Extraction de modèles pour la conception de systèmes sur puce

Jean-François Le Tallec

▶ **To cite this version:**

Jean-François Le Tallec. Extraction de modèles pour la conception de systèmes sur puce. Systèmes embarqués. Université Nice Sophia Antipolis, 2012. Français. NNT : . tel-00767040

HAL Id: tel-00767040
https://theses.hal.science/tel-00767040

Submitted on 19 Dec 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

techBibliographie technique

Université de Nice - Sophia Antipolis – UFR Sciences
École Doctorale STIC

THÈSE

Présentée pour obtenir le titre de :

Docteur en Sciences de l'Université de Nice - Sophia Antipolis

Spécialité : Informatique

par

Jean-François Le Tallec

Équipe d'accueil : AOSTE – INRIA / I3S

Extraction de modèles pour la conception de systèmes sur puce

Thèse dirigée par Charles André

Soutenance à l'INRIA le 25 Janvier 2012, à 10h30 devant le jury composé de :

Président :	Prénom	Nom	Université / Affiliation
Directeur :	Charles	André	Université de Nice Sophia-Antipolis
Rapporteurs :	Florence	Maraninchi	INP Grenoble / ENSIMAG
	Christophe	Moy	Supélec Rennes
Examinateurs :	Jean-Luc	Dekeyser	Université de Lille 1
	Robert	de Simone	INRIA Sophia Antipolis Méditerrannée

Thèse

Extraction de modèles pour la conception de systèmes sur puce

Model extraction for Systems On Chip Design

Jean-François Le Tallec

Janvier 2012

Résumé

La conception des systèmes sur puce s'appuie souvent sur SystemC/C++ qui permet des descriptions architecturales et comportementales à différents niveaux d'abstraction. D'autres approches se tournent vers l'automatisation de l'assemblage de plates-formes dites virtuelles (format IP-Xact). L'utilisation des techniques de l'ingénierie des modèles est une voie plus récente avec des profils UML tels que MARTE. Dans cette thèse, nous étudions les possibilités de modélisation de ces différentes approches et les passerelles disponibles entre elles. Motivés par la disponibilité de modèles SystemC et par les facilités offertes par MARTE, nous traitons de l'export des modèles SystemC. Au-delà de la simple conversion entre formats, nous décrivons la mise en oeuvre d'une passerelle entre l'implémentation SystemC d'un design et sa version modèle dans le format IP-Xact. La représentation IP-Xact peut ensuite être de nouveau transformée en modèles MARTE par des outils déjà existants. Nous présentons les travaux connexes avant d'exposer notre vision et sa réalisation au travers de l'outil SCiPX (SystemC to IP-Xact). Dans un second temps, nous présentons plus en détails les possibilités permises par le profil UML-MARTE, son modèle de temps et le langage de spécification de contraintes temporelles CCSL. Nous abordons les problèmes liés à la modélisation de protocoles à différents niveaux d'abstraction et plus spécialement ceux posés par le raffinement entre les niveaux TLM et RTL. Cette étude met en évidence des insuffisances de CCSL concernant la spécification des priorités. Nous proposons un enrichissement de CCSL pour lui permettre de manipuler ce concept de priorité.

Mots-clés : systèmes sur puce, conception électronique, modélisation, ingénierie des modèles, contraintes temporelles, temps logique, SystemC, IP-Xact, UML-MARTE, CCSL.

Abstract

The design of System on Chip mostly relies on SystemC/C++. This language allows architectural and behavioural descriptions at different abstraction levels. Others approaches consider automated assembly of components into actual or virtual platforms (IP-Xact format). Using Model Driven Engineering techniques is a new trend, which may benefit from UML profiles (especially MARTE). In this thesis, we study the modelling power of these approaches and the possible bridges between them. SystemC provides a great deal of examples while MARTE offers facilities in system modelling at different levels. So, we try to export SystemC designs to MARTE models. Beyond the mere conversion between formats, we propose an abstraction mechanism from SystemC code to models in IP-Xact formats. The IP-Xact description is then transformed into MARTE models with existing tools. We review related works and propose our solution leading to a dedicated tool called SCiPX (standing for SystemC to IP-Xact). In the second part of the thesis we apply the UML profile MARTE, its time model, and the associated language for specification of temporal constraints (CCSL) to specify interactions among components. A special attention is paid to protocol refinement. This study reveals a lack of CCSL for capturing the concept of priority. An improvement in the CCSL constraint solver is proposed to overcome this limitation.

Keywords : system on chip, electronic design, modeling, model-driven engineering, temporal constraints, logical time, SystemC, IP-Xact, UML-MARTE, CCSL.

Dédicace

A ma famille de sang et de coeur mais avant tout à ma soeur

Remerciements

Je tiens en premier lieu à remercier Charles André qui, malgré sa retraite bien méritée, a été présent jusqu'ici pour me prodiguer conseils et expertises. Merci à Robert de Simone pour ses critiques souvent obscures au premier abord mais ô combien constructives lorsque l'on en saisit le sens. A Julien DeAntoni pour son aide et sa manière de dire les choses sans passer par quatre chemins malgré nos divergences d'opinions. A Régis Gascon pour son côté faussement bougon et tous les moments de rigolades (et de réflexion bien sûr) passés dans le même bureau. A Patricia Lachaume pour sa joie de vivre et son côté maman poule fort apaisant. Aux personnes de l'équipe AOSTE (passé et présent) pour les diverses discussions tant scientifiques que plus générales. Aux différentes personnes de l'INRIA que j'ai croisées à la machine à café me faisant presque oublier que j'étais sur mon lieu de travail. A mes proches, pour leur patience et leur réconfort dans les moments de doutes. Plus généralement, merci à toutes les personnes qui ont rendues ce travail possible par le soutien, l'aide ou la critique. Et bien évidement, merci aux membres du jury pour avoir sacrifier de leur temps à l'étude de mes travaux et plus particulièrement à Florence Maraninchi pour l'attention particulière qu'elle y a prêtée et les améliorations qu'elle m'a permis d'y apporter.

Jean-François Le Tallec
Jean-Francois.Le_Tallec@inria.fr
Sophia Antipolis, France

Sommaire

Figures

Tables

Chapitre 1

Introduction

Ce travail s'inscrit dans le contexte de la conception de systèmes électroniques et de systèmes sur puce. Initialement conçus comme assemblages de circuits intégrés, des niveaux plus abstraits ont été introduits pour faire face à la complexité croissante de ces systèmes. On est ainsi passé à des modélisations à base de registres, puis à des assemblages de composants ou même de sous-systèmes. Les comportements ont également évolué des descriptions électroniques initiales vers des descriptions de transferts entre registres puis à des visions encore plus abstraites dites transactionnelles.

Sous des différences de forme parfois très importantes, les Systèmes sur Puce (*System-on-a-Chip*, SoC) partagent des principes généraux de construction assez forts, hérités de la tradition du design de systèmes électroniques. Ces principes établis permettent de composer plus rapidement des plates-formes matérielles en assemblant des composants préexistants (dits blocs IP, pour "*Intellectual Property*"), implantant des types de fonctionnalités connues ou des améliorations de ces mêmes types. L'idée est désormais reprise pour composer de manière similaire, à un niveau plus simple et plus abstrait, des *plates-formes virtuelles* (VPF), basées sur des *modèles de ces composants*. Les plates-formes virtuelles sont donc composées à partir de modèles de blocs IP pré-existants. Ces blocs peuvent représenter des unités de calcul (processeurs), de stockage (mémoire), des utilitaires spécifiques (UART, DMA,...) ou des périphériques (I/O). Ils peuvent également représenter des *unités d'interconnexion* pour les communications de données entre ces blocs dédiés au calcul. Ces unités d'interconnexion sont des bus simples ou complexes et parfois des réseaux sur puce (*Network On-Chip*, NoC).

En général les blocs de calcul et stockage ne sont jamais directement liés entre eux, mais toujours au travers de blocs d'interconnexion assurant cette liaison et leur communication. Que ces communications soient simples dans le cas d'interruptions ou plus complexes pour le cas

d'une transaction, le canal de communication est vu comme un simple fil (*wire*) et abstrait de la représentation. Les communications complexes entre blocs de calcul/stockage, traversant ce réseau d'interconnexion, donnent donc lieu à des transactions entre les blocs IP, régies par des protocoles, eux-mêmes souvent standardisés afin que le design de blocs IP élémentaires reste invariant vis-à-vis de leurs interfaces. Pour ces communications, le design définit une notion de maîtres et d'esclaves (ou initiateurs actifs et accepteurs passifs) ayant ou non la possibilité d'engager une transaction. En général on peut considérer les blocs de calcul (*processing*) comme maîtres et ceux de stockage et mémorisation comme esclaves, mais bien souvent dans des cas plus flous de la classification un même composant peut se voir maître ou esclave sur des ports distincts de son interface, selon certaines fonctions. La notion de maître/esclave est distincte de celle d'émetteur/récepteur, car un maître peut commander la lecture d'une valeur depuis une mémoire.

A ce niveau, on peut considérer que ces *communications* sont rarement purement synchrones, mais consistent en des rendez-vous successifs à chaque extrémité du réseau d'interconnexion, entre chaque composant et le réseau lui-même, qui lui modélise explicitement le transport des données (et sa latence). On peut aussi distinguer entre des *communications bloquantes* (parfois dénommées synchrones en logiciel), au cours desquelles l'initiateur envoie une requête nécessitant le retour d'un résultat (comme dans un appel de fonction à distance), et *communications non-bloquantes*, consistant en le simple envoi de messages ou d'événement, sans que l'initiateur attende de retour. Une transaction consiste donc en une succession de phases, rarement très complexes. Typiquement on a les phases suivantes : demande d'accès au medium d'interconnexion de la part du maître (avec arbitrage si plusieurs maîtres et temporisation jusqu'à ce que la ressource se libère) ; puis transmission de la requête, qui comporte en général l'adresse du ou des composants esclaves comme argument de type données, et qui est routée en fonction de celle-ci (à nouveau la disponibilité de la ressource est attendue) ; puis phase de communication de données (lecture ou écriture) ; enfin libération des ressources et validation de la transaction.

Ces concepts importants pour la compréhension comportementale fine sont souvent hors de portée de la modélisation pour des raisons de performances en simulation, et donc la conception de la plate-forme virtuelle s'en tient à une version simple des transactions, qui se ramènent à des communications élémentaires (de type lecture/écriture) entre blocs IP. En d'autres termes, la conception de l'architecture de la plate-forme reste relativement orthogonale au développement des composants IP eux-mêmes. La sémantique globale du système dépendra évidemment de l'union comportementale des composants et de la plate-forme, mais les incertitudes éventuelles sur cette dernière ne devraient (idéalement) que mener à un non-déterminisme temporel, pas fonctionnel. On fait généralement les hypothèses suivantes : deux transactions s'exécutant en parallèle ne doivent pas interférer l'une avec l'autre ; aucun choix sur l'instant ne sera en contradiction avec les évolutions du système (confluence du comportement).

Dans cette thèse, nous étudions trois approches de modélisation : SystemC, IP-Xact et UML-MARTE.

Le formalisme **SystemC** a été introduit pour rendre compte explicitement des comportements, aussi bien des composants (calcul et interconnexion) que des communications (bloquantes ou non), au niveau transactionnel. Il permet de rendre compte des aspects temporels (physiques) à des degrés divers selon la finesse de modélisation (et surtout de simulation) désirée. Il existe désormais de nombreux environnements de développement, tant académiques qu'industriels, pour la conception de plates-formes virtuelles. On peut alors y puiser des composants déjà décrits à différents niveaux d'abstraction afin de composer une plate-forme.

IP-Xact, un autre standard, permet de décrire les interfaces d'interconnexion et certaines propriétés des communications et des protocoles associés, par un langage de description d'architecture (*Architecture Description Language*, ADL) dédié. Son usage permet de définir certains types de bus et de protocoles dans leurs aspects interfaces qui seront reprises (idéalement) pour l'interconnexion des composants eux-mêmes au réseau.

Le profil **UML-MARTE** dédié à la modélisation et à l'analyse de systèmes embarqués temps réel possède des éléments distinctifs dédiés à la modélisation de plates-formes virtuelles à un niveau de conception très abstrait et descriptif. Il a été proposé dans le passé récent pour couvrir ce domaine des plates-formes matérielles.

La question d'une représentation comportementale de haut niveau basée sur les modèles (à la fois des composants et des interconnexions) est bien plus délicate que celle des aspects structurels (pour ne rester que dans le cadre fonctionnel). Une des difficultés est que le temps (latence, délais, temporisations) peut dans l'absolu avoir un rôle fonctionnel important, et impacter les comportements de manière définitive. A ce titre le modèle temporel de MARTE, et le langage de spécification de contraintes d'horloge (*Clock Contraint Specification Language*, CCSL) associé définissent une notion de temps logique, ou virtuel, dans lequel tout événement créant une suite d'occurrences et participant comme condition d'activation au déclenchement de traitements peut être vu comme une horloge (insistons : logique). C'est le cas de l'horloge d'un processeur moderne, qui peut être ralentie ou stoppée en fonction de l'énergie disponible dans la source de courant (batteries par exemple), mais la notion peut être généralisée et rendue indépendante du niveau de modélisation (plus aucun rapport alors avec le quartz d'un système électronique).

Organisation du mémoire

Dans le chapitre 2, nous étudions les possibilités de modélisation de ces différentes approches. Nous donnons, par la même occasion, les informations nécessaires à leur différenciation. *Notre objectif* étant d'utiliser les facilités offertes par le profil UML-MARTE pour modéliser les designs, il faut disposer de modèles de composants ou d'IPs dans ce format. Puisque la majorité des modèles existants sont spécifiés en SystemC, il nous a fallu considérer des passerelles (existantes

ou à améliorer) entre SystemC et MARTE. La figure 1.1 représente les passerelles déjà existantes (en noir) ainsi que celle que nous avons dû concevoir (en rouge) afin d'automatiser la navigation entre ces formats. IP-Xact joue le rôle d'un format intermédiaire.

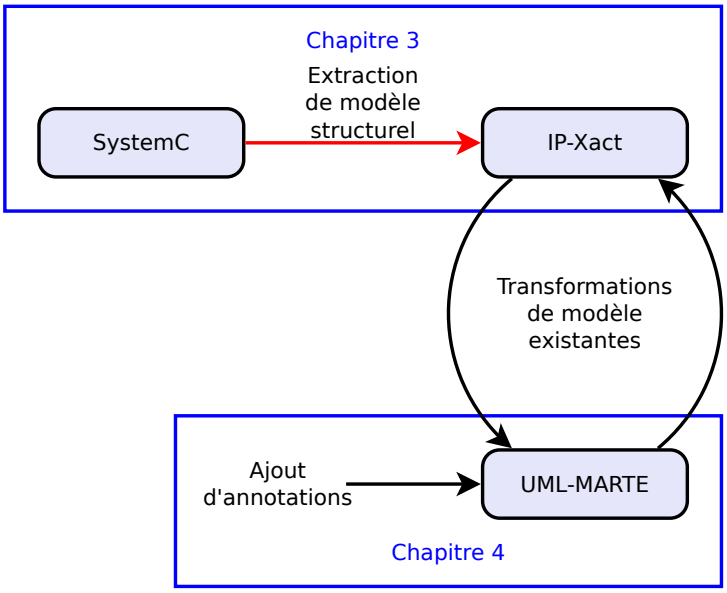


Figure 1.1 – Relations entre formats/modèles

Dans le chapitre 3, nous traitons de cette transformation. Nous présentons une étude des travaux existants pour ensuite exposer notre vision[?] et sa réalisation au travers de l'outil SCiPX[?] (*SystemC to iP-Xact*). Au delà de la simple conversion entre formats, cette passerelle est une extraction de la vue structurelle d'un design, permettant ainsi d'obtenir une version "modèle" (au sens de l'ingénierie des modèles) à partir de l'implémentation SystemC d'un design disponible sous la forme d'un code C++. Le format cible (IP-Xact) est ensuite transformé en modèles MARTE par des outils déjà existants.

