XML par une structure prédéfinie. Le code XML généré par notre compilateur pour ESyS.Net (C#) correspond au modèle interne, car nous générons directement le modèle XML à partir de C#. Nous utilisons des générateurs de code XML pour SystemC et ESyS.Net. La structure entre les XMLs générés est différente, donc nous devons les rendre semblables. Pour illustrer le fonctionnement d'une transformation, nous utilisons SystemC et ESyS.Net. L'objectif est d'avoir un XML qui représente un modèle unique pour les langages supportés. De plus, nous supportons un sous-ensemble des langages C++ et C#. Les codes 5.1 et 5.2 sont respectivement une déclaration d'un canal dans ESyS.Net et SystemC. Nous voulons que le XML du SystemC soit identique à celle d'ESyS.Net. Nous utilisons un modèle de transformation illustré au code 5.3. Pour simplifier notre exemple, nous avons trois mots clefs soit « fifo », « access » et « class ». Nous voulons déclarer une classe publique nommée « fifo ».

Listing 5.1 – Extrait d'un code XML généré à partir de C#

```
1 <class name="fifo" access="public">
2 </class>
```

Listing 5.2 – Extrait d'un code XML généré à partir de SystemC

```
1 <class>
2     class <name> fifo</name>
3     <super>:
4         <specifier>public</specifier>
5     </super>
6 </class>
```

Nous construisons la structure du XML de C# en parcourant la structure de SystemC à l'aide d'une transformation « Extensible Stylesheet Language » (XSL) du code 5.3 [Kirda et al., 2001; Sal, 2005]. La ligne 3 parcourt chaque classe de SystemC. Les lignes 6 et 9 permettent d'extraire le nom et le type d'accès de la classe. La structure XSL correspond au modèle interne.

Listing 5.3 – Extrait d'un code XSL pour la transformation de SystemC en modèle interne

```
1  <?xml version='1.0' ?>
2  <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/
       Transform" xmlns:a="http://www.sdml.info/srcML/src">
3  <xsl:for-each select="a:unit/a:class">
4              <class>
5                  <xsl:attribute name="name">
6                      <xsl:value-of select="a:name"/>
7                  </xsl:attribute>
8                      <xsl:attribute name="access">
9                  <xsl:value-of select="substring-after(a:super/
                       a:specifier,'virtual')"/>
10                 </xsl:attribute>
11             </class>
12         </xsl:for-each>
13     </xsl:template>
14 </xsl:stylesheet>
```

Une fois que le modèle interne est construit, nous utilisons un autre modèle pour générer le code comme illustré au code 5.4. Dans notre exemple, la classe peut être générée en C# à l'aide du moteur de génération de code « String Template (ST) ». Nous devons mettre l'information du XML dans ce dernier en utilisant les attributs préalablement définis. Dans notre exemple, nous avons « access » et « name ». Le compilateur met les

valeurs appropriées dans les attributs et le code C# est généré. Nous pouvons utiliser le modèle correspondant de C++, VHDL ou bien Java pour que notre compilateur génère ceux-ci.

Listing 5.4 – Un modèle d'une classe en C# pour la génération de code

```
1 class(access,name,baseargs,declarations,functions,thread) ::= <<
2 <access> class <name> <if(baseargs)>: <baseargs; separator=","><endif>
3 {
4       <declarations; separator="\n">
5       <thread; separator="\n">
6       <functions; separator="\n">
7 }
8 >>
```

Les transformations XML permettent d'avoir un XML unique que notre compilateur peut traiter. Une fois que nous obtenons une représentation commune. Nous pouvons construire des graphes pour effectuer des analyses et des traitements.

5.2 Génération automatique du TLM et du RTL

Il faut distinguer deux types d'analyses, la première est celle d'une modélisation transactionnelle et la deuxième est celle au niveau des registres. L'objectif de la compilation TLM et RTL est d'avoir un code séquentiel d'un modèle hétérogène ayant plusieurs processus pour obtenir un code séquentiel qui peut être exécuté dans plusieurs langages.

Nous avons expliqué sommairement comment avoir un code homogène, les sections suivantes présentent l'analyse et le traitement pour les niveaux d'abstractions mentionnés.

5.2.1 TLM

Nous devons établir quelques définitions pour permettre une représentation interne commune des langages supportant un sous-ensemble de chaque langage hôte.

Définition 1 (Composant : Canal et Module). Un composant peut être une représentation d'un module « m $\in$ M » ou d'un Canal « ch $\in$ CH ». « M » et « CH » sont des descriptions des composants du modèle TLM, soit des canaux et des modules.

Définition 2 (Interface). Une interface permet au module d'accéder à des fonctions définies dans un canal. Une liste d'interfaces « I » contenant $\{i_1 , \ldots , i_n\}$ est notée « i $\in$ I ».

Définition 3 (Processus). Un processus est un sous-ensemble d'une fonctionnalité appartenant à un module. Un processus peut être composé d'éléments évènementiels « evs » avec des noeuds d'attentes « vw » et des noeuds vn. Les processus « P = $\{p_1 , \ldots , p_n\}$ » sont exécutés concuramment.

Définition 4 (Évènement). Un évènement e est contrôlé par deux noeuds distincts soit des noeuds d'attente « vw » et des noeuds vn pour la notification. Nous avons minimalement un noeud d'attente et un noeud de notification. Une liste d'évènements « EV » contenant $\{ev_1 , \ldots , ev_n\}$ est noté ev $\in$ EV. Aussi, on peut dire « $\forall$ ev $\in$ EV $\exists$ vw $\in$ VW ».

« VW » est l'ensemble des noeuds d'attente du système qui bloque un processus. De plus, un noeud « vw » peut attendre plusieurs « evs » pour s'exécuter. La figure 5.1 illustre deux processus avec des évènements.

À la fin du processus de compilation, nous voulons particulièrement obtenir l'élimination des interfaces, des modules et des canaux pour avoir un seul et unique processus que nous pouvons exécuter. Contrairement à la méthode standard qui consiste à distribuer les processus sur plusieurs processeurs.

De manière générale, une description d'un système avec une liste uniforme « ($\{m_1, \ldots , m_n\}$, $\{i_1, \ldots , i_k\}$,$\{ch_1, \ldots , ch_n\}$) » dans un modèle unique permet d'avoir une simulation accélérée. Les ensembles « M, I, CH » sont respectivement les modules, les interfaces et les canaux. L'objectif est d'avoir « M, I, CH » ayant plusieurs processus dans un même processus p. Les évènements « ev » seront émulés et les noeuds d'attente « vn » continueront leur exécution lorsque le prochain élément de la liste « EV » correspondra à « vn ».

Dans le cadre notre recherche, nous validons les modèles qui servent d'exemple par la simulation. Tout d'abord, nous simulons les modèles par les méthodes traditionnelles avec un seul langage de modélisation. Ensuite, nous les simulons par notre compilateur. Et pour terminer, nous comparons les résultats entre notre approche et les méthodes stan-

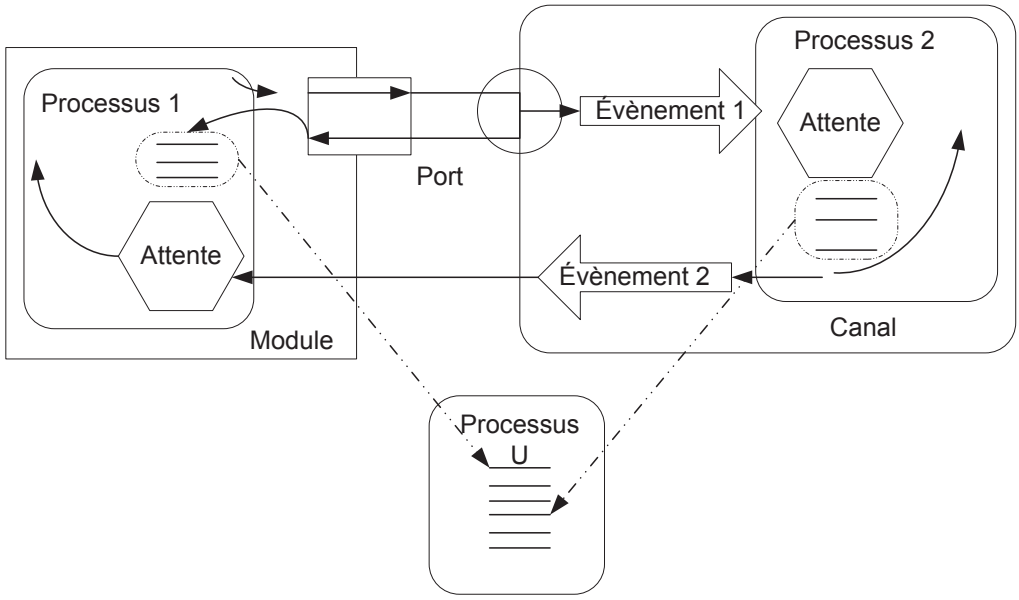


Figure 5.1 – Exemple d'attentes et de notifications

dards. Dans tous les cas, nous obtenons les mêmes résultats que les méthodes standards.

Nous pouvons l'expliquer par une équivalence dans l'ordonnancement qui se fait dynamiquement lors d'une (co)simulation. L'ordre des évènements est préservé et les zones du TLM garde leur ordre d'exécution en suivant les délimitations des blocs par les « waits » et « notify ».

Notre architecture utilise les étapes suivantes.

- Lire le XML
- Extraction des modules, des signaux et du système
- Créer des graphes généraux pour les modules (PDG, CDG, FDG)
- Créer des instances déclarées dans le système
- Créer les connexions entre les modules et interfaces
- Construire le système
- Construire le graphe du modèle
- Construire le modèle TLM

- Générer le code

La figure 5.2 présente la structure utilisée pour la génération du TLM compilé. Le noeud est responsable de gérer sa ligne de code. Chaque noeud a une identification unique et a aussi son modèle « String Template (ST) ». Il a également ses relations de dépendance avec les autres noeuds voisins.