Les possibilités d'application du profil UML-MARTE et de son modèle de temps sont exposées dans le chapitre 4. Nous y abordons les problèmes liés à la modélisation de protocole à différents niveaux d'abstraction. Nous nous posons la question du raffinement entre le niveau TLM et le niveau RTL[?] mettant en évidence un manque de CCSL vis-à-vis de la spécification de choix lors de comportements concurrents (prise en compte des priorités). Ce chapitre se termine par une proposition d'enrichissement de CCSL pour lui permettre de manipuler ce concept de priorité.

Le bilan des travaux est fait dans le chapitre 5. Nous y évoquons les pistes d'études non explorées dans les chapitres précédents et proposons de nouveaux axes de recherche pouvant mettre à profit ou étendre nos travaux. Ceci termine le corps du mémoire. Des documents illustratifs, des compléments et des démonstrations ont été reportés en annexe.

Chapitre 2

Conception de plates-formes virtuelles pour systèmes sur puce

Sommaire

2.1 ESL : conception de systèmes au niveau plate-forme

Les circuit digitaux deviennent de plus en plus de véritables systèmes-sur-puce, combinant et assemblant divers composants étant eux-mêmes des compositions d'autres composants, autour d'une infrastructure de communication alliant plusieurs bus, voire des réseaux-sur-puce de diverses topologies.

Face à cette complexification des designs, la conception des circuits de niveau RTL (*Register Tranfert Level*) précis au cycle et bit près se heurte au problème du temps nécessaire à la production d'une simulation fidèle, le test, ou la simple représentation de ces systèmes. Pour y remédier, un niveau de conception plus abstrait a été introduit, dénommé conception de design au niveau système (*Electronic System Level* ou ESL). Le concept central apporté via la conception ESL est celui des plates-formes matérielles virtuelles (*Virtual Hardware Platform* ou VHP), devant permettre de composer rapidement un système complet en intégrant par assemblage des composants préexistants propriétaires (dits *IP Blocks*) au moyen d'interface de communication basées sur des structures de connexion plus ou moins standard, modulaires ou paramétrables. Le lecteur peut se convaincre de l'intérêt ne serait-ce que d'un point de vue visuel en regardant la figure 2.1 représentant le même système mais aux deux niveaux différents, à gauche la version RTL et à droite la version TLM. Les différentes connexions élémentaires en RTL sont représentées par une connexion unique en TLM orientée de l'initiateur d'une communication vers la cible à interroger.

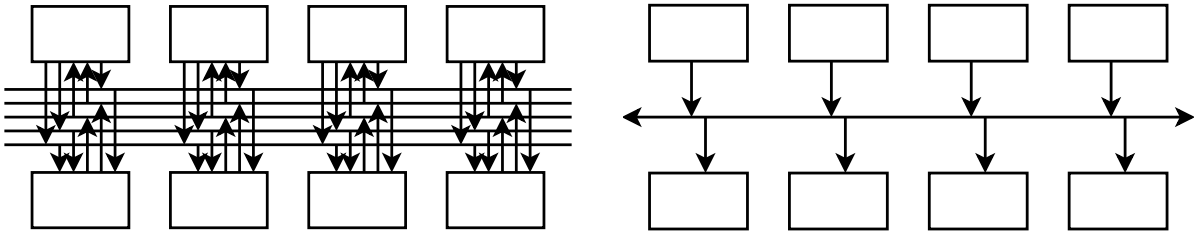


Figure 2.1 – Exemple de design RTL et TLM

Une multitude de logiciels privés se veulent précurseurs dans ce domaine. Parmi eux, on peut citer les outils *Virtualizer* [1] et *Platform Architect* [2] (Synopsys) ou encore un de leurs concurrents directs *Vista* [3] (Mentor Graphics).

Au-delà de l'aspect structurel simplifié, une idée forte est que la simulation à un niveau plus abstrait dit "transactionnel" (TLM, *Transaction Level Modeling*) permet des performances de simulation spectaculaires (comparées à une simulation *cycle accurate* RTL) tout en permettant la

1. `http://www.synopsys.com/systems/virtualprototyping/pages/virtualizer.aspx`
2. `http://www.synopsys.com/systems/architecturedesign/pages/platformarchitect.aspx`
3. `http://www.mentor.com/esl/vista/overview`

même interaction avec des versions bas niveau du code applicatif destiné à tourner sur la plate-forme finale (ultérieurement non virtuelle). Dans le même temps, les composants TLM peuvent être raffinés individuellement et indépendamment pour aboutir à une plate-forme complète RTL opérationnelle, testée "par morceaux".

Ces principes sont malheureusement traités moins idéalement dans la pratique. Les composants RTL sont souvent déjà existants et c'est le modèle TLM qui est alors créé. Dans les deux cas, un juste milieu doit alors être décidé entre le degré de fidélité du modèle TLM vis-à-vis de sa version RTL et son efficacité. C'est dans ce contexte qu'ont été définis et raffinés le langage SystemC dans son niveau TLM, ainsi que le langage de description d'architecture (ADL) IP-Xact, que nous allons décrire ci-dessous.

En plus de ces formalismes, qui utilisent le terme des modèles exprimés à l'aide de langages de programmation (et de la simulation) de systèmes, il nous semble devoir exister une place importante pour des modèles plus formels. Nous décrivons également les formalismes de cette nature présents dans la sphère de la conception ESL. Le propos principal ultérieur de cette thèse sera de considérer les ponts possibles entre ces deux types de modèles ainsi qu'au sein de chacun.

2.2 SystemC

SystemC est un langage de description de matériel/logiciel, permettant de représenter et de simuler les circuits et systèmes sur puce à différents niveaux d'abstraction (en particulier aux niveaux RTL et TLM)[?]. Conçu comme librairie C++, il bénéficie de types de données et d'environnements de compilation du langage C++. Il ajoute des primitives pour pouvoir décrire des processus parallèles, des signaux, des horloges, ainsi que certains concepts d'un langage à composants. Les comportements des modèles décrits sont alors exécutés par un ordonnanceur fourni avec la librairie SystemC. La simulation passe par deux phases successives. Tout d'abord, la phase d'*élaboration* instancie les processus parallèles et le réseau statique de composants (*i.e.,*les sc_modules et leurs interconnexions). Puis la *simulation* réelle prend place, activant les instances de composants en fonction de l'ordonnanceur et des événements générés par le système. La simulation peut être atemporelle (causale), précise au niveau cycle, approximative ou encore vaguement temporisée. SystemC a été au départ largement utilisé pour la modélisation de modèles formels de calcul et de communication (MoCC). On peut le relier aux précédents travaux de Daniel Gajski et de ses collègues autour de SpecC[?]. De plus, on peut trouver dans la spécification originale de SystemC des références aux Réseaux de Processus de Kahn (KPN). La modélisation de KPN, CSP (*Communicating Sequential Processes*) et systèmes réactifs synchrones en SystemC est également explorée dans les travaux de Fernando Herrera[?]. Les modélisations hétérogènes en SystemC sont aussi abordées dans les travaux de Sandeep K. Shukla[?]

et Fernando Herrera[?]. Cependant, les liens restent implicites, donnant lieu à un style de codage particulier pour chaque MoCC.

La phase d'élaboration peut être vue comme la construction des pièces structurelles de modèles (réseaux de processus et topologie d'interconnexion), tandis que la phase de simulation résulte de l'exécution de la composition de tous les comportements des composants.

Le langage de modélisation SystemC est développé par l'Open SystemC Initiative (OSCI) et est décrit dans la norme IEEE 1666-2005[?]. L'OSCI fournit une implémentation open-source de la librairie SystemC. Grâce à cette librairie, un modèle SystemC peut être compilé par un compilateur C^{++} standard et être exécuté. Cette exécution simule le modèle et fournit des informations qui peuvent être utilisées pour vérifier le comportement dynamique du modèle. Pour cela, la librairie SystemC se compose de deux choses : d'une part, des classes, macros et patrons permettant de décrire des systèmes concurrents communicants ; d'autre part, un noyau de simulation qui est utilisé pour exécuter un modèle SystemC. Nous étudierons en premier lieu la mécanique générale de fonctionnement d'un programme SystemC dans la section 2.2.1. Puis, nous nous intéresserons aux ressources de modélisation fournies par la librairie SystemC dans la section 2.2.2. Dans la section 2.2.3, nous terminerons par une description de l'extension officielle TLM qui facilite la modélisation au niveau transactionnel d'un design.

2.2.1 Présentation générale

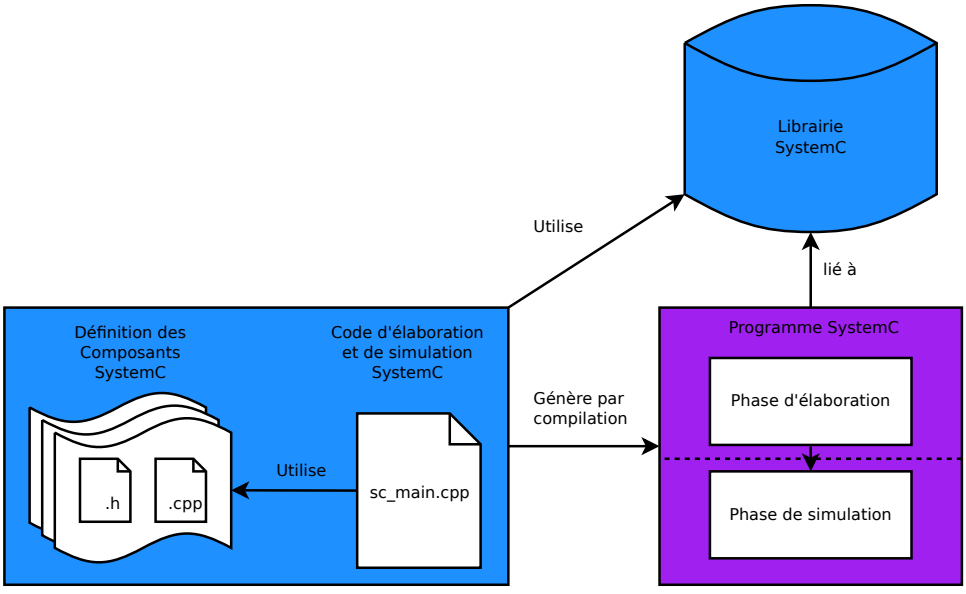


Figure 2.2 – Utilisation de la bibliothèque SystemC

Un programme SystemC est et reste un programme C++. Il doit inclure le fichier d'en-tête SystemC. Comme le décrit la figure 2.2, lorsque ce programme est compilé et lié avec la librairie

SystemC, une version exécutable du modèle est produite. Un développeur ne doit pas définir de fonction principale car cette fonction est déjà définie dans la librairie SystemC. Au lieu de cela, le développeur peut modifie sa fonctionnalité par l'intermédiaire de la fonction sc_main (celle-ci contiendra l'instanciation de son design). La fonction *main* est, elle, cachée dans la librairie SystemC. Quand un programme SystemC/C^{++} est exécuté, le *main* instancie tout d'abord le simulateur intégré à la bibliothèque SystemC. Il exécute ensuite la fonction sc_main écrite par le développeur. Par l'exécution de cette fonction particulière, les phases de construction puis simulation sont alors enchaînées. La phase d'élaboration consiste à construire le design tel que déclaré dans la fonction sc_main. Le constructeur de chacun des objets est alors exécuté et peut à son tour, créer et initialiser des sous-composants, des ports, des processus, des canaux et connexions liant leurs sous-composants. Tous les éléments qui héritent de sc_object (détaillé dans la section 2.2.2) sont alors enregistrés dans différentes structures, elles-mêmes référencées par la classe sc_simcontext (et conservées tout au long de l'exécution). Finalement, sc_start est appelée dans la fonction sc_main. Durant l'exécution de cette fonction, la méthode before_end_of_elaboration de chaque sc_object est exécutée suivie de l'exécution de la méthode end_of_elaboration de chaque sc_object (celles-ci vérifient et terminent d'initialiser le design). Ce n'est qu'alors que le design, défini dans le code SystemC du développeur, devient figé en terme de structure. Après cela, la fonction sc_start entre dans la phase de simulation et le comportement de l'ensemble du système est calculé par le simulateur événementiel instancié en premier lieu. Il génère alors la dynamique du système au fur et à mesure que lui arrivent les événements, réveillant les parties du design abonnées aux événements actifs.

Il est à noter que le simulateur décrit ci-dessus est dit *non-préemptif*. Si une fonction est réveillée elle s'exécute jusqu'à sa fin avant de rendre la main à l'ordonnanceur SystemC. Ceci peut occasionner des inversions de priorités conduisant à des comportements indésirable, voire incorrects. A cela s'ajoute la mise en œuvre de l'émulation du parallélisme du simulateur, dépendant de l'ordre de déclaration des objets du design (un exemple est donné en annexe C).

2.2.2 Description de la bibliothèque

L'architecture logicielle de la librairie est composée essentiellement de deux types de classes, les classes nécessaires à la description d'une architecture et les classes nécessaires à la simulation de cette architecture. Le premier type de classes regroupe les éléments essentiels d'un langage à composants. Il permet ainsi de décrire une architecture et son comportement. Ces classes peuvent être identifiées par leur héritage de la classe sc_core::sc_object. La figure 2.3 représente l'héritage immédiat de la classe sc_object mais plus généralement la majorité des éléments servant *a posteriori* la simulation héritent de cette classe mère. Le deuxième type de classes regroupe les mécanismes permettant de sauvegarder des informations sur les instances de premier type afin de pouvoir effectuer des tests de cohérence durant la phase d'élaboration ainsi que la simulation

de l'architecture durant la phase de simulation. Nous discuterons de ces différents types de classes dans les parties suivantes.

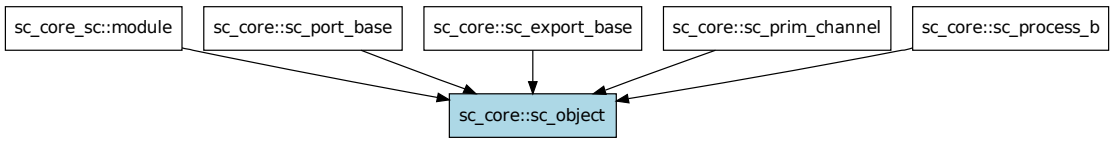


Figure 2.3 – Graphe d'héritage immédiat de sc_object

Un modèle SystemC est structuré en *modules*. Un module encapsule une partie du système modélisé et des ports de communication pour interagir avec d'autres modules dans le modèle. Un module peut contenir d'autres modules. Les modules sont définis par héritage de la classe sc_module. Un module peut avoir des points de connexion vers l'extérieur de sa structure. Ils sont appelés *ports* en tant que membres de données de sa classe. Un port doit hériter publiquement de la classe mère sc_port. A l'extérieur du composant, les ports peuvent alors être connectés par l'intermédiaire de *canaux*. Plusieurs canaux primitifs sont définis dans la librairie SystemC et héritent de la classe sc_prim_channel.