La figure 5.3 représente la structure évènementielle utilisée. Nous avons 4 principaux évènements soit ExamineEdge, FinishVertex, BackEdge, InitializeVertex [Halleux, 2006]. FinishVertex est appelé chaque fois qu'un noeud a été visité. InitializeVertex permet l'initialisation de chaque noeud. ExamineEdge est utile pour exécuter un code lors de l'observation d'un arc. BackEdge permet de détecter s'il y a un cycle dans le graphe. De plus, le noeud a trois couleurs possibles soit blanc, gris et noir. Le blanc indique que le noeud n'est pas visité et le noir est qu'il a été visité. Le gris permet de savoir si le noeud a été découvert. Dans le cas de notre recherche, nous utiliserons le parcours en profondeur pour visiter un arbre (les codes 5.5 et 5.6).

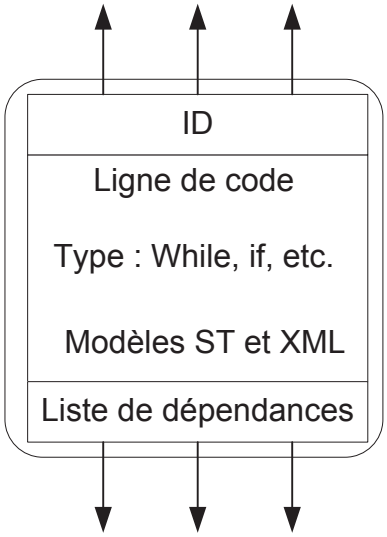


Figure 5.2 – Exemple d'un noeud TLM

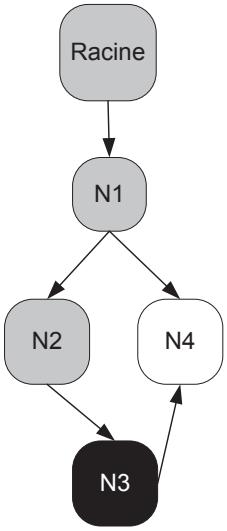


Figure 5.3 – Exemple d'une struture utilisée

Listing 5.5 – Algorithme DFS [Cormen et al., 2001]

```
1  DFS(G)
2  Pour chaque noeud u dans V
3        couleur[u] := blanc
4        p[u] = u
5  time := 0
6  s'il y a un noeud partant s
7  Appel DFS−VISIT(G, s)
8  pour chaque noeud u dans V
9   si couleur[u] = blanc
10    Appel DFS−VISIT(G, u)
11 retourne (p,d_time,f_time)
```

Listing 5.6 – Fonction visit du DFS

```
1  DFS−VISIT(G, u)
2  couleur[u] := gris
3  d_time[u] := time := time + 1
4  Pour chaque v dans Adj[u]
5   si (couleur[v] = blanc)
6        p[v] = u
7        Appel DFS−VISIT(G, v)
8   sinon si (couleur[v] = gris)
9        ...
10  sinon si (couleur[v] = black)
11       ...
12  −−finish vertex u
13  couleur[u] := blanc
14  f_time[u] := time := time + 1
```

C'est un algorithme de recherche qui progresse à partir d'une racine en s'appelant de manière récursive.

À partir du DFS et des graphes CDF, FDG, PDG et SDG, nous pouvons réaliser la compilation du TLM à l'aide du pseudo-code 5.7. À chaque fois qu'un noeud est visité, il faut l'identifier pour savoir si c'est un processus.

Listing 5.7 – Algorithme proposé du TLM

```
1  Identification de type processus
2  Obtenir la liste A des arcs sortants
3  Tant que A n'est pas vide
4      si A.cible== Statement
5          Construire le statement
6          Écrire le noeud
7          Met statement de la cible dans le modèle ST (initzone).
8      si A.cible== while
9          Construit test et met le modèle ST while
10         Obtenir les arcs de sorties de A.cible dans une liste C
11         Tant que C n'est pas vide
12             si la couleur de l'arc est black
13                 si appel de fonction
14                     Met le statement
15                     Appel traitement de l'appel(TA)
16                 sinon
17                     Met le code de la cible dans le modèle ST
```

Au début, nous avons l'identification des processus et l'obtention de leur liste d'arcs sortants. Il faut ensuite parcourir la liste A comme illustré à la ligne 3. Dans notre exemple, nous utiliserons le graphe SDG créé préalablement à la figure 4.7. Nous avons deux processus « EnterProc_run_producer » et « EnterProc_run_consumer ». Pour simplifier nos explications, nous utiliserons le sous arbre de « EnterProc_run_producer ». La liste A contiendra les arcs ayant la racine « EnterProc_run_ producer » et les noeuds 66, 67 et 69. Nous détectons alors deux types de noeud « statement » et « while ». Les expressions seront copiées dans la zone d'initialisation du modèle TLM. Le « while » est traité en deux phases. La première est de mettre le test du « while » dans le modèle ST et le deuxième est de parcourir les expressions de celui-ci. Nous obtenons la liste des arcs de sortie du « while » pour construire une liste C. Il faut noter l'utilisation d'un graphe colorié. Nous avons conservé une séparation des types arcs lors du regroupement

du CDF et FDG. Les arcs blancs sont pour le CDG et les arcs rouges sont pour le FDG. Nous conservons les arcs blancs qui représentent le chemin de contrôle du code. À partir d'arcs blancs, nous aurons les noeuds dépendants du « while ». Dans notre cas, la liste C est composée d'arcs entre 69 et 71. Le regroupement de processus nécessite l'élimination des fonctions et des interfaces pour avoir un processus unique. Nous allons détecter les appels de fonction. Si nous détectons un appel de la fonction on exécute une fonction TA (code 5.8) avec le noeud de l'appel sinon le « statement » est placé dans le modèle ST.

Listing 5.8 – Cherche les paramètres d'entrées d'une fonction

```
1 TA (Noeud de l'appel)
2     Obtenir la liste A des noeuds de sortie du noeud de l'appel
3         tant que A n'est pas vide
4             Cherchefonction(noeud)
```

TA permet la recherche des paramètres d'entrées d'une fonction. Lorsque nous construisons le SDG, nous avons des arcs qui relient les paramètres avec les appels des fonctions. Par exemple, la fonction écriture de la ligne 71 a deux noeuds « call_interface_argstr » et « call_interface_argcnt ». Ces deux noeuds sont reliés au noeud « call_param_arg_writec ». Dans TA, on obtient la liste des noeuds de sortie de l'appel (« call_interface_argstr » et « call_interface_argcnt »). L'objectif est de trouver la racine de la fonction appelée. À partir des noeuds de paramètres, nous appelons la fonction de recherche (code 5.9).

Listing 5.9 – Recherche la fonction

```
1 Cherchefonction (noeud)
2     Obtenir la liste A des noeuds de sortie du noeud
3         tant que A n'est pas vide
4             Obtenir la liste B des entrées de A
5                 si (noeud entrant==fonction)
6                     Construit la fonction
7                     retourne la fonction
```

Pour les noeuds « call_interface », nous obtenons une liste de leur noeud de sortie. Le noeud de sortie est « call_param_arg_writec ». À partir de celui-ci, nous obtenons une autre liste correspondante à ses entrées. Pour chaque élément de la liste, nous vérifions

s'il y a le type fonction. Nous obtenons la racine de la fonction « Enter_write_fifo » en plus d'avoir trouvé la fonction éliminée, les « waits » et les notifications.

Nous construisons la fonction avec l'algorithme DFS et nous notons les relations entre les évènements et les notifications. Lors du parcours du graphe, nous détectons le type « wait » ou « notification » (code 5.10) .

Listing 5.10 – Détection du type de noeud (« wait ou notify »)

```
1 FinishVertex
2 si noeud est du type simulateur
3       si wait change wait
4       si Notify change Notify
```

Le « wait » et le « notify » sont des noeuds de type simulation, car ce sont des fonctions que, normalement, le simulateur exécute selon son propre noyau de simulation. Lorsque nous visitons un « wait », nous devons le changer en « break » à l'aide du code 5.11.

Listing 5.11 – Change « wait »

```
1 this.type=break;
```

Lorsqu'on visite un « notify », on peut le changer en un autre statement (code 5.12). Pour changer un notify, on obtient le modèle ST d'une expression en XML. Dans le modèle, on remplit les attributs gauche et droite de l'expression. ST génère le XML correspondant et on l'ajoute au noeud. Par exemple la fonction « notify(A) » sera remplacée par A=true.

Listing 5.12 – Change Notify

```
1 Obtenir le modèle String Template(ST) d'une expression en XML
2 Mettre les attributs de gauche et droite de l'expression
3 Lire le XML créé par le ST
4 Ajouter le code XML dans le noeud.
```

La génération de code fonctionne avec le DFS. Le code inférieur à l'arbre est lu en premier jusqu'à la racine. Chaque noeud inférieur transmet son modèle avec ses attributs au noeud supérieur. Le code complet est généré en imprimant la racine de l'arbre tel qu'illustré à 5.4.

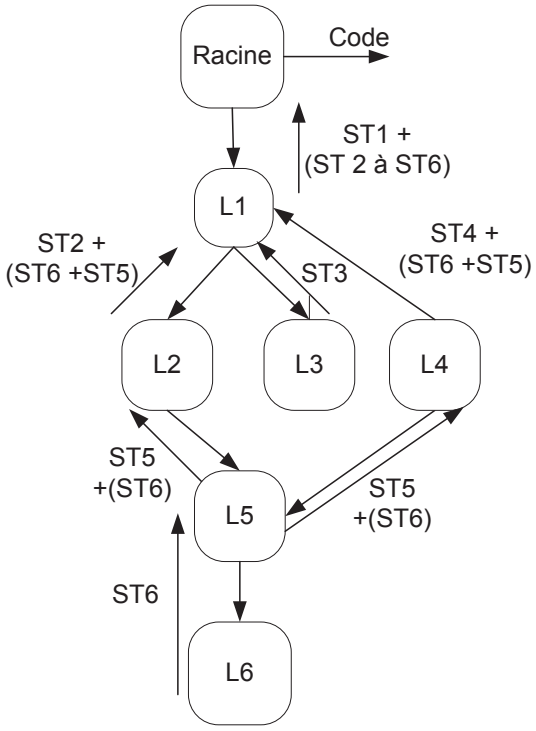


Figure 5.4 – Exemple pour un modèle TLM

Les lignes L1 à L6 passeront leur modèle ST chargés par leur attribut respectif jusqu'à la racine pour obtenir le code final. Les attributs sont remplis lorsqu'il y a une visite du parcours de graphes en profondeur.