Les Composants (sc_core::sc_module) Un *composant* est un élément de base d'un ensemble plus complexe et peut être lui-même un assemblage de composants. On désigne par composant élémentaire un composant ne contenant aucun autre composant. Les autres sont dits composites. Un composant (élémentaire ou composite) en SystemC est un objet dérivant de la classe sc_module. L'héritage de la classe sc_module est nécessaire afin que le mécanisme interne de SystemC puisse enregistrer le composant lors de la phase d'élaboration. De plus, le constructeur par défaut d'un composant devra impérativement prendre en paramètre un nom unique afin de pouvoir *a posteriori* différencier les instances d'un même type de composant.

Les interconnexions (sc_core::sc_prim_channel) et interfaces Plus communément appelées canaux ou *channels*, les interconnexions doivent contenir/décrire les moyens de transferts d'information permis pour l'ensemble des entités qui s'y connectent.

Une *interface* est une classe dérivée de la classe sc_interface. Elle a pour but de contenir un ensemble de fonctions virtuelles. Ces fonctions servent par la suite à spécifier des points d'interactions d'un module afin qu'il puisse dialoguer avec l'extérieur. Ces fonctions restent virtuelles et devront être implémentées dans les canaux de communication dérivant de cette interface.

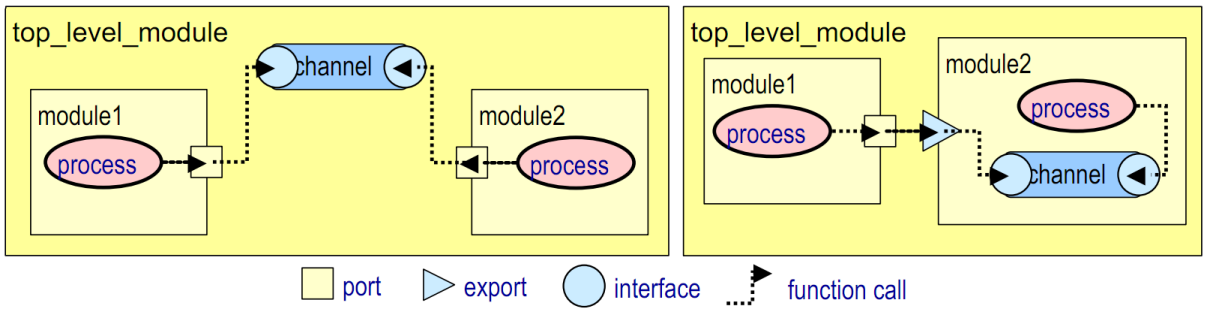


Figure 2.4 – sc_port & sc_export

Les ports et exports (sc_core::sc_port_base et sc_core::sc_export_base) Les ports (dérivant de la classe sc_port) sont des éléments perméables à certains événements explicitement connectés. Ils se situent sur les bordures externes des modules et permettent ainsi la propagation d'information au travers des parois du module. Lorsqu'un composant souhaite propager le résultat d'un calcul vers l'extérieur, il utilise une méthode d'un de ses ports. Celle-ci transmet alors automatiquement l'appel au canal externe auquel est relié le port. Un port définit un ensemble de services (relatifs au type du port) qui sont *requis* par le module contenant le port. Si un module doit appeler une fonction membre appartenant à un canal qui est en dehors du module lui-même, cet appel doit être fait en utilisant un appel de méthode d'interface via un port du module. Si un module doit appeler une fonction membre appartenant à une instance de canal dans un module fils, cet appel doit être fait au travers d'un export du module fils. La classe sc_export permet alors à un module fils de fournir une interface à son module parent. Les méthodes de l'interface d'exportation transmettent les appels au canal lié. La figure 2.4, extraite de la documentation officielle SystemC, donne un aperçu des constructions possibles.

Les processus (sc_core::sc_process) Le comportement d'un module SystemC est spécifié par un ou plusieurs *processus* SystemC. Un processus SystemC est défini sous la forme d'une fonction membre qui est enregistrée par le noyau de simulation SystemC à l'aide des macros SC_THREAD, SC_CTHREAD, ou SC_METHOD. Chaque processus dispose d'une liste de sensibilité qui est une liste d'événements SystemC. Un événement est quelque chose qui arrive au cours de l'exécution dans le temps, par exemple un changement de valeur sur un port d'entrée.

Un processus SC_METHOD est démarré par le noyau de simulation SystemC chaque fois qu'un événement auquel il est sensible se produit. Il s'exécute toujours intégralement avant de rendre le contrôle au noyau de simulation. De plus, il s'exécute en "temps nul", c'est-à-dire que le temps global de simulation reste inchangé après son exécution.

Un processus SC_THREAD n'est lancé qu'une fois par le noyau de simulation et ne se termine

qu'à la fin complète de la simulation. Un processus SC_THREAD peut être suspendu en des points statiquement définis dans le code SystemC. Ces points sont identifiables par l'appel de la fonction d'attente wait(). Le processus SC_THREAD est repris par le noyau de simulation lorsque l'un des événements sur sa liste de sensibilité se produit ou bien lorsque le délai passé en paramètre de la fonction wait() est écoulé. Un processus SC_CTHREAD est un type spécial de SC_THREAD qui n'est sensible qu'à un certain type d'événement provenant d'un port d'entrée horloge.

2.2.3 l'extension TLM

Les temps de simulation d'un design décrit au niveau RTL n'étant pas assez performants, une vue plus abstraite des communications a été introduite : le *niveau transactionnel*. Ce niveau permet d'abstraire un ensemble de connexions en une connexion simple orientée du demandeur d'un service vers le fournisseur du service. Dans le langage commun des concepteurs de plates-formes, on utilise plutôt les termes de maîtres (pour le "client") et d'esclaves (pour le "serveur"). La librairie SystemC n'imposant aucune contrainte concernant la définition de ces nouveaux types de port de communications, plusieurs versions d'implémentations de ceux-ci ont vu le jour (à peu près autant que de librairies de composants existantes). Il existe désormais une version officielle TLM, extension de la librairie SystemC d'origine. Elle introduit de nouveaux types de connexion. Les informations de cette section sont en partie extraites de la documentation officielle de TLM 2.0[?]. Quatre concepts de base émergent en TLM : les *Interfaces*, les ports TLM (appelés *Sockets*) et leur connexions, les transactions et enfin le *Protocol*.

Les *Interfaces* contiennent la définition des fonctions primaires de transport qui sont utilisées pour la définition du protocole. Elles sont implémentées sous la forme de méthodes (fonction membre) du modèle TLM. On en distingue généralement deux types : les fonctions de transport bloquantes et les non-bloquantes. Les bloquantes stoppent l'exécution du *client* tant que le *serveur* le nécessite. Le *serveur* hérite alors de l'autorisation de s'exécuter qu'avait obtenu le *client* de la part de l'ordonnanceur SystemC. Les non-bloquantes, pour leur part, ne font que notifier au *serveur* la tâche qu'il a à accomplir tout en permettant au *client* de continuer de s'exécuter.

Les *Sockets* sont les points de connexion des modèles TLM avec leur environnement. Afin de différencier ces connexions de simples connexions filaires, il existe deux types de connexions représentatives du rôle du composant dans la transaction. Les composants "initiateurs" ou "maîtres" et les "targets" ou "esclaves" doivent utiliser la connexion qui leur correspond (figure 2.5). Les connexions entre composants sont décrites comme allant d'un "maître" vers un "esclave" mais représentent un échange d'information bilatéral. Pour la version basique, une connexion met en jeu deux chemins de données. Le chemin aller est piloté par l'initiateur de la transaction. Le chemin de retour, lui, est piloté par la cible de la transaction. Quoi qu'il en soit, la

représentation courante adoptée est une connexion simple orientée du maître vers l'esclave. Des

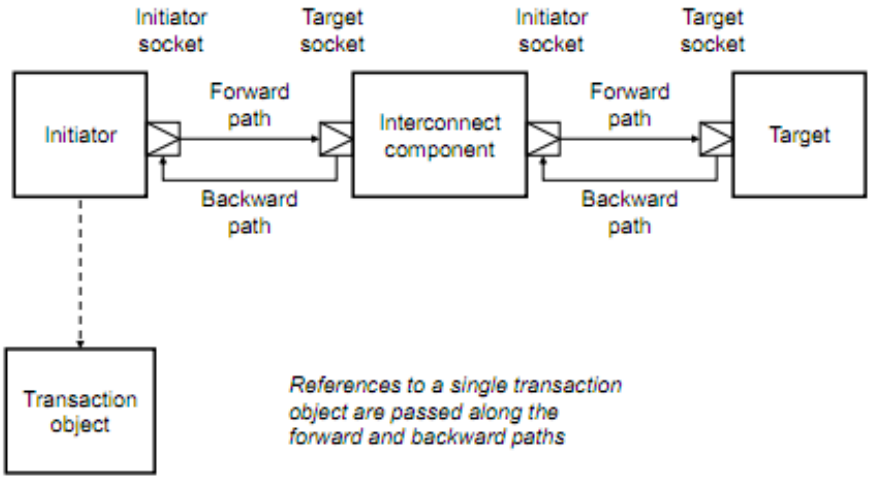


Figure 2.5 – Illustration des connexions par socket en SystemC TLM

Sockets *initiator* et *target* connectés doivent impérativement partager la même interface pour être connectés. Le designer ne crée aucun canal explicitement pour lier un socket maître et un socket esclave, la liaison se fait de la manière suivante : `initiator->init_socket.bind(target->targ_socket);` où *initiator* et *target* sont respectivement des pointeurs sur les composants SystemC initiateur et cible des transactions.

Le *Payload* est la structure qui va servir à contenir les informations d'une *transaction* tout au long de sa durée de vie (*i.e.*,du moment où elle débute jusqu'à sa complétion). On mémorise dans le *Payload* les informations de contrôle et de données de la transaction ainsi que certaines autres informations relatives à son statut telles que la phase dans laquelle elle se trouve.

Pour finir, le *Protocol* est l'entité dans laquelle on définit le fonctionnement réel d'une transaction. Les enchaînements d'appels aux fonctions de transport définis par les interfaces y sont spécifiés. Plusieurs manières de faire sont possibles et cela a donné lieu à plusieurs versions aux noms qui se veulent représentatifs du niveau d'abstraction modélisé. Ainsi, on rencontre dans la littérature les appellations *Untimed* (UT), *Programmers' View* (PV), *Programmers' View with Timing* (PVT), *Bit Accurate* (BA) ou encore *Cycle Accurate* (CA). Afin de clarifier tout cela, la spécification TLM introduit deux styles de modélisation temporelle : le *Loosely Timed* (LT) pour une modélisation ayant des performances en temps de simulation remarquables mais au détriment de la précision temporelle ; le *Approximately Timed* (AT) moins rapide que le premier mais plus réaliste. Le premier niveau utilise des transports bloquants pour éviter les changements de contexte et les DMI (*Direct Memory Interface*) pour découpler la lecture des instructions par un processeur et les transferts de données. Le second fournit les mécanismes

pour décrire les différentes phases d'une transaction et utilise plutôt des transports non bloquants pour offrir la possibilité aux composants de se resynchroniser sur le temps global de la simulation.

2.3 MBD : Model-Based Design

La conception dirigée par les modèles est une manière d'aborder le design de systèmes complexes. Elle est utilisée dans différents domaines tels que le traitement du signal, les systèmes de communication, l'aérospatial et même les logiciels embarqués. Elle se base sur des modèles mathématiques formellement définis et ayant un sens dans un domaine considéré. On trouve cette méthodologie dans un des premiers livres traitant de la modélisation de design en SystemC[?] sous le nom de MoC (*Model of Computation*). En particulier, le MoC "DE" (*Discrete Event*) y est traité. Ce dernier englobe le modèle de calcul utilisé pour représenter du matériel au niveau RTL (mis en œuvre dans les simulateurs événementiels de Verilog et VHDL) ainsi que le niveau TLM si l'on considère qu'une transaction peut être vue comme ayant une date de début, une date de fin et une durée.

La communauté de l'*ingénierie des modèles* se rapproche de cette philosophie en proposant de séparer clairement un méta-modèle de son modèle. Le méta-modèle définit les concepts nécessaires à la description du modèle lui-même. Il est aussi possible de regrouper différents méta-modèles sous un autre méta-modèle plus général. Cependant, ces représentations restent centrées sur la spécification de ce qu'il est permis de décrire (ou non) en termes de structure/-composition.

C'est l'approche qui a été adoptée à l'Université de Camtabria. Les chercheurs ont débuté par l'implémentation de différents MoCs en SystemC[?,?]. Ils ont ensuite abordé le problème du mélange de différents MoCs[?], ce qui donna lieu à un framework[?] ainsi qu'à la définition de transformations valides entre différents MoCs. Profitant de ces expertises, ils se sont ensuite tournés vers des environnements UML notamment dans le projet *SATURN* dont le but était de construire une passerelle entre la modélisation et la vérification/synthèse de systèmes embarqués décrits par des modèles UML (aussi bien le logiciel que le matériel).

C'est donc naturellement que s'est imposé le choix d'étudier les possibilités permises par ce genre d'approche dans le domaine de la modélisation de systèmes embarqués. Nous vous présenterons dans la suite de ce chapitre, deux formats issus de cette approche. Nous commencerons par le standard IP-Xact (section 2.4) qui est un format standard de description architecturale. Il se pose en leader parmi les industriels et bénéficie d'un effort particulier de la part des outilleurs en terme de compatibilité. D'autre part, nous présenterons le profil UML-MARTE (section 2.5), qui est d'origine académique mais d'application industrielle (standardisation internationale OMG (*Object Management Group*)).

2.4 IP-Xact

IP-Xact est une proposition de standardisation pour un ADL (*Architecture Description Language*) permettant l'assemblage de composants IP (*Intellectual Property blocks*, souvent d'origine commerciale et fournis comme boîtes noires) au sein d'une plate-forme matérielle virtuelle modélisant un Système sur Puce complet. La particularité première de ce format est que ces blocs IPs sont en général interconnectés au moyen de structures de communication (*interconnect fabric*) clairement définies, même si elles ne sont pas uniques et souvent même commercialement concurrentes entre elles. On peut citer les bus AMBA de ARM, les bus OCP-IP, en plus des nouveaux réseaux sur puce (*Networks On-Chip*). Le format IP-Xact comporte deux niveaux. Tout d'abord, un premier niveau ou l'on considère la possibilité de décrire et construire les éléments d'interface et de protocole d'un réseau d'interconnexion donné (qui pourrait ensuite figurer en bibliothèque). Et dans un second niveau, on s'assure qu'un nouvel IP proposé respecte la discipline IP-Xact imposée pour ce protocole, et les interfaces qui en découlent. A cette condition le composant pourra être facilement intégré dans toute plate-forme matérielle virtuelle basée sur la structure d'interconnexion concernée. Les interfaces peuvent être paramétriques (*e.g.,*un bus par le nombre d'IPs maîtres ou esclaves qui s'y attachent) et également être annotées par des informations additionnelles non-fonctionnelles. Ces annotations restent malheureusement souvent non-standardisées et décrites comme extensions particulières (*vendor extensions*). Ce dernier terme marque bien la finalité commerciale du standard, résultat de négociations entre sociétés de technologie du domaine au sein du consortium SPIRIT puis par le consortium Accellera, ce qui fait que le standard (IEEE 1685) manque de certaines qualités de clarté et d'orthogonalité des concepts. Récemment, le consortium Accellera a lui-même fusionné avec l'OSCI (*Open* S*ystem*C *Initiative*), ce qui pourrait augurer d'un alignement entre IP-Xact et SystemC.