Après tout ce travail, nous voulons obtenir un code séquentiel qui ne contient plus de « notify » et de « wait ». L'ordonnancement des processus se fait à l'aide d'une liste comme dans les simulateurs SystemC et ESyS.Net. Notre ordonnanceur est inclus dans le code généré et les processus sont séparés en zone. De plus, nous n'avons pas de limitation dans le langage que notre compilateur peut générer. Nous utiliserons un langage qui s'exécute le plus rapidement possible comme C++. La génération de code n'est pas une technique, mais combiné à un simulateur, elle permet d'avoir une durée de simulation plus court. La nouveauté repose sur l'éliminitation des « waits » et des « notify » ainsi que la décomposition des processus. Par exemple, le code 5.13 en C# est généré

par notre compilateur et correspond au modèle du producteur et consommateur en utilisant le modèle général de la figure 4.10 et des algorithmes que nous venons expliquer. Notez que le code est précis au niveau transactionnel. L'exemple typique est celui de la référence du manuel de SystemC.

Listing 5.13 – Code généré par notre compilateur pour un producteur et un consommateur avec C#

```
1  class eventtest{
2  private int max = 10; private char[] data = new char[10];
3  private int num_elements,first,choix,max;
4  private byte next;
5  private bool end, N0E0, N0E1 = false;
6  public void Run(){
7  IPriorityQueue PQ = new BinaryPriorityQueue();
8  PQ.Push(0); PQ.Push(1); PQ.Push(100);
9  //initialisation du consommateur
10 char c = 'r';
11 //initialisation du producteur
12 int cnt = 0;
13 int start = 0;
14 while(end!=true){
15       if (N0E0) { PQ.Push(1); N0E0 = false; }
16       if (N0E1) { PQ.Push(0); N0E1 = false; }
17       n = (int)PQ.Pop();
18       switch (n){
19       case 1: while (cnt < 500000){
20            if ((int)num_elements == max) {break;}
21            data[(first + num_elements) % max] = str[cnt];
22            ++num_elements;cnt++;
23            N0E1 = true;}
24            break;
25       case 0: while(true) {
26            if (num_elements <= 0){break;}
27            c = data[first];
28            −−num_elements;
29            first = (first + 1) % max;
30            N0E0 = true;}
31            break;
32       case 100: end = true; break;
33       default: break;
34 }}}}
```

Nous utilisons un tampon de données fixe de 10 pour la taille du FIFO. Nous simulons différentes quantités que le producteur va produire. Les lignes 2 à 4 sont tous les éléments dans le FIFO que nous avons déplacés dans une nouvelle classe « eventtest ». La ligne 5 déclare tous les évènements transformés en utilisant une valeur booléenne. Les lignes 6 à 36 décrivent la fonction simulée en regroupant le producteur, le consommateur et le FIFO ensemble. La phase d'initialisation de la simulation correspond à la zone aux lignes 9 à 15 dans le modèle du producteur et du consommateur. Dans les lignes 16 à 36, nous avons une boucle pour émuler un simulateur. Chaque évènement correspond à une lecture ou une écriture qui est remplacée par N0E0 et N0E1 respectivement. Comme illustré à la ligne 22, le « wait » est remplacé par un « break » et la notification à la ligne 32 pour ce « wait » est remplacé par un booléen. Toutes les fonctions appelées sont regroupées directement dans le code appelant. Dans notre exemple, le code FIFO est inséré dans les lignes 21 à 25 et 28 à 31. Cependant, la fonction appelée n'a pas besoin de branchement à la classe instanciée du code appelant. Les lignes 17 à 18 insèrent dans la liste de priorité les évènements qui ont lieu à la ligne 25 ou 32. Les deux cas du « switch statement » correspondent aux deux processus qui appellent le FIFO et l'autre cas permet de sortir de la simulation. Le code a aussi été transformé en Java et C++. Il est important de noter que le code généré est complètement différent du modèle d'entrée.

Un autre exemple est un producteur et 3 consommateurs qui sont montrés aux codes 5.14 et 5.15. Les lignes 12 et 13 sont une vue d'ensemble pour émuler des événements. Chaque événement est remplacé par une valeur booléenne et chaque « wait » est remplacé par un « break ». Les lignes 5, 6, 7 et 14 sont employées pour synchroniser le cas du « switch case » pour obtenir une simulation appropriée. Les trois consommateurs sont respectivement décrits aux lignes 28, 33 et 38. Toutes les fonctions appelées sont copiées directement en leur code appelant. Les lignes 16 à 27 sont le producteur et les trois FIFOs sont regroupés ensemble dans le même « switch case ».

Listing 5.14 – Code généré par notre compilateur pour un producteur et 3 consommateurs avec C#

```
1  class eventtest {
2  //Déclaration de l'instance FIFO (donnée interne)
3  //Déclaration des évènements émulés
4  public void Run() {
5  IPriorityQueue PQ = new BinaryPriorityQueue();
6  PQ.Push(3);
7  PQ.Push(100);
8  //Initialisation des consommateurs et du producteur
9  int n = 1;
10 Random r = new Random();
11 while(end!=true) {
12     if (N0E0) { PQ.Push(3); N0E0 = false; }
13     if (N1E0,N2E0,N0E1,N1E1,N2E1) //Même if que la ligne précédente avec chaque N1
           XX et N2XX
14     n = (int)PQ.Pop();
15     switch (n) {
16     case 3: while (cnt < 10000){
17         i=r.Next(3)+1;
18         if(i==1){if ((int)num_elements == max) {break;}
19         data[(first + num_elements) % max] = i;
20         ++num_elements; N0E1 = true;}
21         if(i==2){ if ((int)num_elementsf1 == maxf1) {break;}
22         dataf1[(firstf1 + num_elementsf1) % maxf1] = i;
23         ++num_elementsf1; N1E1 = true;}
24         if(i==3){if ((int)num_elementsf2 == maxf2) {break;}
25         dataf2[(firstf2 + num_elementsf2) % maxf2] = i;
26         ++num_elementsf2; N2E1 = true;}
27         cnt++;} break;
28     case 0: while(true){
29         if (num_elements <= 0){break;}
30         c = data[first]; −−num_elements;
31         first = (first + 1) % max;
32         N0E0 = true;} break;
33     case 1: while(true){
34         if (num_elementsf1 <= 0){break;}
35         cf1 = dataf1[firstf1]; −−num_elementsf1;
36         firstf1 = (firstf1 + 1) % maxf1;
37         N1E0 = true;}break;
```

Listing 5.15 – Code généré par notre compilateur pour un producteur et 3 consommateurs avec C# (suite)

```
38      case 2: while(true){
39          if (num_elementsf2 <= 0){break;}
40          cf2 = dataf2[firstf2];
41          −−num_elementsf2;
42          firstf2 = (firstf2 + 1) % maxf2;
43          N2E0 = true;}break;
44      case 100: end = true; break;
45      default:
46          Console.WriteLine("Invalide");
47      break;
48  }}}}}
```

Dans les lignes 18, 21 et 24 ça décrit le maximum, maxf1 et maxf2. Un FIFO a une taille limitée. Les trois instances du consommateur seront prolongées avec F1 et F2. La première instance est normale ; la seconde est marquée F1, la troisième F2 et ainsi de suite. Dans les lignes 11 à 48, nous avons une boucle pour émuler un simulateur. Chaque événement correspondant à une lecture ou à une écriture est remplacé par N0E0 et N0E1 respectivement. Par conséquent, la fonction appelée n'a pas besoin de se connecter à la classe FIFO et elle peut être simulée directement. Les codes 5.14 et 5.15 ont été également traduits en C++. Nous avons expliqué le fonctionnement de la génération du code TLM, la section suivante présente celle du RTL.

5.2.2 RTL

L'objectif du RTL est d'avoir un modèle compilé à partir d'une description multi-langage et de permettre à l'usager d'avoir la possibilité d'ajouter des modèles pour la vérification ou autres. À partir d'un modèle interne, nous allons générer un code séquentiel qui émule le comportement d'une horloge. Les étapes suivantes présentent les opérations effectuées par notre compilateur.

- Extraction des modules, des signaux et du système
- Création des instances des modules
- Regrouper les modules et faire la propagation des IOs
- Construire le reset
- Détermination du type de signaux combinatoire ou sensitif
- Ordonnancer les parties combinatoires
- Générer le code

Au début, les modules sont extraits et stockés dans une structure, mais ils ne seront pas nécessairement utilisés. Le compilateur détermine lesquels seront utilisés par la description du système. De plus, il peut y avoir la création de deux instances ou plus d'un module. Dans la création des instances, on créera une ou plusieurs instances selon le cas en utilisant le nom défini par le système. Par exemple, un système utilise deux modules additionneurs nommés A et B. A et B réfèrent à un module additionneur. Dans la version compilée, il faut copier le module additionneur 2 fois avec ses structures.

Lorsque la création des modules est réalisée, il faut les regrouper selon la description du système. À partir des signaux les connectant, nous pouvons les propager dans ceux-ci.

Par exemple, selon le code 5.24, un signal « sig2 » est défini dans un système et relie « feeder » et additionneur par le « portb » du « feeder » et le « portb » de l'additionneur. L'opération propagation sera de remplacer le nom « portb » de l'instance « feeder » et additionneur par « sig2 ». Le nom du port est remplacé par le signal du système.

Lors de la compilation d'un code, les déclarations d'initialisations des signaux sont sous la forme d'un AST. Il est important de construire un reset pour les signaux. À cette

étape, un tableau est créé pour avoir tous les signaux et leurs états initiaux. De plus, il faut définir, pour chaque signal, à quel type il appartient.