2.4.1 Présentation générale

IP-Xact est donc un standard dédié à la conception, l'intégration et la réutilisation d'IPs en permettant la configuration automatique et l'intégration à l'aide d'outils industriels. Le standard IP-Xact spécifie les règles de conformité de modèles de systèmes électroniques décrits en XML (*eXtensible Markup Language*). Les formes de méta-données qui y sont normalisées comprennent d'une part les concepts nécessaires à la description d'un modèle d'architecture tels que les composants, un design, les interfaces et interconnexions, et d'autre part les chemins d'accès aux fichiers d'implémentations (écrits dans un langage tel que Verilog, VHDL, ou encore SystemC) pour permettre un assemblage automatisé d'un design. Il inclut de surcroît un ensemble de schémas XML (XSD) et de règles de cohérence sémantique (*Semantic Coherence Rules*) qui jouent le rôle de méta-modèle et permettent une validation structurelle d'un design. Ce format de représentation vise à rester indépendant des outils d'assemblage et du langage de description

choisis. Il offre donc une vue générique d'un modèle d'architecture qui peut être dans l'absolu importé/produit dans/par divers outils.

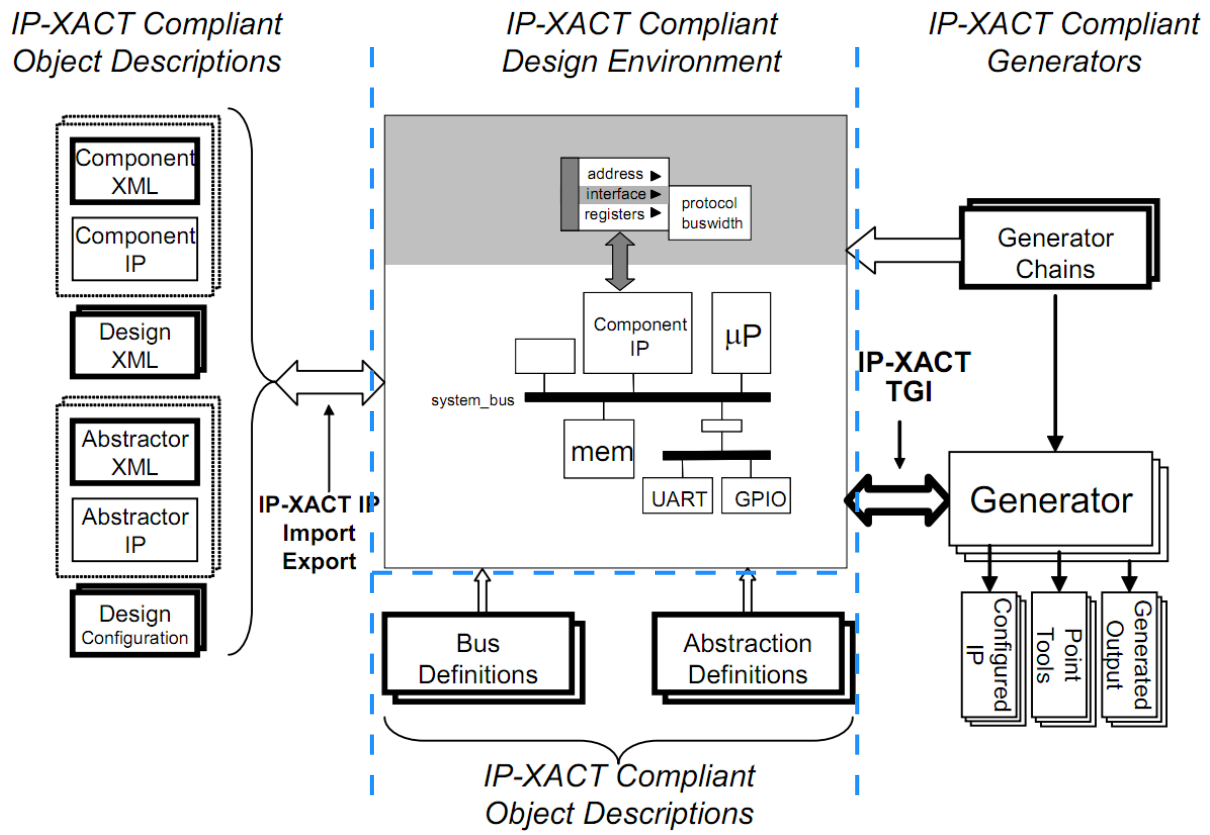


Figure 2.6 – Utilisation du format selon le consortium SPIRIT

La figure 2.6, extraite du standard officiel[7] donne un aperçu de l'utilisation d'un tel format dans un flot général plus complet. La partie centrale représente la vue d'un design au format IP-Xact. Celui-ci utilise des définitions de composants (partie gauche) dont les interconnexions sont, elles aussi, décrites en IP-Xact. Des mécanismes permettent de fournir de manière générique des informations concernant des interconnexions (partie basse) complexes telles que les bus (partie inférieure de la figure). Ensuite, toutes ces descriptions peuvent être liées à des outils de génération (partie droite) par l'intermédiaire d'une interface standardisée (partie droite de la figure).

2.4.2 Description du format

Le format IP-Xact contient sept types de modèles. Les règles de construction syntaxiques de chacun sont spécifiées en XSD (XML *Schematic Description*). Plutôt que les schémas XSD du

standard, nous avons préféré retenir une méta-modélisation plus proche d'UML. Les diagrammes présentés sont extraits de la thèse d'Aamir Mehmood Khan[?]. Au préalable nous énumérons les sept types de description et leur rôle :

- *bus definition description* pour les types d'un bus ;
- *abstraction definition description* pour la représentation plus détaillée d'un bus ;
- *component description* pour la structure d'un composant ;
- *design description* pour les systèmes et sous-systèmes ;
- *abstractor description* pour des adaptations entre différentes interfaces ;
- *generator chain description* pour la partie génération de l'assemblage ;
- *design configuration description* pour des informations complémentaires la *generator chain* ou la *design description*.

Dans cette section, nous nous concentrons sur les concepts essentiels à la construction d'un modèle de design. La figure 2.7 donne un aperçu global des paquetages IP-Xact nécessaires ainsi que leurs inter-dépendances. Chacun de ces paquetages correspond à un concept différent d'IP-Xact. Le paquetage Component contient les éléments qui décrivent la structure d'un composant (ou "IP"). Il est étroitement lié aux informations contenues dans les paquetages BusDefinition et AbstractionDefinition. Le paquetage BusDefinition donne les informations de base concernant un bus de communication utilisé par le système, alors que l'AbstractionDefinition donne des informations relatives au niveau d'abstraction de communication utilisé (*i.e.,*TLM, RTL). Le paquetage Abstractor contient les éléments nécessaires à la création/sélection de passerelles de communications entre différents composants IP-Xact décrits à différents niveaux d'abstraction. Il décrit en particulier comment un port transactionnel regroupe un ensemble de ports de base (RTL) représentant les signalisations nécessaires à la transaction. Enfin, le paquetage Design contient les divers éléments de modèle permettant de manipuler, référencer et instancier les composants utilisés pour l'élaboration d'un système.

Des propriétés fonctionnelles et extra-fonctionnelles peuvent être intégrées dans une définition de composant ou d'un design en utilisant ce qu'on appelle des *vendor extensions*. Le format et le type d'information contenu dans une *vendor extension* propriétaire ne font pas partie de la norme. Ces informations ne peuvent donc pas être partagées entre différents outils et fournisseurs sans l'implémentation du mécanisme de reconnaissance et d'interprétation spécifique. Cela empêche l'intégration des IPs développées à la main ou au travers des différents outils disponibles. Une autre limitation, mise en évidence dans la thèse de Sébastien Revol[?], est l'absence de mécanisme permettant de gérer facilement les structures répétitives comme pour des tableaux de registres ou d'instances d'un composant. Ces limitations et d'autres qui seront révélées dans la description détaillée d'IP-Xact nous incitent à penser que les formalismes de définition plus génériques tels que ceux qu'on trouve en UML pourraient être utiles.

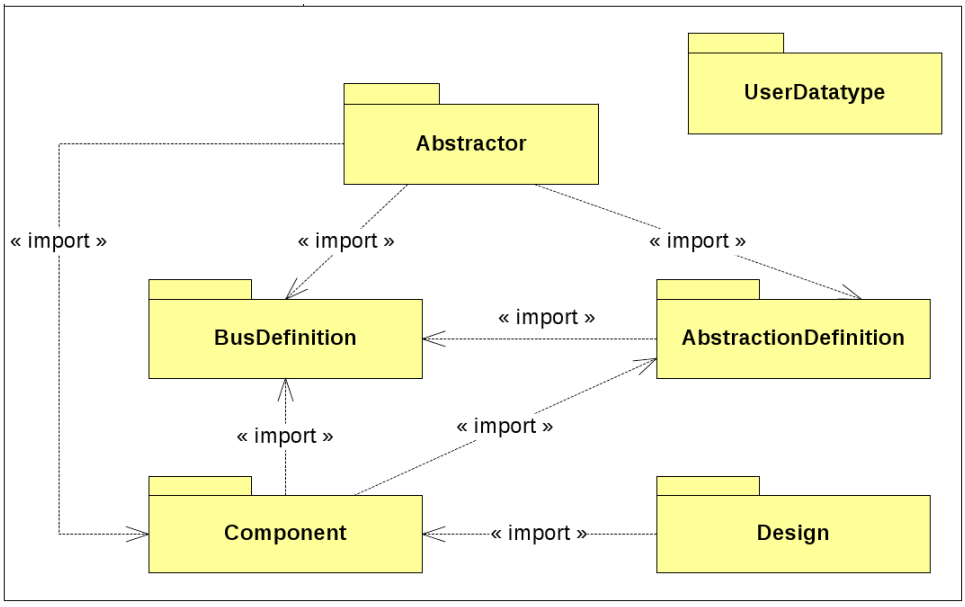


Figure 2.7 – Aperçu du méta-modèle IP-Xact

Composant En IP-Xact, un processeur, un périphérique, un élément de stockage, un bus, tous sont des composants. En général, les composants peuvent être statiques ou configurables. Les composants statiques ne peuvent pas être modifiés mais seulement instanciés. En revanche les composants configurables peuvent nécessiter un paramétrage lors de leur instanciation. Contrairement à d'autres modèles de composants, IP-Xact ne permet pas d'intégrer directement un composant dans un autre. Pour modéliser une hiérachie il faut passer par un design (voir page 22).

La figure 2.8 montre une vue simplifiée du méta-modèle Component. Un composant peut posséder de nombreuses propriétés dont des interfaces, des descriptions d'espace mémoire et indirectement des ports physiques. Il contient également des références aux descritions de son comportement. La seule propriété obligatoire est son identificateur unique (identifiant vlnv pour Vendor, Library, Name, Version) qui permet de le référencer.

Les interfaces Les ports qui participent à un même protocole de communication sont regroupés sous le concept de *Bus Interface*. Les attributs d'une interface de bus sont spécifiés dans deux méta-classes distinctes mais complémentaires : BusDefinition et BusAbstraction. Le concept de *Bus Interface* introduit une vision abstraite des communications avec l'introduction des *ports logiques*.

La définition de bus L'élément busDefinition décrit les attributs de haut niveau des interfaces connectées à un bus ou réseau. Il définit les informations structurelles associées à un type de bus, indépendant du niveau d'abstraction, comme le nombre maximum de maîtres et

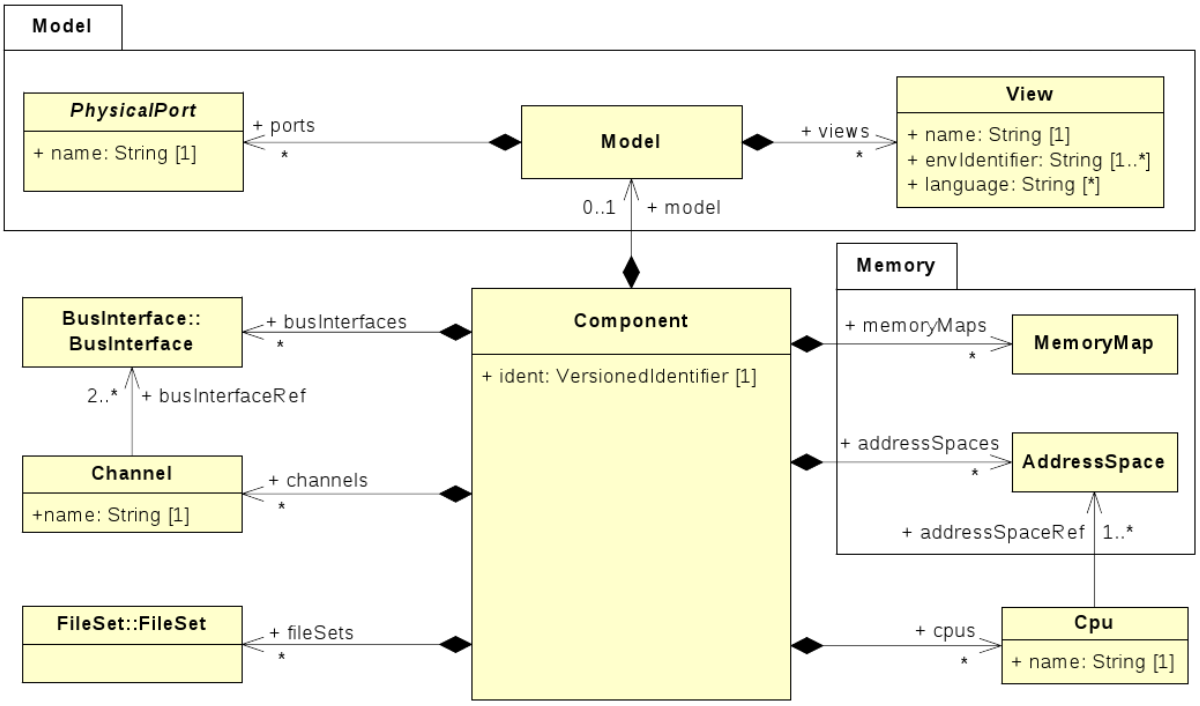


Figure 2.8 – Méta-modèle de composants IP-Xact

d'esclaves admis sur le bus. Une variable booléenne directConnection obligatoire précise quelles connexions sont autorisées. Une valeur vraie spécifie que ces interfaces peuvent être connectées directement d'un maître à un esclave alors qu'une valeur fausse indique que la connexion ne peut se faire que vers un miroir de l'interface considérée. Le dernier booléen obligatoire est la variable isAddressable qui spécifie, si elle est à vrai, que le bus possède des informations d'adressage mémoire. IP-Xact fournit également un mécanisme visant à étendre ces définitions de bus. L'extension d'une définition de bus permet la définition des règles de compatibilité avec les bus existants. Par exemple, la définition d'un bus AHB (*Advanced High-performance Bus*) peut étendre la définition de sa version simplifiée l'AHBlite. Une faiblesse notoire de ce type d'extension est de ne pas s'appuyer sur la généralisation/spécialisation que l'on rencontre dans les modèles de classes. La notion d'héritage n'est pas supportée en IP-Xact.

La définition d'abstraction L'AbstractionDefinition décrit les aspects de bas niveau d'un bus. Une AbstractionDefinition donne des attributs plus précis pour une définition de bus donné. Il peut y avoir plusieurs définitions d'abstraction pour la définition d'un même bus, par exemple une pour la version RTL et une autre pour la version TLM. Une AbstractionDefinition doit contenir deux éléments obligatoires, son busType et ses ports. L'attribut busType donne la référence au

BusDefinition pour lequel cette AbstractionDefinition est définie. Comme pour busDefinition, une AbstractionDefinition peut étendre une autre AbstractionDefinition modulo certaines contraintes. L'AbstractionDefinition peut étendre ou modifier la définition de ports logiques (à ne pas confondre avec les ports physiques que possède un composant), en ajouter de nouveaux, ou en rendre certains illégaux.

Le design Un design IP-XACT est un assemblage d'objets composant. Il représente un système ou un sous-système définissant l'ensemble des instances de composants et leurs interconnexions, (figure 2.9). Les interconnexions peuvent être entre interfaces ou entre ports de composants. Les instances de composants référencent leur description générique. Comme ces éléments du design sont des sous-systèmes potentiellement paramétrés, ils peuvent être configurés pour répondre aux besoins spécifiques du design. Ces valeurs de configuration sont spécifiées à l'aide d'éléments configurableElementValues (référencés par les instances de composants). Le concepteur peut alors connecter les composants en utilisant différents types d'éléments de connexion. Les différents types de connexions possibles sont l'Interconnection, MonitorInterconnection, AdHocInterconnection et HierConnection. L'Interconnection est une simple connexion point à point entre *Bus Interfaces* de deux composantes compatibles. Elle est par conséquent la plus utilisée pour la connexion de composants dans un design pourvu d'un bus de communication. Cependant, sa nature point à point lui interdit tout broadcast d'informations à des cibles multiples. Le MonitorInterconnection est un type spécial d'interconnexion qui spécifie la connexion entre une interface et une liste d'interfaces de supervision/monitorage. L'AdHocConnection relie deux ports directement, les ports de type *wire* mais aussi les ports transactionnels, sans l'aide d'une interface de bus. Il peut aussi connecter des ports (internes) d'instance de composants contenus dans un composant aux ports (externes) de ce dernier dans le cas d'un composant hiérarchique. Enfin, les connexions hiérarchiques (HierConnection) connectent des composants provenant de différents niveaux hiérarchiques au travers d'une interface de bus.

Les Abstracteurs L'Abstractor est l'élément IP-Xact de plus haut niveau qui soit inclus dans le format. Il est utilisé pour adapter les communications entre deux types d'interfaces de bus ayant un niveau d'abstraction différent et partageant les mêmes types de bus. Des designs contenant des composants décrits à différents niveaux doivent contenir ce type d'éléments. Un abstracteur contient deux interfaces, qui doivent être la définition du bus et les définitions d'abstraction des différents niveaux. Contrairement à un composant, un abstracteur n'est pas référencé à partir d'un design, mais plutôt référencé par une configuration d'un design. L'élément de modèle abstracteur ressemble beaucoup à une interface de bus.

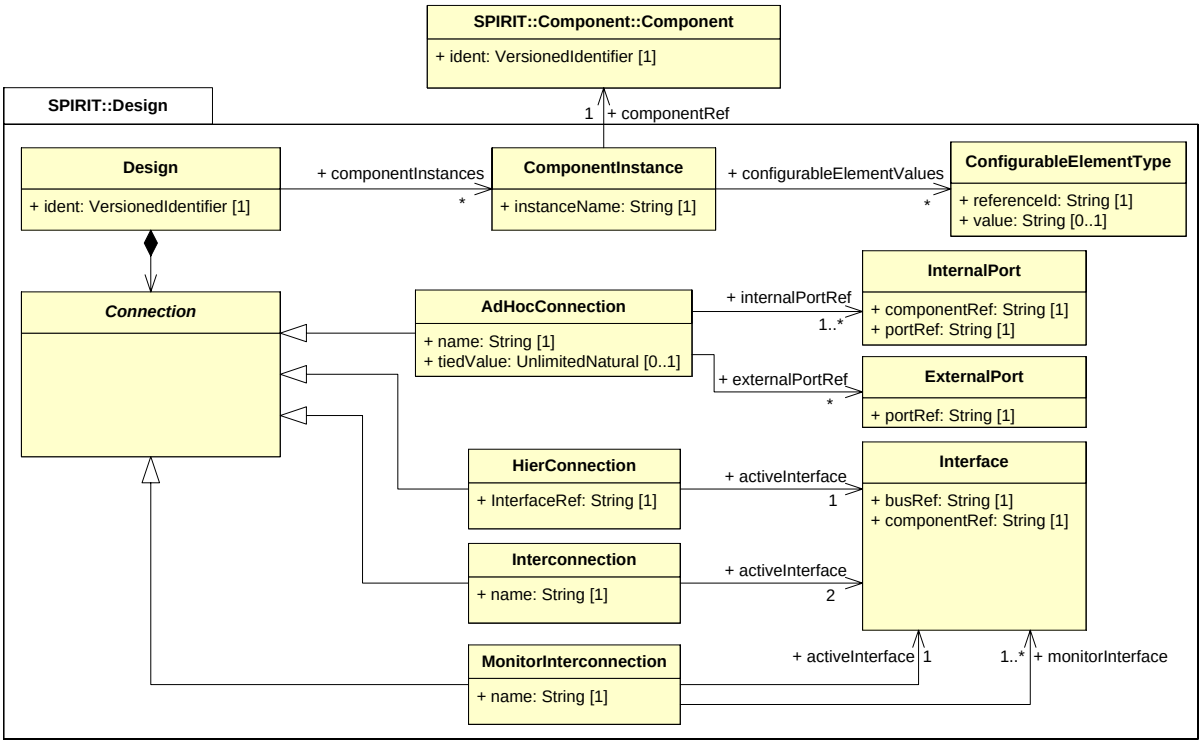


Figure 2.9 – Méta-modèle de design IP-Xact

2.5 Le profil UML-Marte

MARTE[?] (*Modeling and Analysis of Real-Time and Embedded*) est un profil UML qui a pour but de permettre la modélisation et l'analyse de systèmes temps réel embarqués. MARTE remplace et étend le profil UML *Schedulability, Performance and Time* (SPT[?]). MARTE traite de nouvelles exigences telles que la spécification des modèles logiciels et matériels, la séparation entre modèles abstraits d'applications et plates-formes d'exécution, la modélisation de l'allocation d'un modèle d'application sur un modèle de plate-forme, ainsi que la modélisation de différentes notions de temps et de propriétés extra-fonctionnelles. Il hérite de plusieurs aspects UML (tels que les composants pour la conception structurale, les *FSMs* hiérarchiques et des *data-flow graphs* pour les comportements). Il fournit, de plus, des fonctions d'annotation standard (appelées "stéréotypes") pour représenter les propriétés fonctionnelles et extra-fonctionnelles, ainsi que des moyens pour en introduire de nouvelles. En conséquence, une spécialisation plus poussée de MARTE permettrait l'encodage des notions IP-Xact telle que VendorExtension, mais cette fois de manière publiquement accessible et interprétable. Les méthodes issues de l'ingénierie des modèles concernant les transformations de modèle devraient ensuite permettre à l'information pertinente d'être redirigée vers n'importe quel outil propriétaire capable de com-

prendre ce format public. Par ailleurs, MARTE intègre un modèle de temps logique[?] permettant la description de contraintes liées à différentes vues d'un modèle comme le timing, la sécurité ou encore la consommation et éventuellement une méthode assistée de raffinement entre différents niveaux de représentation de modèles[?]. La modélisation des structures répétitives (RSM) définie dans le profil standard UML MARTE simplifie également la représentation de structures paramétrables complexes.

2.5.1 Description de l'aspect structurel

La figure 2.10 représente une vue d'ensemble du profil MARTE qui est composé de trois paquetages principaux. Le premier définit les concepts fondamentaux utilisés dans le domaine temps réel embarqué. Ces concepts sont spécialisés dans les deux autres paquetages de *modélisation* et d'*analyse* des systèmes temps réel embarqués. Le deuxième paquetage concerne la conception dirigée par les modèles. Il fournit des modèles de haut niveau permettant de spécifier les caractéristiques temporelles et embarquées d'applications. Le troisième paquetage fournit une base générique pour l'analyse de performance.

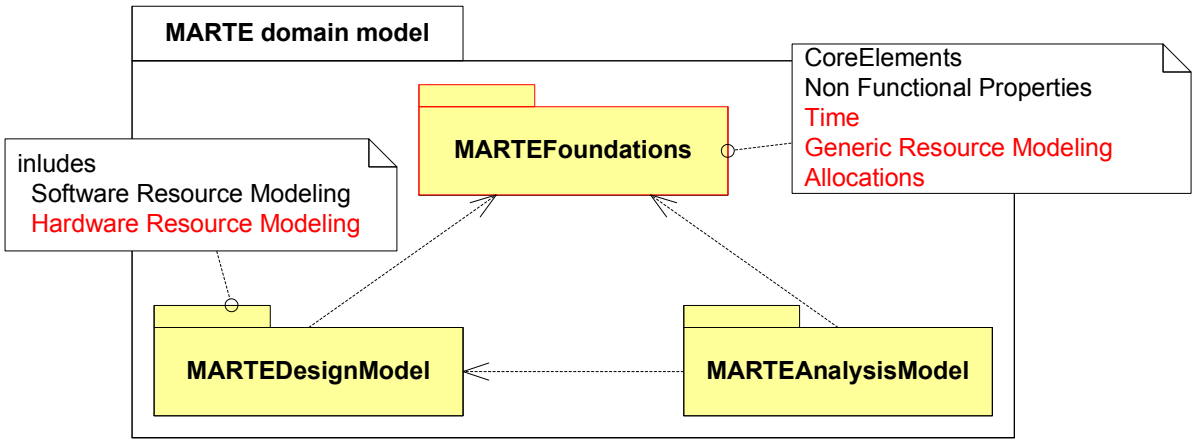


Figure 2.10 – Le profil Marte

Le concept central concernant les ressources est introduit dans le paquetage GenericResource Modeling (GRM) de MARTE. Une ressource représente une entité physique ou logique persistante qui offre un ou plusieurs services. Une ressource est un classeur au sens UML doté d'un comportement. Un classeur décrit un ensemble d'objets. Un objet est une entité spécifique possédant un état et des relations avec d'autres objets. C'est pourquoi un classeur décrit un ensemble d'instances qui ont des propriétés communes. Un classeur est une méta-classe abstraite. On ne peut l'instancier directement, seules ses spécialisations sont instanciables. De plus, un ResourceService est le comportement que l'on peut associer à une ressource. La figure 2.11 donne un aperçu de

la vue de domaine de ces différents concepts. Deux stéréotypes sont définis, Resource et GrService, pour représenter les deux concepts indiqués.

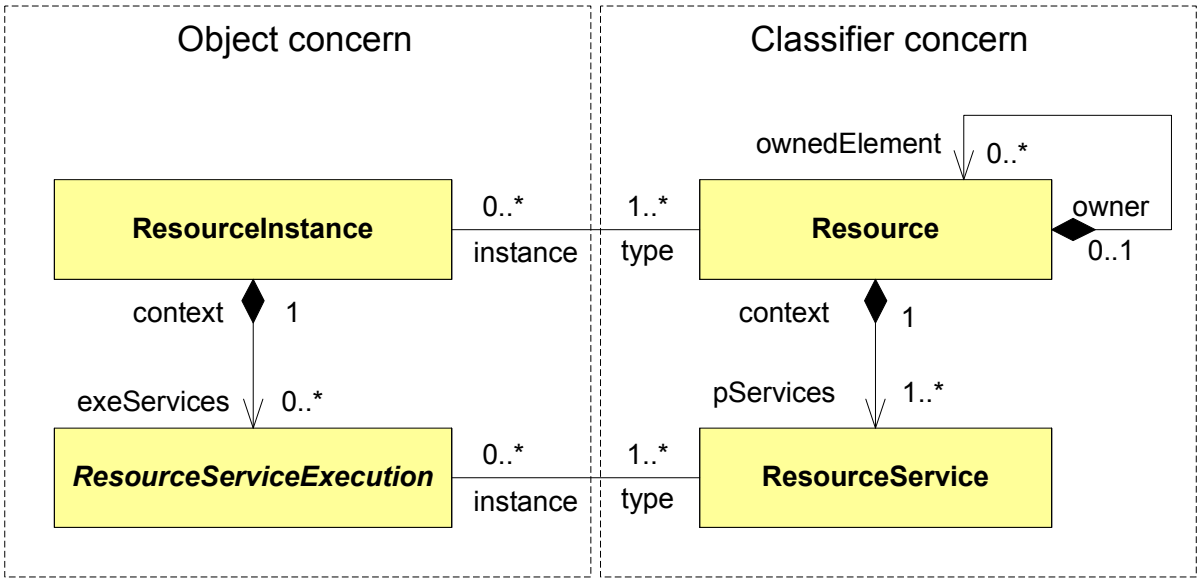


Figure 2.11 – Vue de domaine du paquetage GRM

Plusieurs types de Ressources prédéfinies sont proposées dans MARTE comme les Computing Resource, StorageResource, CommunicationResource et TimingResource. Parmi celles-ci, deux types particuliers de ressources de communication sont définis : les CommunicationMedia et Communication EndPoint. Le CommunicationEndPoint est le point de connexion sur lequel est raccordé un média de communication. Les services associés à ce point de connexion sont des envois/réceptions de données.

Pour la spécification d'une structure, MARTE enrichit les concepts définis dans les structures composites UML. Par exemple, la méta-classe UML Port a été étendue en deux stéréotypes : FlowPort et ClientServerPort. Les *Flow Ports* sont des points d'interaction d'un modèle aux travers desquels l'information passe. La direction du flot est spécifiée par un méta-attribut de direction de flot. Les valeurs possibles de cet attribut sont in, out ou inout (entrée, sortie ou bi-directionnel). Les *Client Server Ports* ont été introduits pour faciliter l'utilisation des ports UML. Ils contiennent un méta-attribut clientSeverKind qui permet de préciser si le port en question fournit et/ou exige certaines opérations. Ces deux types de ports peuvent être utiles pour représenter les échanges de données au niveau RTL (FlowPort) et niveau TLM (ClientServerPort).

En ce qui concerne l'allocation, deux types sont disponibles en MARTE. Le premier type d'allocation lie une description fonctionnelle d'une application à des ressources matérielles disponibles. Cette allocation considère à la fois la distribution spatiale d'un application sur diffé-

rents composants ainsi que ses aspects d'ordonnancement temporel. Le second type d'allocation est de type *raffinement*. Il permet de naviguer entre les différents niveaux de description d'un modèle unique.

Le paquetage Detailed Resource Modeling (DRM) spécialise les concepts précédemment énoncés dans ses deux sous-paquetages : SoftwareResourceModeling (SRM) et HardwareResourceModeling (HRM). Nous ne traitons que de structure dans la suite du manuscrit, c'est pourquoi nous ne détaillons pas ici le paquetage relatif au logiciel. Le paquetage relatif au matériel spécialise la Resource en HwResource. Une telle ressource fournit au moins un ResourceService et peut nécessiter des services issus d'autres ressources. Une HwResource est hiérarchique et peut donc contenir d'autres HwResources. C'est donc la brique de base représentant le composant en MARTE et elle peut être encore spécialisée en différents types tels que les processeurs, mémoires et autres.

2.5.2 Le modèle de temps et CCSL

L'*Allocation* et les *Resource*s se réfèrent au modèle de temps défini dans le paquetage Time de MARTE. Une présentation simplifiée de ce modèle et de ses usages est disponible en français[?].

MARTE introduit avec ce paquetage deux modèles distincts appelés temps chronométrique et temps logique. Le premier remplace le modèle de temps du profil SPT[?] et les valeurs de temps sont exprimées en unités de temps classiques (seconde, minute). Le second peut «compter» le temps en "tics", cycles, ou toute autre unité. En fait, tout événement peut définir une horloge logique qui tique à chaque occurrence de l'événement. Ainsi, le temps logique se concentre sur l'ordre des instants, et non pas sur la durée physique entre instants. Une autre caractéristique remarquable du modèle de temps de MARTE est la possibilité de pouvoir exprimer un temps multi-horloges, nécessaire pour traiter des systèmes embarqués distribués.