Selon la définition du RTL de la revue de littérature, nous avons les signaux dépendant d'une horloge et ceux qui sont combinatoires. Chaque signal sera noté sensitif ou combinatoire. De même, nous dirons qu'une ligne de code est sensitive si elle dépend de l'horloge ; sinon, elle sera combinatoire. La ligne de code aura comme nom le signal le définissant et une liste sera définie de tous les signaux du côté droit de l'assignation. Par exemple, « a=b+c » aura « a » comme nom et une liste ayant « b » et « c » sera créée.

Pour ordonner les lignes combinatoires, on obtient l'ensemble des lignes combinatoires dans un même AST non ordonnancé. Chaque signal peut être représenté comme étant un noeud avec des entrées et des sorties tel qu'illustré à la figure 5.5. Contrairement à la littérature, nous ne créons pas un graph avec les registes et les parties combinatoires. Nous utiliserons l'AST directement pour ordonnancer et créer le code compilé. L'algorithme en pseudo-code 5.16 permet de créer les noeuds d'entrées pour chaque signal.

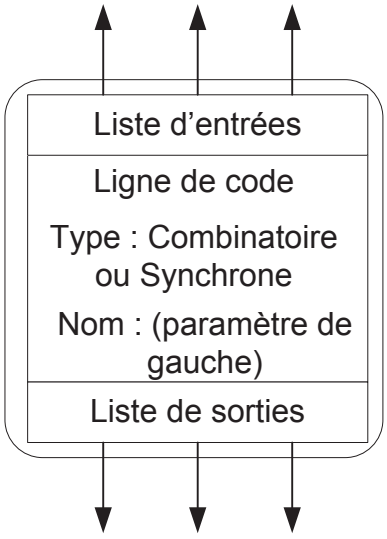


Figure 5.5 – Exemple de noeud

Listing 5.16 – Création des noeuds d'entrées pour chaque noeud

```
1  Créer une liste A
2  Ajouter un noeud racine dans la liste A
3  Pour chaque ligne de code
4      Créer un noeud et mettre la valeur de gauche comme nom
5          Pour chaque élément à droite de l'expression
6              Si l'élément est différent du nom
7                  Ajouter l'élément dans la liste d'entrée du noeud
8                  Si l'élément n'est pas dans la ligne de code
9                      Créer un noeud avec le nom de l'élément
10                     Ajouter TOP dans sa liste d'entrées
11                     Ajouter le noeud dans la liste A
12     Ajouter le noeud dans la liste A
```

Pour construire la relation entre les noeuds et effectuer un ordonnancement, il faut, pour chacun d'eux, définir les entrées des noeuds. Nous avons la création d'une liste vide contenant tous ces derniers avec une racine.

À chaque ligne de code, un noeud est créé à partir de son expression. Par exemple, X = A + B crée le noeud X. Pour éviter de créer un graphe cyclique, nous évitons d'avoir un arc qui se réfère sur lui-même comme X = X+1. Nous ajoutons les expressions de droite dans la liste d'entrées de ce dernier. Si l'expression est une déclaration alors le noeud sera relié à la racine. Une fois que les listes des noeuds entrants sont créées pour chacun, nous pouvons créer la liste des noeuds sortant pour chacun d'eux.

L'algorithme en pseudo-code 5.17 permet de créer les noeuds de sortie pour chaque signal.

Listing 5.17 – Création des noeuds de sortie pour chaque noeud

```
1  Pour chaque noeud dans la liste A
2      Pour chaque élément dans la liste d'entrée
3          Obtenir le noeud dans la liste A correspondant à l'élément
4          Ajouter dans la liste de sortie du noeud trouvé l'élément
```

Chaque noeud est parcouru dans la liste créée précédemment et celle des entrées est parcourue pour chacun. Nous regardons le noeud dans la liste A qui correspond à l'élément de sortie. L'idée est de créer une liste de sorties pour chaque noeud à partir de liste d'entrées. Une fois que chaque noeud possède ses entrées et ses sorties par rapport à l'autre, nous pouvons commencer l'ordonnancement. Pour ordonner adéquatement le

code, il faut créer des niveaux pour les noeuds à l'aide du code 5.18.

Le niveau 0 correspond à la racine et les niveaux sont incrémentés. Nous allons attribuer un niveau pour chaque noeud tel que la racine aura le niveau 0.

Listing 5.18 – Numéroration des noeuds

```
1 Numérotation(Arbre,niveau)
2         Pour chaque élément dans la liste de sorties
3         Prendre le noeud correspondant à l'élément dans la liste A
4     Attribuer le niveau en cours à l'élément
5     Numérotation (noeud,niveau + 1)
```

À partir de la racine, les noeuds sont parcourus selon les entrées et les sorties que nous avons précédemment créées. C'est une fonction récursive. Nous parcourons les listes de sorties de chaque noeud et, avant de monter d'un niveau, chaque noeud est numéroté sortant du noeud actuel d'un niveau inférieur. Par exemple à la figure 5.6, la racine contient le noeud « sig1 » et « sig2 » alors le niveau 1 est attribué pour chacun d'eux. De même, « sig1 » attribuera le niveau 2 pour le noeud « sig3 ».

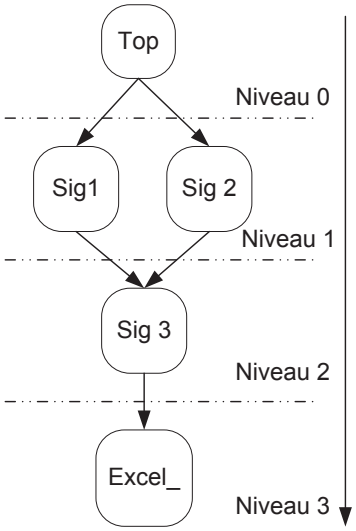


Figure 5.6 – Exemple de niveaux

Nous utilisons deux listes : B et niveau. La liste niveau contient les listes B de chaque niveau. Pour chaque niveau, nous avons une liste contenant tous les noeuds appartenant à ce niveau. La création d'une liste niveau est illustrée au code 5.19.

Listing 5.19 – Création d'une liste niveau

```
Créer une liste niveau vide
Pour chaque noeud dans la liste A
    Si le noeud est dans la liste Niveau
        Mise à jour du nom noeud dans la liste B
    sinon
        Créer une Liste B
        Ajouter le nom du noeud dans la liste B
        Ajouter la liste B dans la liste Niveau
```

Chaque noeud est parcouru pour déterminer le niveau auquel il appartient. Par exemple, le noeud racine aura une liste B contenant la racine et cette liste B sera dans la liste niveau. En procédant de cette manière, le code combinatoire est ordonnancé. À la fin de l'opération, il reste juste à parcourir les niveaux pour générer le code combinatoire (5.20).

Listing 5.20 – Génération du code combinatoire

```
Pour chaque niveau
    Pour chaque noeud du niveau
        Obtenir le code AST du noeud
        Ajouter au nouveau AST
```

Le code AST est généré à partir de la racine et chaque expression combinatoire sera ordonnancée dans un ordre adéquat.

Les codes 5.21 et 5.22 permettent d'avoir un exemple d'un code RTL de la figure 4.8. Nous avons la classe multiplication et additionneur. Nous avons aussi la classe « feeder » (code 5.23) et la déclaration du système (code 5.24).

Listing 5.21 – Classe Adder

```
public class Adder : BaseModule{
public IntSignal porta,portb,portc;

[MethodProcess]
[EventList("sensitive","porta","portb")]
private void Run() {
    portc.Value = porta + portb;}}
```

Listing 5.22 – La classe multiplication

```
public class Multiplation : BaseModule
{ public IntSignal porta2,portb2,portc2;

[MethodProcess]
[EventList("sensitive", "porta", "portb")]
private void Run()
    {portc2.Value = porta2 * portb2;}
}
```

Listing 5.23 – La classe feeder

```
1  public class feeder : BaseModule
2  { public IntSignal porta,portb,portc;
3  private int i = 1;
4
5  [MethodProcess]
6  [EventList("sensitive","clk")]
7  private void Run()
8  {i=i+1;
9  porta.Value = i;
10 portb.Value = i;
11 }
12 }
```

Listing 5.24 – La classe système

```
1  public class MySystem : SystemModel{
2  private Clock clk = new Clock(2);
3  private Multiplation Mult = new
       Multiplation();
4  private Adder adder = new Adder();
5  private feeder feed = new feeder();
6  private IntSignal sig1,sig2,sig3 = new
       IntSignal();
7  private IntSignal Excel_ = new IntSignal();
8  public MySystem(Simulator manager,
       string name):base(manager, name){
9  //Lien des ports
10 feed.porta = sig1;feed.portb = sig2;
11 adder.porta = sig1; adder.portb = sig2;
12 adder.portc = sig3;
13 Mult.porta2 = sig3; Mult.portb2 = sig3;
14 Mult.portc2 = Excel_;}}
```

Le mot clef « Excel_ » dit au compilateur que c'est un signal Excel. Le compilateur va mettre les données reçues dans le signal « Excel_ » et dans un fichier Excel. Nous avons la création d'un modèle Excel tel qu'illustré au code 5.25. Nous avons créé un mot réservé pour permettre à l'usager de tracer un signal dans un graphe avec un modèle Excel.