En fait, ce modèle de temps est un ensemble de bases de temps, une base de temps étant un ensemble ordonné d'instants. Les instants des différentes bases peuvent être liés par les relations (coïncidence ou précédence), si bien que les bases de temps ne sont plus indépendantes. Il en résulte que les instants sont partiellement ordonnés. Cette organisation partielle des instants caractérise la structure temporelle de l'application. Ce modèle de temps permet alors de vérifier le bon fonctionnement logique d'une l'application.

Une horloge est l'élément de modèle qui donne accès aux instants d'une base de temps. Une ClockConstraint (ou contrainte d'horloges) impose des dépendances entre des instants de bases de temps différentes. Des structures temporelles complexes et les propriétés qui les lient peuvent être spécifiées par une utilisation combinée d'horloges et de contraintes d'horloge.

MARTE introduit également la notion de TimedElement. Un TimedElement associe au moins une horloge avec un élément de modèle. Cette association enrichit la sémantique de l'élément avec les aspects temporels. Ainsi, une TimedValueSpecification renvoie nécessairement à des horloges. Un TimedEvent est un événement dont les occurrences sont explicitement liés à une horloge. Un

TimedProcessing représente une activité qui a un début, une fin et une durée explicitement liés à des horloges. Le stéréotype TimedProcessing peut être appliquée à l'Action, au Behavior et même au Message d'UML.

CCSL[?] (Clock Constraint Specification Language) est le langage formel de spécification de contraintes d'horloges.

Une horloge est définie comme un ensemble totalement ordonné d'instant. Une spécification CCSL est un ensemble d'horloges plus un ensemble de relations entre instants. Trois relations binaires fondamentales entre instants sont définies :

- La *précédence stricte* ($\prec$) : Pour chaque instant i et j, $i \prec j$ signifie que les seules évolutions acceptables sont celles où i arrive strictement avant j
- La *coïncidence* ($\equiv$) : $i \equiv j$ imposes que les instants i et j se produisent en coïncidence. Si l'un se produit alors l'autre aussi, sinon ni l'un ni l'autre ne se produisent.
- L'*exclusion* ($\#$) : $i \# j$ interdit que deux instants soient coïncidents , ils ne peuvent arriver en même temps.

Une quatrième relation entre instants, *précédence non stricte* ($\preccurlyeq$), est dérivée des précédentes : $\preccurlyeq \triangleq \prec \cup \equiv$. Des relations entre horloges sont alors construites à partir des relations élémentaires entre instants. Nous présentons ici seulement une partie de ces relations mais le lecteur peut se référer à un rapport de recherche [?] pour une description plus complète. Pour une horloge discrète c, on notera $c[k]$ son k^e instant. Afin de différencier les opérateurs entre instants et ceux entre horloges, nous noterons ces derniers dans des boîtes.

- La *coïncidence*, notée $\boxed{=}$ est une relation synchrone forte qui impose la coïncidence des instants par paires :

$$a \boxed{=} b \text{ ssi } \forall k \in \mathbb{N}^\star, a[k] \equiv b[k]$$

- L'*exclusion*, notée $\boxed{\#}$ impose que deux horloges n'aient aucun instant en coïncidence :

$$a \boxed{\#} b \text{ ssi } \forall j, k \in \mathbb{N}^\star, a[j] \# b[k]$$

- La *précédence stricte*, notée $\boxed{\prec}$ est une contrainte asynchrone qui impose que le k^e instant de l'horloge à gauche de l'opérateur précède toujours le k^e instant de l'horloge de droite :

$$a \boxed{\prec} b \text{ ssi } \forall k \in \mathbb{N}^\star, a[k] \prec b[k]$$

- La *précédence faible*, notée $\boxed{\preccurlyeq}$ est une version affaiblie de la *précédence stricte*. Le k^e instant de l'horloge de gauche précède toujours ou bien coïncide avec le k^e instant de

l'horloge de droite :

$$a \boxed{\preccurlyeq} b \text{ ssi } \forall k \in \mathbb{N}^\star, a[k] \preccurlyeq b[k]$$

– L'*alternance stricte*, notée $\boxed{\sim}$ est dérivée des relations de précédence stricte :

$$a \boxed{\sim} b \text{ ssi } \forall k \in \mathbb{N}^\star, a[k] \prec b[k] \prec a[k+1]$$

– L'*alternance faible*, notée $\boxed{\simeq}$ est dérivée des relations de précédence faible :

$$a \boxed{\simeq} b \text{ ssi } \forall k \in \mathbb{N}^\star, a[k] \preccurlyeq b[k] \preccurlyeq a[k+1]$$

CCSL définit aussi des expressions d'horloges (*clock expressions*). Une expression d'horloges définit une nouvelle horloge à partir d'autres horloges. Voici un aperçu de celles utilisées dans cette thèse :

– L'*union*, noté $+$, crée une nouvelle horloge qui tique chaque fois qu'un de ces deux opérandes tique. Pour faciliter la lecture, nous considérons par la suite que

$$\bigcup_{i=1..M} Sig_i = (Sig_1 + Sig_2 + ... + Sig_M) \tag{2.1}$$

– Le *minus*, noté $-$, crée une nouvelle horloge qui ne tique que lorsque son opérande de gauche tique sans que son opérande de droite ne tique.
– L'*intersection*, notée $*$, crée une nouvelle horloge qui tique chaque fois que ses deux opérandes tiquent en coïncidence.

Ce sous-ensemble de relations et expressions entre horloges est suffisant pour décrire, dans le cadre de cette thèse, différents types de protocoles et propriétés associées à un composant.

2.5.3 Simulation de spécification CCSL

Une spécification CCSL est une conjonction de contraintes d'horloges logiques. Elle limite les évolutions du système. Afin de pouvoir simuler une des évolutions autorisées par la spécification, celle-ci est traduite en une formule propositionnelle dépendante de la configuration du système à l'instant (logique) courant. Pour cela, chaque horloge de la spécification est associée à une variable propositionnelle. Chaque relation ou expression est convertie en clauses portant sur ces variables en prenant en compte la configuration courante du système. La conjonction de toutes ces clauses permet ainsi d'obtenir la formule propositionnelle de l'instant logique. Toute valuation des variables propositionnelles qui satisfait la formule de l'instant, excepté la valuation qui affecte faux à toutes les variables, est une solution acceptable vis-à-vis de la spécification

CCSL. Le choix d'une des solutions acceptables détermine la configuration du système à l'instant logique suivant. L'interprétation d'une solution est simple : une horloge logique *tique* si et seulement si la variable associée est à vrai dans la solution choisie.

Remarque Si la seule solution qui satisfait la formule de l'instant est la solution dans laquelle toutes les variables sont à faux, alors aucune évolution n'est possible (*deadlock*). On ne peut plus satisfaire aux contraintes imposées.

L'outils TimeSquare tstool développé au sein de l'équipe INRIA AOSTE permet de générer des traces d'exécutions conforment à une spécification CCSL. Il produit un fichier contenant les relations liants les instants d'activations des horloges conformément aux équations de la spécification qu'on lui fournit en entrée. Il est alors possible de visualiser ces traces sous la forme d'un chronogramme directement dans l'outil ou à l'aide d'un afficheur de "waveform" classique. La figure 2.12 donne un aperçu d'une telle visualisation pour une spécification contenant une simple précédence faible entre deux horloges. L'avance du temps logique se fait de la gauche vers la droite et chaque "créneau" y représente l'activation de l'horloge qui étiquette la ligne. Les flèches en pointillés représentent les relations de précédence entre instants logiques des diverses horloges.

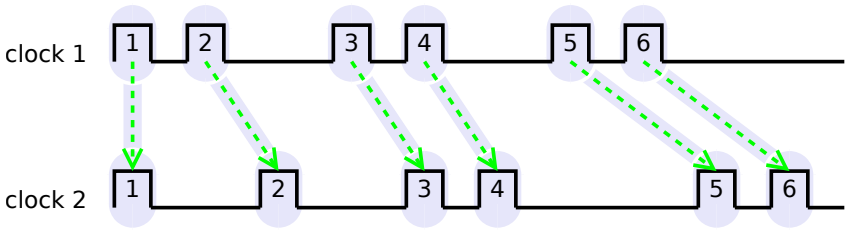


Figure 2.12 – Relation de précédence faible

2.6 Bilan et objectifs

Dans la section 2.2, nous avons analysé les possibilités de modélisation de la librairie SystemC. Cette librairie comporte tous les ingrédients nécessaires à la description architecturale d'une plate-forme à différents niveaux. Elle permet de plus, de réaliser des simulations de ces différents niveaux parfois même entremêlés afin de vérifier par l'observation le comportement global d'un design. Cependant, la structure d'un design est pour cela encodée en C^{++} rendant complexes son interprétation et son utilisation par des outils d'analyse. De plus, le simulateur intégré à la librairie SystemC peut poser des problèmes de modélisation fidèle d'évolutions concurrentes. Tout ceci nous a amené à considérer d'autres manières de représenter ces informations, dans d'autres environnements permettant de s'affranchir des problèmes mentionnés.

Dans la section 2.4 nous avons présenté le format IP-Xact. Il permet la description de systèmes d'un point de vue architectural et offre des possibilités d'assemblage automatique de plates-formes et cela sous forme de modèle conforme aux spécifications permises dans le standard. Le concept d'abstraction de communication (*Bus Interface*) est une innovation intéressante. Le point faible de ce format reste son manque de mécanismes pour représenter des comportements. Il n'en reste pas moins un bon candidat en tant qu'ADL (Architecture Description Language) de par son format ouvert et sa large diffusion au sein des outils industriels.

La section 2.5 s'est intéressée au profil UML-MARTE. Il bénéficie de la puissance de modélisation d'UML et permet donc une approche dirigée par les modèles (*Model Driven Engineering*). Il définit les concepts nécessaires à la modélisation matérielle permettant de décrire la structure d'une plate-forme. Il intègre de plus une partie comportementale permettant de décrire les évolutions possibles d'un système. Dans une certaine mesure, le parallélisme explicite d'un système peut être identifié par la structure de son design. De plus, ce même mécanisme peut être utilisé pour enrichir un modèle par l'ajout de propriétés non-fonctionnelles à un certain niveau de représentation et ainsi raffiner/préciser le modèle considéré.

Comme déjà énoncé dans l'introduction de cette thèse, nous traiterons dans le chapitre 3, de la réalisation d'un passerelle entre SystemC et IP-Xact. Un modèle ainsi extrait pourra alors être transformé dans un modèle MARTE. Le chapitre 4 présentera des possibilités du profil UML-MARTE et son modèle de temps.

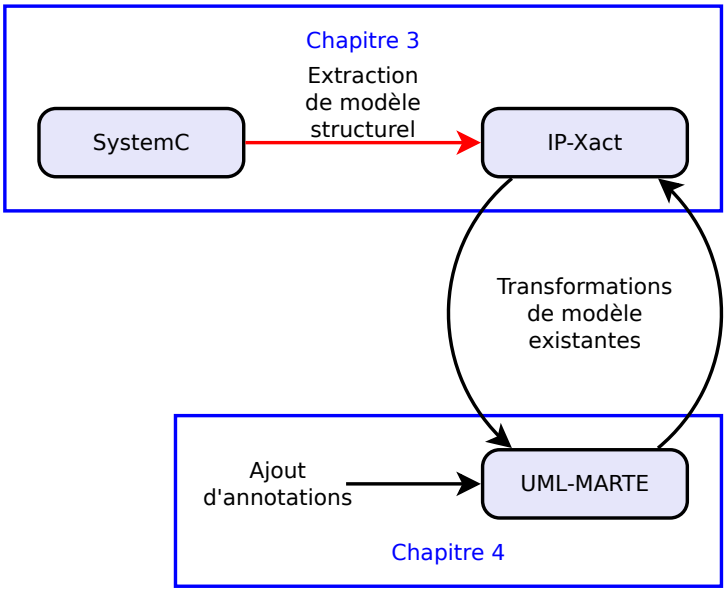


Figure 2.13 – Relations entre formats/modèles

Chapitre 3

Extraction de modèle structurel SystemC et exportation en IP-Xact

Sommaire

3.1 Introduction

Il existe une ambiguïté (ou du moins un flou sémantique) sur l'utilisation du mot "modèle" dans la domaine de la conception au niveau ESL (*Electronic System Level*) des systèmes sur puce. Alors qu'aux niveaux inférieurs (électroniques et logiques) les circuits matériels bénéficient de modèles physiques et mathématiques clairs, le niveau d'abstraction dit TLM (*Transaction-level modeling*) est surtout basé sur des programmes et des bibliothèques, ces dernières fournissant des modes de simulation basés sur des événements discrets temporisés. En ce sens les modèles sont des modèles de programmation, disciplines avec lesquelles le programmeur doit composer pour créer un programme dont le comportement soit prévisible. Une alternative à cette approche est le recours à des modèles dont la sémantique formelle est explicitement définie. Les modèles sont alors plus directement analysables par des techniques et méthodes formelles. Devant cet état de fait, il nous a semblé intéressant et utile de considérer comment un programme SystemC (un modèle de calcul décrit en langage de programmation) pouvait être converti en un modèle plus facilement analysable (plus formel). L'approche privilégiée est d'extraire la structure et la topologie des composants et de leur interconnexions. Ultérieurement, ces programmes/modèles peuvent être assemblés au sein d'une plate-forme virtuelle. On peut aussi vouloir compléter cette vision modèle de la structure du programme en y annotant des informations additionnelles dans des domaines extra-fonctionnels. Nous nous concentrons donc dans ce chapitre sur l'extraction d'un modèle structurel d'interfaces et d'interconnexions entre composants, en partant d'un design SystemC RTL ou TLM. Ce modèle est ensuite traduit en une représentation IP-Xact.

Ce chapitre traite de transformation effective de modèles (SystemC vers IP-Xact). Nous y exposons les difficultés rencontrées et les solutions apportées. La section 3.2 présente les outils et modèles connexes considérés lors de la construction d'une telle passerelle. La section 3.3 décrit les options de conception retenues nous conduisant à la réalisation de l'outil SCiPX (section3.4). La section 3.5 évalue cet outil au travers d'exemples de designs et tire un bilan des possibilités de l'outil.

3.1.1 Difficultés et limites des approches statiques et dynamiques

Comme déjà indiqué dans la section 2.2, SystemC est une bibliothèque C++. La description d'un design à l'aide de celui-ci vient avec tous les avantages que peut amener C++ en terme de modularité potentielle ainsi que la large disponibilité d'outils et environnements informatiques de développement (compilateurs, débugger). Cependant, l'extraction d'informations d'un code SystemC pâtit de toute la complexité qu'engendre l'analyse syntaxique d'un programme C++. Extraire les informations d'un design décrit en SystemC nécessite dans l'absolu de réécrire

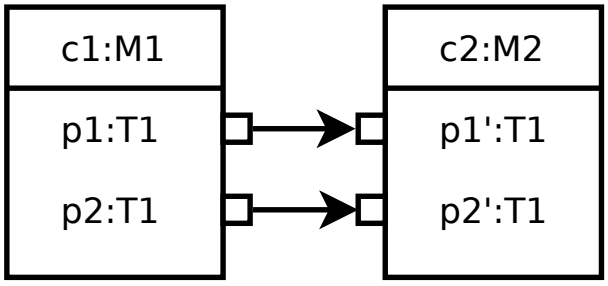


Figure 3.1 – Design simple

un analyseur syntaxique basé sur le langage C++, mais pas seulement. Un design décrit en SystemC peut profiter des facilités de déclarations dynamiques permises en C++. Toutefois, la compilation du programme SystemC/C++ perd certaines informations durant la traduction en langage machine. Lors de son exécution, il passe par une phase d'élaboration qui se charge d'instancier le design avant de commencer une simulation. Le design sera figé dans sa topologie et sa structure d'interconnexion pour les communications dès lors que le programme atteindra la fin de cette phase.