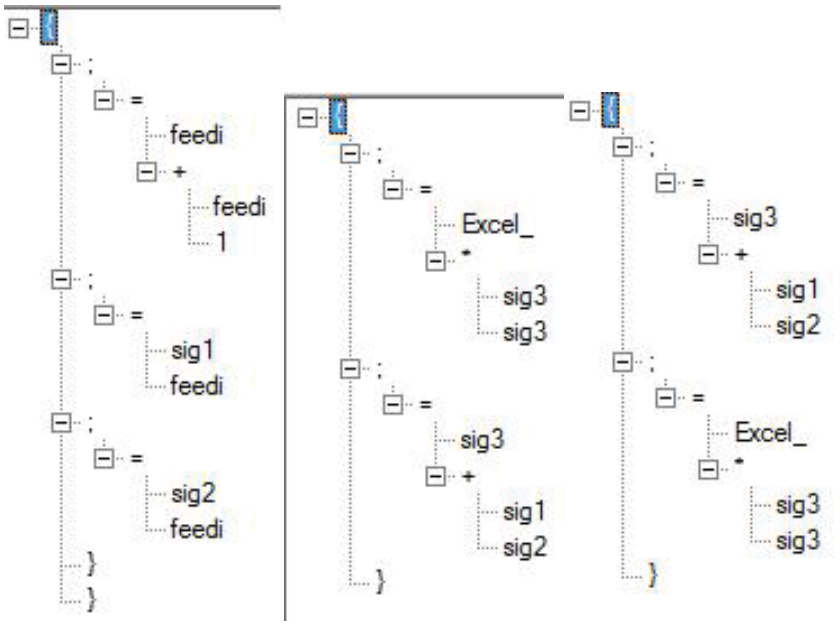
Listing 5.25 – Exemple d'un modèle Excel

```
1  initexcel()::=<<
2  object fileName = @"c:\Book3.xls";
3  Excel._Application xlApp; Excel._Workbook xlClasseur;
4  Excel._Worksheet xlFeuill1;
5  xlApp = new Excel.Application();
6  xlApp.Visible = true;
7  ...
8  >>
9
10 endexcel()::=<<
11 xlClasseur.Save(); // Enregistre les modifs des cellules
12 >>
13 cellexcel()::=<< xlFeuill1.Cells[j + 1, 1] >>
```

Dans ce modèle, nous avons l'initialisation, la fermeture et la cellule d'Excel. Nous

ouvrons et fermons un fichier Excel. Pendant l'exécution du code, les valeurs simulées sont écrites dans un fichier. Il faut noter que l'initialisation, la fermeture et la cellule Excel sont respectivement dans les zones 1,5,6 du modèle RTL de la figure 4.9.

Les figures 5.7a, 5.7b et 5.7c présentent un exemple d'ordonnancement combinatoire ainsi que des lignes synchrones.



(a) Lignes synchrones (b) Avant ordonnancement (c) Après ordonnancement

Figure 5.7 – Ordonnancement d'un modèle RTL

Nous pouvons constater que les processus ont été fusionnés et qu'il y a une commutation dans l'AST des lignes pour « sig3 » et « Excel_ ». Les lignes synchrones n'ont pas été ordonnancées. Nous pouvons aussi constater qu'il n'y a pas de « .now » et de « .last » parce que le code n'est pas généré par ST. La commutation du signal entre le temps passé et maintenant, est générée par un modèle « com(sig) » (code 5.26).

Listing 5.26 – Modèle en String Template pour le RTL

```
1 com(sig)::=<< <sig>.last = <sig>.now; >>
2 fieldcominit(name,init) ::=
3 << <name>.last =<init> ; >>
```

De la même manière, nous pouvons définir un modèle d'initialisation « fieldcomi-

nit ». Au début, nous avons défini une structure avec « now » et « last » comme integer. Les signaux synchrones auront « .now » et « .last ». Chaque signal a été préalablement identifié pour savoir s'il est combinatoire ou synchrone. La prochaine section discutera des résultats d'expérimentations pour le TLM.

5.3 Expérimentations

Nous nous concentrerons dans cette section sur le gain pouvant être obtenu pour un modèle TLM en utilisant SystemC et ESyS.Net. Le gain d'accélération pour une simulation d'un modèle RTL par un simulateur compilé est déjà connu. Un ordre de grandeur de 2 est montré pour RTL [Jennings, 1991]. En [Kupriyanov et al., 2004], un gain minimun de 24 est obtenu en simulant un processeur en RTL comparativement à un simulateur événementiel discret. Notre méthodologie peut accélérer une simulation en générant du code C, C # et Java à partir d'un modèle de SystemC et d'ESyS.Net.

5.3.1 Accélérations pour des simulations homogènes

Le tableau 5.1 donne le temps de simulation en ms pour le code 5.13. Les codes générés par notre compilateur pour la simulation qui ont été les plus efficaces sont C# et C++. Java est le plus lent et nous l'avons mis comme comparaison seulement. L'expérimentation a été réalisée en utilisant un IBM ThinkPad R40 1.5GHz et 512M.

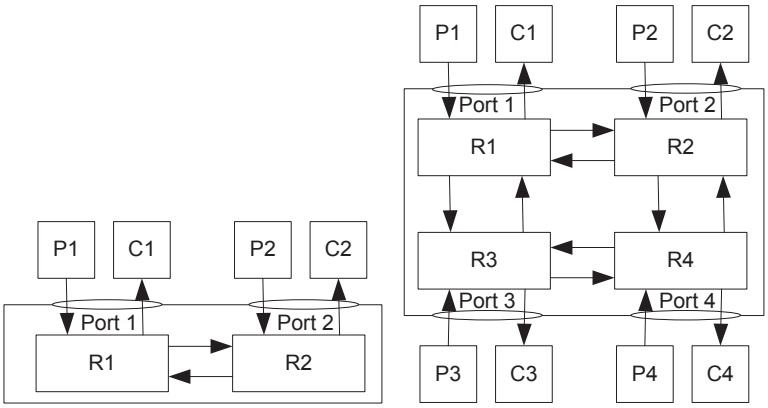
Tableau 5.1 – Producteur et un consommateur (temps de simulation en ms)

	Modèle classique		Nouveau TLM compilé		
Producteur (K)	ESyS.Net	SystemC	C++	C#	Java
500	5 267	20	10	10	40
1 000	10 223	40	20	20	50
2 000	20 537	90	40	40	80
4 000	41 266	180	80	70	130
8 000	82 782	360	160	130	240
16 000	169 800	420	310	260	350

Visual Studio .Net 2003 [Powers and Snell, 2006] est utilisé pour la compilation de

notre exemple. Nous avons utilisé l'environnement MinGW Developer Studio [Bernard, 2004] pour C++ et SystemC 2.2. En utilisant plusieurs modèles (langages), nous pouvons accélérer la simulation. ESyS.Net est le plus lent simulateur pour cette application. La technique du TLM compilée simule environ 500 fois plus rapidement qu'ESyS.Net dans notre expérience. Il est important de noter que le gain est fixe et que le temps de simulation est proportionnel au nombre de données produit par le producteur. Le nouveau TLM compilé est plus rapide que SystemC pour cet exemple.

Les figures 5.8a et 5.8b présentent deux autres exemples en modélisant des routeurs de 2 et de 4 ports. Chaque flèche représente un FIFO et chaque port est composé d'un FIFO en entrant et d'un autre en sortant. Nous avons des ports pour les paires P1/C1, P2/C2 ,P3/C3 et P4/C4. Un producteur (P) et un consommateur (C) écrivent et reçoivent des données des routeurs. Le producteur envoie une valeur et une adresse pour qu'un consommateur la reçoive. R1 à R4 servent à gérer les liens entre les ports. L'architecture et le routage sont basés sur un hybercube 1-D et 2-D. Chaque R possède un processus pour chaque entrée de FIFO qu'il possède. R a 2 processus dans l'exemple des 2 ports, tandis qu'il en a 3 pour celui du 4 ports. Notre objectif n'est pas d'évaluer les algorithmes de routage, mais plutôt la vitesse d'exécution des processus et leur interaction par rapport à un simulateur. Le modèle de 2 ports à 10 processus et celui de 4 ports possède 20 processus.



(a) 2 ports (b) 4 ports

Figure 5.8 – Exemple de routeurs

Tableau 5.2 – Routeur 2 ports

	Modèle classique		Nouveau TLM compilé		
Producteur	ESyS.Net	SystemC	C++	C#	Java
1 250	215	<20	<1	<10	<20
2 500	225	<20	<1	<10	<40
5 000	250	<20	10	<20	<40
10 000	274	<40	20	<20	<40
12 500	298	40	20	25	40
25 000	390	50	40	60	60
50 000	510	80	80	110	100
100 000	786	160	150	170	160

Tableau 5.3 – Routeur 4 ports

	Modèle classique		Nouveau TLM compilé		
Producteur	ESyS.Net	SystemC	C++	C#	Java
1 250	438	<1	<1	<40	<40
2 500	450	10	10	40	40
5 000	505	10	20	60	60
10 000	590	50	50	80	71
12 500	623	60	60	90	120
25 000	835	130	120	150	140
50 000	1 240	250	240	270	221
100 000	2 100	340	330	380	470

Les tableaux 5.2 et 5.3 présentent les résultats pour des simulations avec les routeurs. Chaque producteur envoie des données à tous les consommateurs C1 à C4. Il faut un minimum de quantité de données pour évaluer le temps de commutation entre les processus pour la simulation. Chaque FIFO possède des évènements et selon ceux-ci le simulateur va choisir le prochain processus à exécuter. Un routeur 2 ports possède 12 évènements (6 FIFOs x2), tandis que celui avec 4 ports en possède 32 (16 FIFOs x2). Chaque FIFO possède 2 évènements pour lire et écrire. Le rapport du temps d'exécution entre le routeur 2 ports et celui du 4 ports est proportionnel aux nombres d'évènements actifs à gérer. Par exemple, si on prend ESyS.Net avec une production de 100 K données, on obtient

un rapport de 2.67 soit 2 100 sur 786 qui est l'équivalent à celui des évènements des systèmes soit 32 divisé par 12.

SystemC et le TLM compilé en C++ ont des temps d'exécutions comparables. La gestion des événements est semblable dans les deux cas. Les deux analysent le changement événementiel lorsqu'un processus est terminé pour déterminer le prochain à exécuter. Lors de notre premier exemple du code 5.13, il y avait seulement 2 évènements et le temps de commutation du TLM compilé est au minimum, mais il augmente lorsqu'il y a plusieurs évènements. L'ordonnancement de SystemC et du TLM compilé est semblable et prend un temps similaire pour choisir le prochain processus à exécuter.

Les résultats du TLM compilé avec C # et Java sont plus lents que les autres quand la quantité de données produites est suffisante pour évaluer le temps de commutation.

Notre expérimentation montre que nous ne sommes pas seulement limités à programmer en SystemC pour obtenir une bonne performance de simulation. ESyS.Net peut avoir une durée de simulation équivalente à SystemC avec une compilation MinGW.

Le choix des langages est important pour compiler notre code séquentiel généré par notre compilateur. Un langage natif s'exécutera plus rapidement qu'un langage interprété. Contrairement à la méthode traditionnelle l'ordonnanceur n'est pas lié à l'hôte. Notre compilateur élimine les interfaces, les appels de fonctions et conserve un ordonnanceur qui est inclus avec le code généré. Les gains de simulation proviennent principalement du langage hôte que notre compilateur utilise pour simuler et aussi selon le cas le regroupement des processus. Nous avons aussi comparé dans la prochaine section les performances d'une simulation entre des modèles écrits en utilisant plusieurs langages.

5.3.2 Accélérations pour des simulations hétérogènes

Un autre aspect de la recherche en simulation qui est notre objectif est la co-simulation entre des modèles écrits dans différents langages. Nous montrerons donc une comparaison pour une co-simulation TLM. Pour notre expérimentation, nous utiliserons trois consommateurs et un producteur comme illustrés à la figure 5.9. Dans la figure 5.10, nous utilisons une mémoire partagée pour relier SystemC et ESyS.Net ensemble. Le producteur a un processus et le consommateur en a un aussi. Nous codons le

consommateur en SystemC et le producteur avec ESyS.Net. Le producteur envoie une une quantité fixe de données à un consommateur qui est défini aléatoirement. Nous appliquons une méthode de synchronisation entre les simulateurs. Un processus appelant est connecté à un processus appelé en utilisant un FIFO en mémoire partagée.

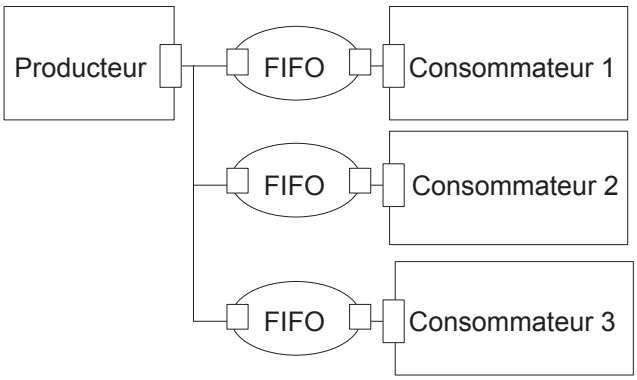


Figure 5.9 – Producteur avec 3 consommateurs [Dubois et al., 2007]

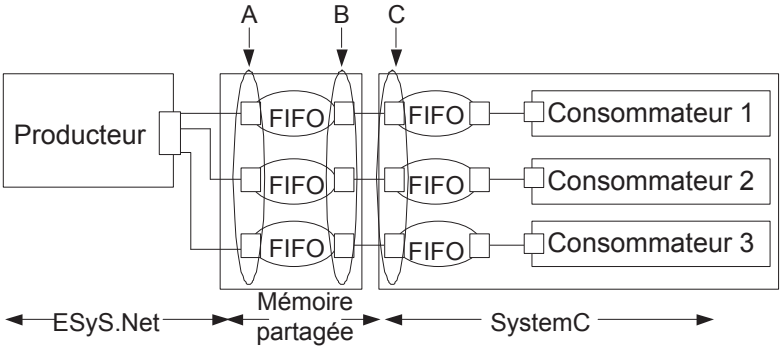


Figure 5.10 – Producteur avec 3 consommateurs utilisant la mémoire partagée [Dubois et al., 2007]

Dans la figure 5.10, A est la fonction appelante représentée par les boîtes pour le producteur écrivant dans le FIFO de la mémoire partagée (FIFO MP). La synchronisation entre A et B est fait par un mutex. C à 3 boites représentant trois processus. Chacun est synchronisé par un processus spécial (SP) dans le modèle. Nous émulons les évènements en utilisant les fonctions « wait » et « notify » pour un simulateur basé sur les

évènements. Cette méthode permet de synchroniser les processus. Le SP est créé pour dire au simulateur lequel des processus dans le modèle peut être synchronisé. Il vérifie aussi s'il y a des données dans les FIFOs à partir du processus appelant en ESyS.Net et lève un évènement correspondant au processus dans SystemC. Par ce chemin, il est possible de créer un lien dédié entre deux processus de différents simulateurs. Tous sont connectés avec un processus distant et sont synchronisés par ce processus spécial. Les autres, dans le modèle, sans connexion externe sont synchronisés par le noyau. Nous utilisons un tampon de données fixe de 100 éléments pour les FIFO MP. Nous simulons différentes quantités de données que le producteur produit.

Pour notre comparaison, nous employons ESyS.Net et SystemC pour examiner une co-simulation et une nouvelle approche en employant une représentation interne.

C'est une comparaison montrant le potentiel d'accélération pour un modèle de co-simulation. Dans le tableau 5.4, nous montrons trois méthodes pour simuler trois consommateurs et un producteur.

Tableau 5.4 – Temps de simulation en ms [Dubois et al., 2007]

	Modèle classique		Nouveau TLM compilé		Co-simulation
Producteur	ESyS.Net	SystemC	C++	C#	Mémoire partagée
1 000	1 100	< 1	< 1	35	5 500
10 000	7 400	10	< 10	45	47 000
50 000	36 071	30	30	75	236 460
100 000	73 746	60	60	100	400 000
200 000	147 832	140	140	160	800 000

La première méthode est une simulation classique comme SystemC et ESyS.Net. Le modèle entier est créé en simulant avec un seul noyau. Le second est l'utilisation de notre modèle TLM en utilisant notre compilateur et le troisième est une co-simulation entre SystemC et ESyS.Net. L'expérimentation a été réalisée avec un IBM ThinkPad R40 1.5GHz et 512M.

Visual Studio .Net 2003 est utilisé pour compiler le modèle FIFO en ESyS.Net, SystemC et C#. Nous remarquons que la co-simulation est 5 à 6.5 fois plus lente qu'ESyS.Net. Nous obtenons une accélération en utilisant GW C++ par un facteur 5714 fois, 5000 pour la version en C# comparativement à un modèle de co-simulation.

Le tableau 5.5 présente les résultats des co-simulations pour les routeurs. ESyS.Net est utilisé pour modéliser les ports 1 et 3, tandis que SystemC a les ports 2 et 4.

Tableau 5.5 – Co-simulation 2 et 4 ports

	Co-simulation	
Producteur	2 ports	4 ports
1 250	1 091	2 082
2 500	2 613	3 565
5 000	3 434	4 917
10 000	4 516	10 274
12 500	4 786	12 057
25 000	6 349	16 507
50 000	13 920	36 175
100 000	19 643	54 262

Dans l'exemple des routeurs, nous avons un facteur d'accélération d'environ 50 en utilisant GW C++ comparativement à un modèle de co-simulation.

Notre expérimentation montre une nouvelle méthode qui a été trouvée pour réduire la durée d'une co-simulation en utilisant une représentation interne et une version compilée du TLM.

CHAPITRE 6

CONCLUSION

La conception des systèmes hétérogènes requiert des langages de modélisations capables de modéliser correctement la fonctionnalité souhaitée. Chaque langage a ses forces et ses faiblesses et l'interconnexion des langages demande des interfaces compatibles. L'interface typiquement utilisée est un bus de communication entre les langages. De plus, l'interface peut être une mémoire partagée ou un lien TCP/IP comme nous l'avons expliqué. L'ajout de mécanismes de communication ralentit la simulation et requiert une synchronisation adéquate.

Ce travail de recherche a permis d'élaborer un simulateur compilé d'une description multi-langage des systèmes hétérogènes tout en ayant une séparation entre l'usager et les méthodes de simulation. La limitation syntaxique à un langage a été un problème à résoudre. Dans notre méthodologie de recherche, nous avons étudié l'impact des mécanismes de co-simulation. Contrairement à la méthode traditionnelle, nous avons présenté l'échange de données directes entre deux simulateurs sans l'ajout d'entête et de mécanisme de synchronisation selon le type de méthode utilisé.

L'étude des méthodes nous a permis d'en développer d'autres et de constater les faiblesses et les forces de chacun d'eux. L'évaluation, l'analyse et la comparaison des méthodes permettent de conclure qu'une simulation utilisant le COM est comparable à celle de la mémoire partagée et une interconnexion entre deux simulateurs gérée et non-gérée peut avoir des impacts sur le temps de simulation. L'environnement de .NET, COM, la mémoire partagée et Pinvoke sont les mécanismes privilégiés ; cependant, si nous avons besoin de l'asynchrone et d'un mécanisme non lié à .NET, le TCP/IP est un compétiteur fort pour la co-simulation efficace.

Après avoir analysé différents mécanismes et voulu obtenir un noyau commun en enlevant le bus de co-simulation, nous avons proposé un noyau, indépendamment de la syntaxe, de modélisation permettant une séparation entre la vue de l'utilisateur et le simulateur. Cette séparation entre le simulateur et une syntaxe de modélisation permet

une meilleure réalisation d'une simulation. La syntaxe utilisée pour exécuter un code peut être différente de celle initialement donnée par l'usager.

Notre environnement supporte la modélisation RTL et TLM hétérogène. Au niveau TLM, nous avons proposé une version compilée permettant d'accélérer la simulation TLM pour certains simulateurs comme ESyS.Net. Il faut noter que le TLM compilé et le simulateur SystemC avec les mêmes paramètres de compilation ont des temps de simulation comparables. Par contre, des accélérations ont été démontrées pour le simulateur ESyS.Net. De plus, la version du TLM compilé permet d'éliminer le noyau.