Conformément aux divers travaux déjà éxistant (détaillés en section 3.2.2), nous ferons donc par la suite la distinction entre deux types d'approches pour l'analyse d'un code SystemC. La première est une approche dite *statique*, basée sur l'analyse syntaxique d'un programme avant la phase de compilation. Celle-ci analysera le code SystemC/C++ vierge de toute modification. Les informations extraites seront proches de la vision qu'a le designer de son modèle. La deuxième approche est dite *dynamique* et tente d'extraire les informations post-compilation. Elle exécute la phase d'élaboration afin de récupérer la hiérarchie des composants instanciés. Cette approche serait parfaite si l'on ne souhaite avoir qu'une vue topologique d'un design. Cependant, la perte d'information déjà mentionnée due à la compilation du programme SystemC rend impossible la distinction entre différents d'objets de même type.

Une compilation en mode debug permettrait de conserver la table de tous les symboles du programme compilé. Mais l'interprétation de celle-ci demanderait à nouveau de réécrire des morceaux du compilateur afin de pouvoir faire remonter ces informations, alors que celles-ci sont disponibles par une analyse statique du code. Nous allons illustrer sur un exemple simple la nature des difficultés rencontrées pour chaque approche.

3.1.2 Illustration de ces difficultés

L'exemple simpliste de la figure 3.1 illustre en partie ces difficultés. c1, c2 sont deux composants, respectivement des instances de M1 et M2. p1, p2, p1', p2' sont des ports de même type T1. p1 est connecté à p1' au travers du canal channel1 ; p2 est relié à p2' au travers du canal channel2.

Le code SystemC correspondant peut être celui décrit dans le listing 3.1.

Listing 3.1 – Design simple en SystemC

```
1  #include <systemc.h>
2  #include <iostream>
3
4  SC_MODULE(M1){
5     sc_in<T1> p1_m1;
6     sc_in<T1> p2_m1;
7     SC_CTOR(M1){}
8  };
9
10 SC_MODULE(M2){
11    sc_out<T1> p1_m2;
12    sc_out<T1> p2_m2;
13    SC_CTOR(M2){}
14 };
15
16 ////////////////////////////////////////
17 // main
18 ////////////////////////////////////////
19
20 int sc_main(int argc, char **argv){
21    M1 c1("c1");
22    M2 c2("c2");
23    sc_signal<T1> s1;
24    sc_signal<T1> s2;
25    c1.p1_m1(s1);
26    c1.p2_m1(s2);
27    c2.p1_m2(s1);
28    c2.p2_m2(s2);
29    sc_start();
30    return 0;}
```

Une analyse syntaxique du code nous permet de retrouver la topologie des composants M1 et M2. Effectivement, tous leurs ports étant déclarés de manière statique, leur identification reste relativement simple. Leur interconnexion ne peut être retrouvée sans interprétation du contenu du sc_main (même si dans le cas présent il ne s'agit que d'une simple association signal/port). De la même manière, le nom des composants (*i.e.,*"c1" et "c2" passés en paramètre des constructeurs des instances c1 et c2) ne peut être retrouvé aisément. Dans le meilleur des cas, toutes ces informations sont statiquement définies avant l'exécution de la phase d'élaboration du programme SystemC. Mais rien n'empêche un programmeur de vouloir passer certains

paramètres à l'exécution. Si tel est le cas, une analyse syntaxique ne peut récupérer que des informations incomplètes sur le design final. Par une analyse syntaxique, on peut cependant récupérer facilement les types et noms des attributs de toutes les classes et structures à partir du moment où ces informations sont définies de manière statique. Si l'on remplace la déclaration du composant M1 par le code décrit dans le listing 3.2 ainsi que les "." par des "->" aux lignes 25 et 28 du listing 3.1, le design reste inchangé mais cela rend l'analyse statique du code un peu plus complexe. Il faut alors interpréter le contenu du constructeur du module M1 afin de pouvoir déterminer que les deux pointeurs de ports sont réellement utilisés.

Listing 3.2 – Déclaration alternative du composant M1

```
1 SC_MODULE(M1){
2    sc_in<T1> * p1_m1;
3    sc_in<T1> * p2_m1;
4    SC_CTOR(M1){
5       p1_m1= new sc_in<T1>;
6       p2_m1= new sc_in<T1>;
7    }
8 };
```

Si à cela on ajoute dans le sc_main un mécanisme permettant à l'utilisateur de pouvoir choisir à l'exécution les connexions entre les différents ports, l'analyse syntaxique (sans interprétation abstraite) de code devient alors totalement impuissante en ce qui concerne l'extraction de la hiérarchie du design.

Une analyse dynamique (à l'exécution) est capable de récupérer les instances de composants et leurs instances de ports, ainsi que le lien entre les ports grâce aux API proposées par la bibliothèque SystemC. Un parcours des objets créés durant la phase d'élaboration permet de récupérer la hiérarchie complète du design sous la forme d'un arbre. Les deux déclarations précédentes sont vues quasiment de la même façon, à une différence près : les ports déclarés comme dans le listing 3.2 sont des objets référencés par le composant M1 et non des attributs contenus dans le corps de l'objet de type M1. Malheureusement, le programme SystemC étant le produit d'une compilation C++, certaines informations deviennent alors potentiellement illisibles voire inexploitables. Certaines peuvent quand même être récupérées comme par exemple, les types instanciés. Le nom des composants est récupérable car la librairie SystemC a prévu un champ spécial dans la classe sc_module qui le contient. Toutefois, les informations relatives aux noms des attributs ne sont plus accessibles simplement car aucun champ n'est prévu à cet effet. Ainsi, il n'est plus possible de différencier deux ports de même type. Quelle que soit la manière de déclarer ces ports, après la compilation ils sont identifiés par leur adresse mémoire. Et de ce fait, après la phase d'élaboration, on ne peut pas faire la différence entre p1 (resp. p1') et p2 (resp. p2') et donc savoir si p1 (resp. p2) est vraiment relié à p1' (resp. p2') ou si p1 (resp. p2)

est relié à p2' (resp. p1').

3.2 Les outils et modèles existants

Afin de permettre une approche orientée modèle, nous avons dû considérer différents points. Tout d'abord, rechercher la manière la plus adéquate pour décrire de manière générique toutes les sortes de structures de design que SystemC permet de définir. Avant de présenter nos propres travaux, nous analysons des contributions visant des objectifs proches. Certains sont parfois antérieures et source d'inspiration (ou de critique), d'autres ont été conduites en parallèle et sont contemporaines de nos développements.

3.2.1 Les méta-modèles SystemC existants

On trouve dans la littérature différents méta-modèles pour SystemC. Ils sont plus ou moins détaillés et souvent dédiés à certains types d'applications. Nous présenterons dans cette section ceux que nous avons analysés par ordre d'apparition.

3.2.1.1 Méta-modèle DaRT (2004)

L'idée d'un méta-modèle pour SystemC apparaît dès 2004 lors du "*Forum on specification and Design Langages*"[?] dans des communications[?,?]. L'objectif des auteurs était de produire un cadre de co-conception, supportant des modèles décrits à différents niveaux de détails. Le contexte d'utilisation de ce méta-modèle est le traitement du signal intensif (*Intensive Signal Processing* ou ISP) et le flot considéré est purement descendant, partant de spécifications de haut niveau en UML avec le profil ISP-UML (où est défini aussi l'*ISP-UML metamodel*), traduites ensuite en SystemC par l'intermédiaire de règles de transformations entre leurs deux méta-modèles. Les auteurs se concentrent sur l'aspect structurel d'un design en délégant le comportement des composants à des éléments appelés CodeMapping. Malheureusement, certaines informations nécessaires à la réalisation d'un flot remontant restent manquantes. Le méta-modèle SystemC est ici utilisé comme méta-modèle cible pour leur transformation et de ce fait, le sous-ensemble des modèles SystemC générés n'est pas représentatif de la pléiade de styles de codage autorisés par SystemC. Compte tenu de la date de parution de ces travaux, aucun mécanisme de prise en compte n'a été prévu pour supporter les constructions de type TLM. Les auteurs caractérisent d'ailleurs leur méta-modèle ainsi :

> *The metamodel presented here is not a definition of the SystemC library. It is rather a metamodel oriented towards SystemC code generation for the particular case of Intensive Signal Processing mapped on a SoC.*

3.2.1.2 Méta-modèle Fudan (2004-06)

A la même période, des chercheurs de l'Université Fudan (Shanghai, Chine) proposaient une alternative[?] plus proche des objets disponibles dans le librairie SystemC. Le méta-modèle intègre cette fois les éléments nécessaires à la modélisation du comportement par l'intermédiaire des sc_process. Cette approche considère aussi un flot descendant avec les mêmes manques que le méta-modèle précédemment présenté. Cependant, ce méta-modèle est le premier à essayer de se rapprocher des informations que peut contenir un code SystemC au sens C^{++} du terme. On y retrouve les types internes à SystemC tels que les sc_port, sc_prim_channel, sc_module, sc_process, sc_interface, etc... Toutefois, il ne considère que les sc_port, éliminant la possibilité d'avoir des sc_export dans un design.

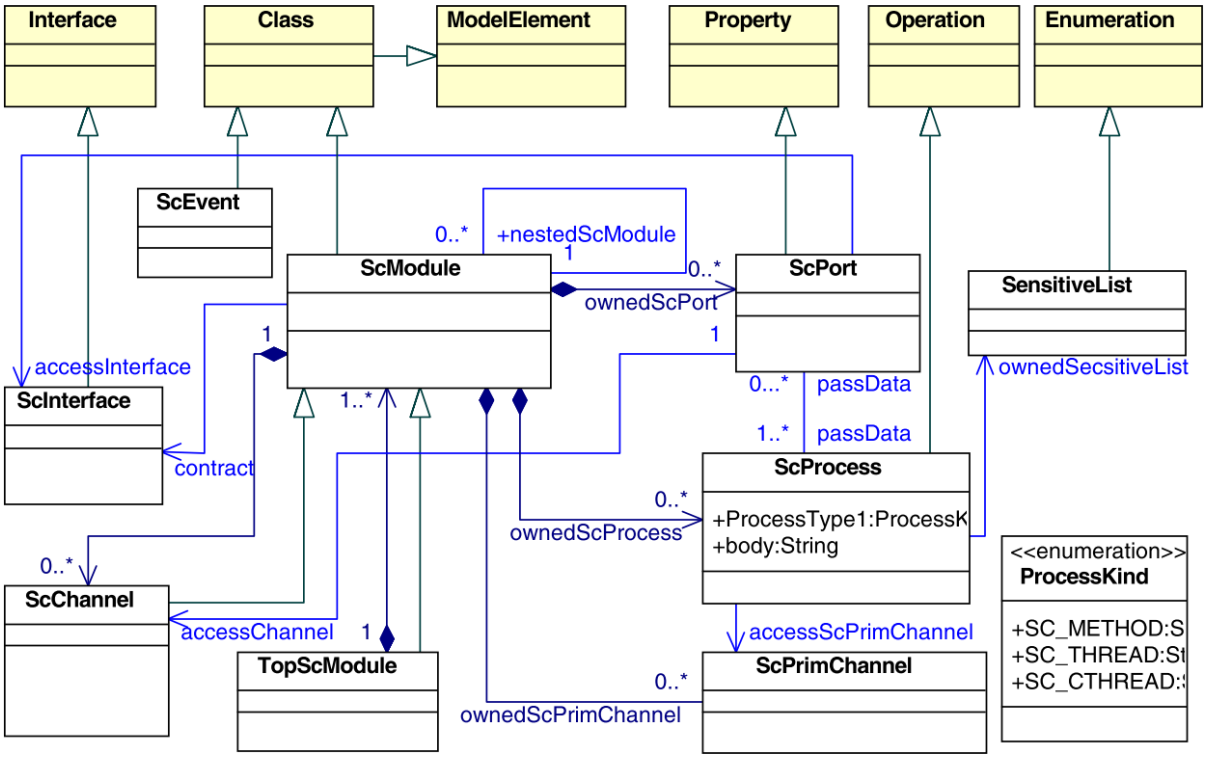


Figure 3.2 – Méta-modèle Fudan 2004

De cette modélisation nous retenons quelques idées. Le niveau le plus haut de la hiérarchie (TopScModule) est une spécialisation du ScModule et peut contenir des *ScModules*, permettant ainsi une modélisation récursive. Le TopScModule possède les mêmes caractéristiques qu'un composant et peut être plus facilement réutilisé en tant que composant dans un nouveau design. L'élément racine d'un modèle ne peut être qu'un *composant*. Ce méta-modèle a été abandonné pour une version plus élaborée[?] qui essaie de séloigner de la syntaxe concrète pour privilégier les

concepts. La figure 3.3 en donne un aperçu. On constate l'apparition du concept Port pouvant alors regrouper sc_port et sc_export ainsi que la possibilité de connecter directement un *Port* à un autre *Port*. De plus, l'élément Channel n'hérite pas du composant Module et est spécialisé en deux types de canaux : Primitive Channel et Hierarchical Channel (anciennement ScPrimChannel et ScChannel). Il semblerait d'ailleurs que le lien d'héritage entre un Hierarchical Channel et un Module soit manquant.

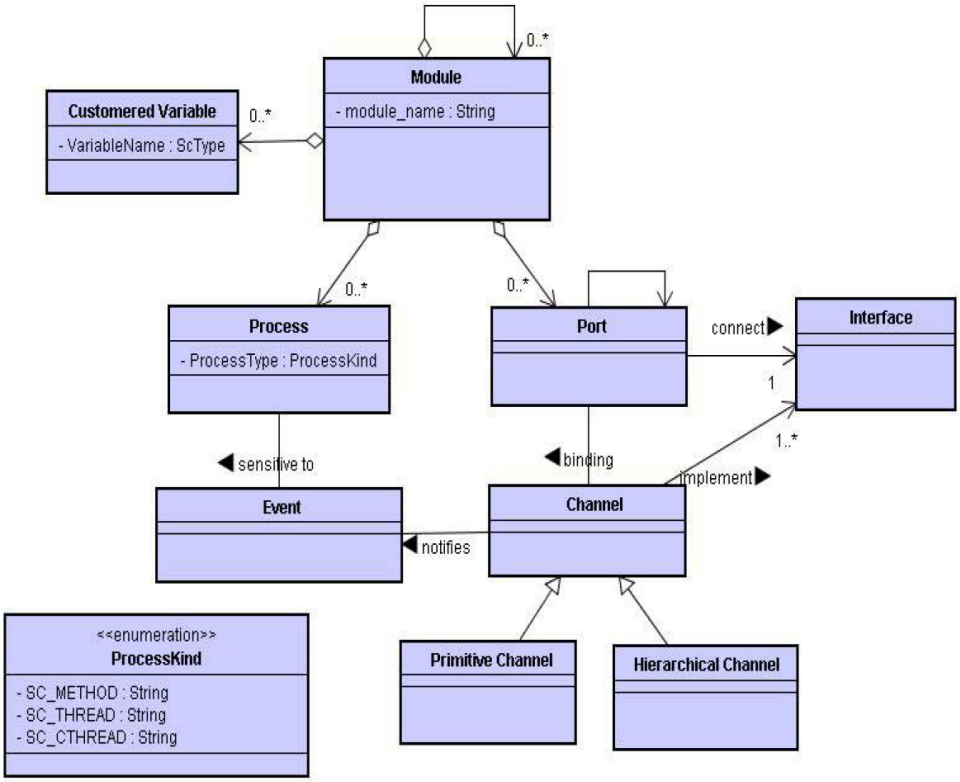


Figure 3.3 – Méta-modèle Fudan 2006

3.2.1.3 Métat-modèle UniMi (2008-09)

L'Université de Milan est très active dans l'étude de l'utilisation de SystemC. Des chercheurs du département technologie de l'information dirigés par Elvinia Riccobene proposent le méta-modèle de la figure 3.4 [?]. C'est une amélioration d'une version antérieure[?], très proche du méta-modèle de Fudan. Comme celui de Fudan, les sc_port et sc_export sont abstraits dans le même concept de Port et un lien est prévu pour permettre la connexion directe entre un *Port* et un autre *Port*. Ce type de lien se rencontre notamment dans les design de type TLM. De plus, l'élément le plus haut dans la hiérarchie (*toplevel* ou sc_main) n'apparaît pas. Les éléments les plus hauts sont de type Module ou Channel.