Nous avons proposé des techniques de minimisation des communications avec la possibilité d'utiliser plusieurs langages. Normalement, les méthodes de compilation se limitent à leurs langages respectifs sans permettre une minimisation entre plusieurs langages. Les processus sont exécutés et les appels sont minimisés dans leur langage respectif, incluant leurs interactions avec les autres langages nécessaires à la simulation complète d'un système. Dans notre compilateur, nous enlevons particulièrement toutes les interconnexions, les adaptateurs qui ne sont pas utiles pour la simulation. L'usager peut continuer à modéliser dans son langage de modélisation préféré et notre compilateur peut le fusionner avec d'autres sans avoir un bus de co-simulation.

La simulation hétérogène par notre approche devrait faciliter l'usage de la simulation d'un système entier durant tout le développement et que cela permet donc d'intégrer encore davantage la simulation au design et au développement.

BIBLIOGRAPHIE

A. A. Abdel-aziz and H. Oakasha. Mapping xml dtds to relational schemas. In *Computer Systems and Applications, 2005. The 3rd ACS/IEEE International Conference*, page 47, 2005. doi : 10.1109/AICCSA.2005.1387044.

P. Alexander. Rosetta : Standardization at the system level. *Computer, IEEE*, 42(1) :108 –110, jan. 2009. ISSN 0018-9162. doi : 10.1109/MC.2009.23.

L. Baresi, C. Bolchini, and D. Sciuto. Software methodologies for vhdl code static analysis based on flow graphs. In *Design Automation Conference, 1996, with EURO-VHDL '96 and Exhibition, Proceedings EURO-DAC '96, European*, pages 406 –411, 16-20 1996. doi : 10.1109/EURDAC.1996.558236.

S. Bensalem, M. Krichen, and S. Tripakis. State identification problems for input/output transition systems. In *Discrete Event Systems, 2008. WODES 2008. 9th International Workshop*, pages 225 –230, 28-30 2008. doi : 10.1109/WODES.2008.4605949.

Joey Bernard. Developing for windows on linux. *Linux J.*, 2004(123) :3, 2004. ISSN 1075-3583.

A. Bernstein, M. Burton, and F. Ghenassia. How to bridge the abstraction gap in system level modeling and design. In *Computer Aided Design, 2004. ICCAD-2004. IEEE/ACM International Conference*, pages 910 – 914, 7-11 2004.

G. Berry, L. Blanc, A. Bouali, and J. Dormoy. Top-level validation of system-on-chip in Esterel Studio. In *High-Level Design Validation and Test Workshop, 2002. Seventh IEEE International*, pages 36 – 41, oct. 2002.

J. Bhasker. Process graph analyzer : a front end tool for vhdl behavioral synthesis. In *System Sciences, 1988. Vol.I. Architecture Track, Proceedings of the Twenty-First Annual Hawaii International Conference*, volume 1, pages 248 –255, 5-8 1988. doi : 10.1109/HICSS.1988.11772.

W.D. Bishop and W.M. Loucks. A heterogeneous environment for hardware/software cosimulation. In *Simulation Symposium, 1997. Proceedings. 30th Annual*, pages 14 –22, 7-9 1997. doi : 10.1109/SIMSYM.1997.586458.

Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava : hardware design in haskell. In *Proceedings of the third ACM SIGPLAN international conference on Functional programming*, ICFP '98, pages 174–184, New York, NY, USA, 1998. ACM. ISBN 1-58113-024-4. doi : http://doi.acm.org/10.1145/289423.289440.

G. Bois, L. Filion, A. Tsikhanovich, and E .M. Aboulhamid. *Modélisation, raffinement et techniques de programmation orientée objet avec SystemC*. Hermes, 2003. ISBN 2-7462-0820-2.

Massimo Bombana and Francesco Bruschi. Systemc-vhdl co-simulation and synthesis in the hw domain. In *DATE '03 : Proceedings of the conference on Design, Automation and Test in Europe*, page 20101, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1870-2-2.

Don Box. *Essential COM*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997. ISBN 0201634465. Foreword By-Booch, Grady and Foreword By-Kindel, Charlie.

P.T. Breuer, L.S. Fernandez, and C.D. Kloos. Clean formal semantics for vhdl. In *European Design and Test Conference, 1994. EDAC, The European Conference on Design Automation. ETC European Test Conference. EUROASIC, The European Event in ASIC Design, Proceedings.*, pages 641 –647, 28 1994. doi : 10.1109/EDTC.1994.326810.

R. Buchmann and A. Greiner. A fully static scheduling approach for fast cycle accurate SystemC simulation of MPSoCs. *Microelectronics, 2007. ICM 2007. Internatonal Conference*, pages 101–104, Dec. 2007. doi : 10.1109/ICM.2007.4497671.

J.P. Calvez, D. Heller, and O. Pasquier. Uninterpreted co-simulation for performance evaluation of hw/sw systems. In *Hardware/Software Co-Design, 1996. (Codes/-*

CASHE '96), Proceedings., Fourth International Workshop, pages 132 –139, 18-20 1996. doi : 10.1109/HCS.1996.492235.

G. Canfora and A. Cimitile. Algorithms for program dependence graph production. In *Software Maintenance, 1995. Proceedings., International Conference*, pages 157 –166, 17-20 1995. doi : 10.1109/ICSM.1995.526538.

A. Cataldo, Edward A. Lee, X. Liu, E. Matsikoudis, and H. Zheng. Discrete-event systems generalizing metric spaces and fixed-point semantics. Technical Report UCB/ERL M05/12, EECS Department, University of California, Berkeley, Apr 2005. URL `http://www.eecs.berkeley.edu/Pubs/TechRpts/2005/4303.html`.

Spear Chris. *SystemVerilog for Verification, Second Edition : A Guide to Learning the Testbench Language Features*. Springer Publishing Company, Incorporated, 2008. ISBN 0387765298, 9780387765297.

Edmund M. Clarke, Masahiro Fujita, Sreeranga P. Rajan, Thomas W. Reps, Subash Shankar, and Tim Teitelbaum. Program slicing for vhdl. *International Journal on Software Tools for Technology Transfer*, 4(1) :125–137, 2002.

Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001. ISBN 0070131511.

R. Curtin. Asic design methods using VHDL. In *Euro ASIC '90*, pages 176 –179, 29 1990. doi : 10.1109/EASIC.1990.207933.

M. Dubois. Modélisation et conception d'une plate-forme de traitement et transmission de signaux vidéo numériques. Master's thesis, École Polytechnique de Montréal, 2004.

M. Dubois and E.M. Aboulhamid. Techniques to improve cosimulation and interoperability of heterogeneous models. In *Electronics, Circuits and Systems, 2005. ICECS 2005. 12th IEEE International Conference*, pages 1 –4, 11-14 2005. doi : 10.1109/ICECS.2005.4633510.

M. Dubois, El Mostapha Aboulhamid, and F. Rousseau. Towards an efficient simulation of multi-language descriptions of heterogeneous systems. In *Circuits and Systems, 2006. APCCAS 2006. IEEE Asia Pacific Conference on*, pages 538 –541, dec. 2006. doi : 10.1109/APCCAS.2006.342527.

M. Dubois, E.-M. Aboulhamid, and F. Rousseau. Acceleration for heterogeneous systems cosimulation. In *Electronics, Circuits and Systems, 2007. ICECS 2007. 14th IEEE International Conference on*, pages 294 –297, dec. 2007. doi : 10.1109/ICECS.2007.4510988.

M. Dubois, E.M. Aboulhamid, and F. Rousseau. *An Introduction to Cosimulation and Compilation Methods*. Chapman and Hill, 2009. URL `http://hal.archives-ouvertes.fr/hal-00472595/en/`. ISBN : 978-1-4398-1211-2.

Mathieu Dubois, Guy Bois, and Yvon Savaria. Double profiling methodology for video processing platform. In *AIC'04 : Proceedings of the 4th WSEAS International Conference on Applied Informatics and Communications*, pages 1–6, Stevens Point, Wisconsin, USA, 2004. World Scientific and Engineering Academy and Society (WSEAS). ISBN 960-8457-06-8.

Harold Elliotte Rusty and Means W. Scott. *XML in a Nutshell : A Desktop Quick Reference*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2000. ISBN 1402857381.

G. Fischer, J. Lusiardi, and J.W. von Gudenberg. Abstract syntax trees - and their role in model driven software development. In *Software Engineering Advances, 2007. ICSEA 2007. International Conference on*, pages 38 –38, 25-31 2007. doi : 10.1109/ICSEA.2007.12.

S. Gani and P. Picuri. The object revolution. how com technology is changing the way we do business. *Computing Control Engineering Journal*, 6(3) :108 –112, jun 1995. ISSN 0956-3385.

J. Gough. Virtual machines, managed code and component technology. In *Software*

Engineering Conference, 2005. Proceedings. 2005 Australian, pages 5 – 12, 29 2005. doi : 10.1109/ASWEC.2005.49.

Martin Grant. SystemC Tools, 2002. URL `www-ti.informatik.uni-tuebingen.de/~systemc/Documents/Presentation-6-OSCI4_martin.pdf`.

J. Halleux. Quickgraph, graph data structures and algorithms for .NET, 2006. URL `http://quickgraph.codeplex.com`.

Zhe Han and Shihong Chen. Constructing cdg for program with transfer statements. In *Informatics in Control, Automation and Robotics (CAR), 2010 2nd International Asia Conference*, volume 3, pages 122 –125, 6-7 2010. doi : 10.1109/CAR.2010.5456643.

N. Hatami, A. Ghofrani, P. Prinetto, and Z. Navabi. Tlm 2.0 simple sockets synthesis to rtl. In *Design Technology of Integrated Systems in Nanoscal Era, 2009. DTIS '09. 4th International Conference*, pages 3 –8, 6-9 2009. doi : 10.1109/DTIS.2009.4938013.

R. Hilderink and T. Grötker. Transaction-level modeling of bus-based systems with SystemC 2.0, 2002.

G. Jennings. A case against event-driven simulation for digital system design. In *Simulation Symposium, 1991., Proceedings of the 24th Annual*, pages 170 –176, 1-5 1991. doi : 10.1109/SIMSYM.1991.151502.

A. A. Jerraya and G. Nicolescu. *La spécification et la validation des systèmes hétérogènes embarqués*. Hermes, 2004. ISBN 2-7462-0820-2.

Peng-Cheng Kao, Chih-Kuang Hsieh, and A.C.-H. Wu. An RTL design-space exploration method for high-level applications. In *Design Automation Conference, 2001. Proceedings of the ASP-DAC 2001. Asia and South Pacific*, pages 162 –167, 2001. doi : 10.1109/ASPDAC.2001.913298.

K.M. Kavi, B.P. Buckles, and U.N. Bhat. A formal definition of data flow graph models. *Computers, IEEE Transactions*, C-35(11) :940 –948, nov. 1986. ISSN 0018-9340. doi : 10.1109/TC.1986.1676696.

Kyuseok Kim, Yongjoo Kim, Youngsoo Shin, and Kiyoung Choi. An integrated hardware-software cosimulation environment with automated interface generation. In *Rapid System Prototyping, 1996. Proceedings., Seventh IEEE International Workshop*, pages 66 –71, 19-21 1996. doi : 10.1109/IWRSP.1996.506729.

E. Kirda, C. Kerer, M. Jazayeri, H. Gall, and R. Kurmanowytsch. Evolution of an organizational web site : migrating to XML/XSL. In *Web Site Evolution, 2001. Proceedings. 3rd International Workshop*, pages 62 – 69, 10 2001.

K. Knack. Panel : Design automation tools for FPGA design. In *Design Automation, 1994. 31st Conference*, pages 676 – 676, 6-10 1994.

A. Kokrady, R. Mehrotra, T.J. Powell, and S. Ramakrishnan. Reducing design verification cycle time through testbench redundancy. In *VLSI Design, 2006. Held jointly with 5th International Conference on Embedded Systems and Design., 19th International Conference on*, pages 6 pp.–, Jan. 2006. doi : 10.1109/VLSID.2006.140.

Alexey Kupriyanov, Frank Hannig, and Jurgen Teich. High-speed event-driven rtl compiled simulation. In Andy D. Pimentel and Stamatis Vassiliadis, editors, *Computer Systems : Architectures, Modeling, and Simulation, 4th International Samos Workshop, SAMOS 2004, Proceedings*, volume 3133 of *Lecture Notes in Computer Science (LNCS)*, pages 519–529, Island of Samos, Greece, July 2004. Springer.

Cho Kwanghyun, Kim Jaebeom, Jung Euibong, Kim Sik, Li Zhenmin, Cho Young-Rae, Min Byeong, and Choi Kyu-Myung. Reusable platform design methodology for SoC integration and verification. *SoC Design Conference, 2008. ISOCC '08. International*, 01 :I–78–I–81, Nov. 2008. doi : 10.1109/SOCDC.2008.4815577.

J. Lapalme, E.M. Aboulhamid, G. Nicolescu, L. Charest, F.R. Boyer, J.P. David, and G. Bois. .net framework - a solution for the next generation tools for system-level

modeling and simulation. In *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, volume 1, pages 732 – 733 Vol.1, 16-20 2004. doi : 10.1109/DATE.2004.1268952.

Wang Laung-Terng, N.E. Hoover, E.H. Porter, and J.J. Zasio. Ssim : A software levelized compiled-code simulator. In *Design Automation, 1987. 24th Conference*, pages 2 – 8, 28-1 1987.

Seokwoo Lee, S. Des, V. Bertacco, and T. Austin. Circuit-aware architectural simulation. In *Design Automation Conference, 2004. Proceedings. 41st*, pages 305 – 310, 2004.

C.K. Lennard, V. Berman, S. Fazzari, M. Indovina, C. Ussery, M. Strik, J. Wilson, O. Florent, F. Remond, and P. Bricaud. Industrially proving the spirit consortium specifications for design chain integration. In *Design, Automation and Test in Europe, 2006. DATE '06. Proceedings*, volume 2, pages 1 –6, 6-10 2006. doi : 10.1109/DATE.2006.243839.

P.E. Livadas and S. Croll. System dependence graph construction for recursive programs. In *Computer Software and Applications Conference, 1993. COMPSAC 93. Proceedings., Seventeenth Annual International*, pages 414 –420, 1-5 1993. doi : 10.1109/CMPSAC.1993.404249.

G. Martin. Systemc : from language to applications, from tools to methodologies. In *Integrated Circuits and Systems Design, 2003. SBCCI 2003. Proceedings. 16th Symposium*, page 3, 8-11 2003.

M. Metzger, F. Bastien, F. Rousseau, J. Vachon, and E.M. Aboulhamid. Introspection mechanisms for semi-formal verification in a system-level design environment. In *Rapid System Prototyping, 2006. Seventeenth IEEE International Workshop*, pages 91 –97, 14-16 2006. doi : 10.1109/RSP.2006.22.

N. Nicolescu and E. Gabriela. *Spécification et validation des systèmes hétérogènes embarqués Thèse de doctorat TIMA, Techniques de l'Informatique et de la Microélectro-*

nique pour l'Architecture des ordinateurs. PhD thesis, Institut National Polytechnique de Grenoble - inpg, 2002.

G. Olsen and W. McDermith. Vhdl for fpga design. In *WESCON/'93. Conference Record*, pages 74 –79, 28-30 1993. doi : 10.1109/WESCON.1993.488412.

Samir Palnitkar. *Verilog HDL : a guide to digital design and synthesis, second edition*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2003. ISBN 0-13-044911-3.

T. J. Parr and R. W. Quong. Antlr : a predicated-ll(k) parser generator. *Softw. Pract. Exper.*, 25(7) :789–810, 1995. ISSN 0038-0644. doi : http://dx.doi.org/10.1002/spe.4380250705.

Terence John Parr. Enforcing strict model-view separation in template engines. In *WWW '04 : Proceedings of the 13th international conference on World Wide Web*, pages 224–233, New York, NY, USA, 2004. ACM. ISBN 1-58113-844-X. doi : http://doi.acm.org/10.1145/988672.988703.

Terence John Parr. A functional language for generating structured text. Draft, 2006. URL `http://www.cs.usfca.edu/~parrt/papers/ST.pdf`.

O. Pasquier and J.P. Calvez. An object-based executable model for simulation of real-time hw/sw systems. In *Design, Automation and Test in Europe Conference and Exhibition 1999. Proceedings*, pages 782 –783, 1999. doi : 10.1109/DATE.1999.761229.

S. Pasricha, Nikil Dutt, and M. Ben-Romdhane. Extending the transaction level modeling approach for fast communication architecture exploration. In *Design Automation Conference, 2004. Proceedings. 41st*, pages 113–118, 2004.

Douglas L. Perry. *VHDL*. McGraw-Hill, Inc., New York, NY, USA, 1998. ISBN 0070494363.

Lars Powers and Mike Snell. *Microsoft Visual Studio 2005 Unleashed (Unleashed)*. Sams, Indianapolis, IN, USA, 2006. ISBN 0672328194.

Virding Robert, Wikstrom Claes, and Williams Mike. *Concurrent Programming in Erlang*. Prentice Hall PTR, 2 edition, 1996. ISBN 013508301X.

Jeffry T Russell. Program slicing for codesign. In *CODES '02 : Proceedings of the tenth international symposium on Hardware/software codesign*, pages 91–96, New York, NY, USA, 2002. ACM. ISBN 1-58113-542-4. doi : http://doi.acm.org/10.1145/774789.774809.

Mangano Sal. *XSLT Cookbook, Second Edition (Cookbooks (O'Reilly))*. O'Reilly Media, Inc., 2005. ISBN 0596009747.

H.-J. Schlebusch. Systemc based hardware synthesis becomes reality. In *Euromicro Conference, 2000. Proceedings of the 26th*, volume 1, page 434 vol.1, 2000. doi : 10.1109/EURMIC.2000.874663.

L. Semeria and A. Ghosh. Methodology for hardware/software co-verification in c/c++. In *Design Automation Conference, 2000. Proceedings of the ASP-DAC 2000. Asia and South Pacific*, pages 405 –408, 2000. doi : 10.1109/ASPDAC.2000.835134.

V. Solovjev and M. Chyzy. Refined cpld macrocell architecture for the effective fsm implementation. In *EUROMICRO Conference, 1999. Proceedings. 25th*, volume 1, pages 102 –109 vol.1, 1999. doi : 10.1109/EURMIC.1999.794455.

Erik Stenman and Konstantinos Sagonas. On reducing interprocess communication overhead in concurrent programs. In *ERLANG '02 : Proceedings of the 2002 ACM SIGPLAN workshop on Erlang*, pages 58–63, New York, NY, USA, 2002. ACM. ISBN 1-58113-592-0. doi : http://doi.acm.org/10.1145/592849.592857.

Dirk Vermeersch, Dündar Dumlugöl, Peter Hardee, Takashi Hasegawa, Adam Rose, Marcello Coppola, Martin Janssen, Thorsten Grötker, Abhijit Ghosh, and Kevin Kranen. Functional specification for SystemC 2.0. In *Update for SystemC 2.0.1*, 2002.

Yuan Weihua, Cheng Lan, and Yang Zhenghua. Using dll for pc/104 data acquiring in labview platform. In *Electronic Measurement and Instruments, 2007. ICEMI '07.*

8th International Conference, pages 4–878 –4–881, aug. 2007. doi : 10.1109/ICEMI.2007.4351283.

Zhu Xinping and S. Malik. A hierarchical modeling framework for on-chip communication architectures [SOC]. In *Computer Aided Design, 2002. ICCAD 2002. IEEE/ACM International Conference*, pages 663–670, Nov. 2002. doi : 10.1109/ICCAD.2002.1167603.

S. Yoo and A.A. Jerraya. Hardware/software cosimulation from interface perspective. *Computers and Digital Techniques, IEE Proceedings* -, 152(3) :369 – 379, may. 2005. ISSN 1350-2387. doi : 10.1049/ip-cdt:20045113.

Luigi Zaffalon. *Programmation Synchrone de Systèmes Réactifs avec Esterel et les SyncCharts*. Presses Polytechniques et Universitaires Romandes, Lausanne, 2005.