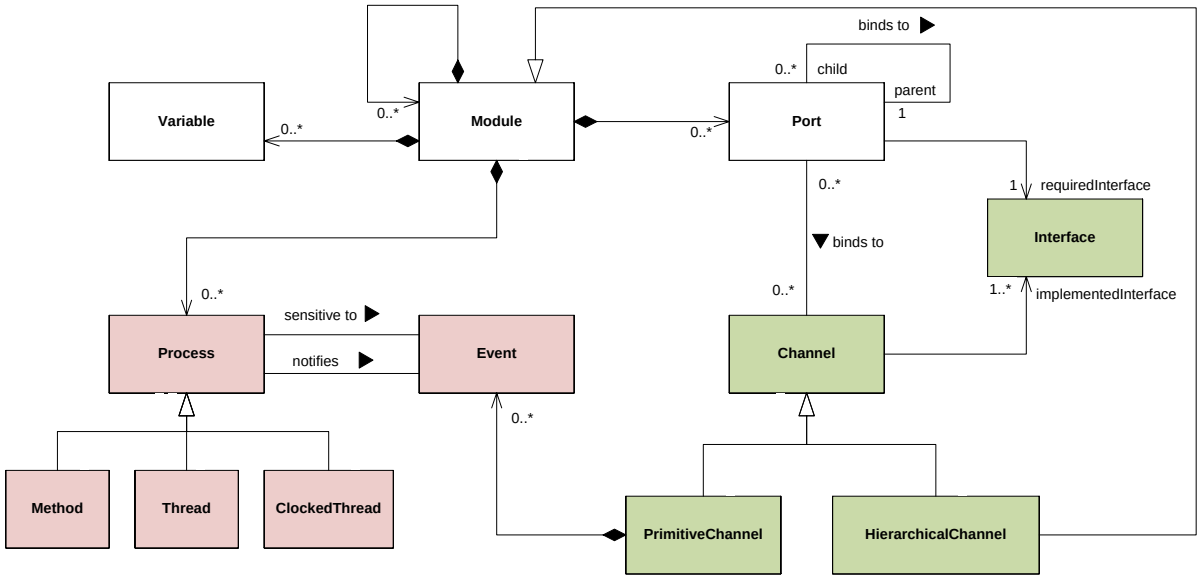


Figure 3.4 – Méta-modèle UniMi

L'amélioration apportée dans ce méta-modèle concerne les relations entre Process, Event et Primitive Channel. Dans la version antérieure, une erreur avait été faite. Ce sont bien les process qui génèrent des événements et non les canaux. Ces derniers se contentent de les transporter.

3.2.1.4 Méta-modèle Delft (2010-11)

Des travaux conduits à l'Université de Delft ont produit un méta-modèle de SystemC proche de celui de Fudan. Le but est similaire au nôtre concernant la possibilité de contenir les informations structurelles d'un design. On trouve les détails de celui-ci tout d'abord dans un rapport de Master[?] qui donna lieu par la suite à un article[?]. Notons que ce travail a été effectué chronologiquement en parallèle avec nos études.

Le méta-modèle utilisé est décrit dans la figure 3.5. Il reprend, trait pour trait, la structure interne des données en SystemC. On peut le remarquer d'ailleurs par le lien de *généralisation* utilisé entre les interfaces et leurs implémentations qui est plus conforme à la vision C++, là où une *réalisation* aurait été plus appropriée en UML. Les arguments développés sont les mêmes que les nôtres et ce méta-modèle est utilisé dans un logiciel nommé "SHaBE" (pour "S*ystem*C *Hierarchy and Behavior Extractor*"). Il en découle un analyseur plus performant que les autres mais limité à la simple extraction et de bas niveau (RTL). L'auteur le mentionne[?] d'ailleurs dans les travaux à venir :

> *Future works : A possible application which uses SHaBE as a front-end, would be a tool which converts a* S*ystem*C *model comprising a parameterized dynamic*

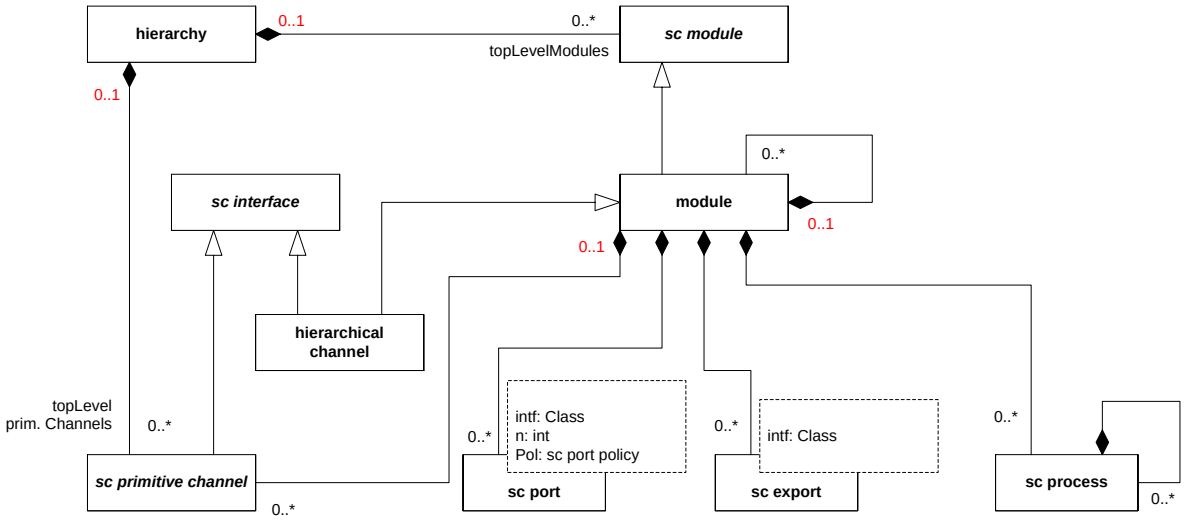


Figure 3.5 – Delft MM

hierarchy into a SystemC *model with a fully expanded hierarchy. The resulting* SystemC *model can then be further processed using existing tools. Also, it would be interesting to investigate if the approach used in SHaBE can also be applied to* SystemC *models which use the* SystemC *TLM and/or the* SystemC *Analog/Mixed-signal (AMS) extensions.*

3.2.1.5 Bilan sur les méta-modèles SystemC

Les méta-modèles décrits précédemment sont pour la plupart dédiés à des applications bien précises. Néanmoins, les concepts mis en avant dans chacun d'eux se recoupent. Nous pouvons noter deux grandes classes de méta-modèles. Certains tendent à se rapprocher de l'implémentation interne de SystemC tels que la première version proposée par les chercheurs de Fudan ou par le travail de l'Université de Delft. D'autres tentent de s'en abstraire en regroupant les concepts qui se recoupent (*i.e.*,sc_port, sc_export). C'est en prenant soin d'étudier tout ces méta-modèles que nous présenterons par la suite, le méta-modèle le plus simple que nous ayons trouvé adapté à notre besoin (*i.e.*,nous permettant de représenter les informations structurelles d'un design SystemC et de définir une transformation vers le méta-modèle d'IP-Xact).

3.2.2 Les outils d'analyse de programmes SystemC

Nous avons recensé deux catégories différentes d'outils permettant l'analyse de programmes SystemC. Ceux de type statique qui adoptent une approche classique d'analyse syntaxique, se heurtant d'une part à toute l'expressivité permise par le C^{++} et d'autre part à l'absence

des paramètres qui pourraient être passés au programme SystemC durant son exécution. Et ceux de type dynamique qui essayent de récupérer les informations durant l'exécution d'un programme SystemC, se heurtant eux à une perte d'information durant la phase de compilation du programme. Un recensement des différents outils existants est disponible dans le papier récent de Matthieu Moy[?]. Il décompose ces outils en différentes catégories, Ceux qui utilisent une grammaire dédiée, ceux qui utilisent un frontend C++ existant, et ceux utilisant une approches dites "hybride" (statique/dynamique). Dans la thèse de Harry Broeders,[?] on trouve un recensement similaire. Cette fois-ci, les catégories distinguées sont statique, dynamique ou hybride. Nous distinguerons par la suite seulement deux types d'approches, statique et dynamique. Nous considèrerons qu'une approche "hybride" appartient à la catégorie statique (resp. dynamique) si elle est enrichie par des informations dynamiques obtenues par exécution symbolique (resp. enrichie par des informations statiques). Nous exposerons dans la suite de cette section les différentes approches recensées et discuterons de leurs atouts et faiblesses.

3.2.2.1 Approche statique

Les outils de cette section utilisent une approche statique pour l'analyse de code SystemC. Ils se basent sur un analyseur de C++ enrichi par la reconnaissance des mots clef spécifiques aux classes SystemC. Cette analyse statique ne suffit plus dès que l'architecture du système dépend de certaines informations passées au moment de l'exécution du programme. Les partisans de cette approche affirment que les modèles pour lesquels la hiérarchie de modules ne peut pas être récupérée ne sont pas très utilisés dans la pratique. Nous pensons que ce n'est justement pas le cas et que se limiter à ce style de codage restreint énormément le nombre de modèles disponibles.

KaSCPar *Karlsruhe* S*ystem*C *Parser* (KaSCPar)[?] a été développé au *ForschungsZentrum Informatik* (FZI). Cet analyseur se compose de deux éléments. SC2AST est un analyseur de SystemC, qui récupère les informations du code SystemC et génère l'arbre de syntaxe abstraite dans un fichier XML. SC2XML utilise cet arbre pour interpréter la phase d'élaboration du code SystemC. Les informations hiérarchiques récupérées de cette façon sont sauvegardées dans un fichier XML. Cet outil, SC2AST, est distribué gratuitement, mais sous forme d'un programme Java compilé. Dans la documentation SC2XML, les auteurs reconnaissent plusieurs limitations. Toutes les formes de code SystemC ne sont pas reconnues par l'outil et certains opérateurs C++, tels que l'opérateur conditionnel (? :) ne sont pas reconnus non plus. Il est, tout de même, utilisé dans divers projets et outils[?,?].

ParSyC *Parser pour* S*ystem*C (ParSyC)[?] est un front-end SystemC développé à l'Université de Brême. Il est construit à partir du *Purdue Compiler Construction Tool Set* (PCCTS)[?]. Par-

SyC prend en entrée un modèle SystemC et produit un arbre de syntaxe abstraite contenant les informations comportementales du modèle. ParSyC fait partie intégrante de SyCE[?], un environnement de développement pour la conception de systèmes développé lui aussi à l'Université de Brême. On retrouve ParSyC dans plusieurs parties de SyCE. Notamment dans l'outil de vérification formelle CheckSyC[?]. L'environnement SyCE contient également l'outil ViSyC[?] qui permet de créer une vue structurelle d'un design écrit en SystemC, mais celui-ci n'utilise pas ParSyC directement. Il récupère les informations du design après la fin de la phase d'élaboration. Ce n'est qu'après cela que l'outil[?] utilise ParSyC afin de pouvoir visualiser la structure et le comportement d'un design écrit en SystemC. Malheureusement, le code source de ParSyC n'est pas accessible au public étant donné qu'il fait partie intégrante d'une suite commerciale d'outils.

SCOOT SCOOT[?] est un outil d'analyse statique de systèmes décrits en SystemC. Il extrait des informations qui peuvent être transmises à des outils de vérification. Il a été développé par l'ETH Zurich et l'Université d'Oxford. SCOOT utilise un front-end pour traduire le code SystemC en graphe de contrôle de flot. Par la suite, des techniques d'analyse statique de pointeurs[?] sont utilisées pour déterminer la hiérarchie du module, la liste de sensibilité des processus, et les liaisons de ports. Après extraction de cette information, SCOOT re-synthétise un programme C++ qui ne dépend pas de la bibliothèque SystemC et de son simulateur. Il s'affranchit donc du simulateur de SystemC en le remplaçant par un simulateur particulier. Selon les auteurs, le simulateur exécute alors le modèle plus vite que le simulateur original. Là encore, le code source de SCOOT n'est pas accessible au public.

SystemCXML SystemCXML[?] a été initialement développé dans le cadre du projet INRIA Espresso. Il fait également partie du framework CARH[?] développé au centre FERMAT (Virginia Tech). CARH est utilisé pour la validation des modèles SystemC au niveau système. SystemCXML est une sur-couche qui utilise Doxygen, un outil qui génère une documentation formatée en XML. La partie statique de l'information hiérarchique peut être facilement récupérée à partir d'un modèle SystemC. Le fichier XML produit par Doxygen est transformé en un fichier XML qui décrit la structure de chaque module. toutefois, SystemCXML n'est pas capable de déterminer quels modules sont instanciés et comment ils sont connectés. SystemCXML crée une structure de données interne qui peut être accessible via une API pour un traitement ultérieur. L'utilisation de Doxygen élimine la nécessité d'utiliser un analyseur complexe de SystemC (C++). SystemCXML n'est pas capable de récupérer les informations comportementales à partir d'un modèle SystemC comme le reconnaissent ses auteurs.

3.2.2.2 Approche dynamique

Le principal atout de l'approche dynamique est le fait que l'on peut s'affranchir de la complexité de développement d'un analyseur de SystemC/C++. Techniquement, cette approche utilise, de manière cachée, un analyseur généraliste du langage C++. Le compilateur C++ qui est utilisé pour compiler le modèle SystemC contient évidemment ce genre de mécanismes pour pouvoir compiler le code C++. Dans cette approche, le modèle est réellement exécuté. Les mécanismes internes à SystemC permettent de récupérer la hiérarchie des modules instanciés au prix d'une perte d'informations durant la phase de compilation. Le plus grand défi pour les analyseurs de la catégorie dynamique est la récupération du comportement d'un modèle SystemC. L'arrivée de modèles SystemC optimisés pour le temps de simulation rend les choses encore plus délicates. Une API SystemC peut être utilisée pour trouver les processus qui sont utilisés pour implémenter le comportement du module. Il est alors possible d'accéder à certaines propriétés de ces processus, comme par exemple le type des processus : SC_METHOD, SC_THREAD ou SC_CTHREAD. Ces processus contiennent également des références vers le code machine (issu de la compilation) qui implémente le comportement du module, mais ne contiennent aucune référence au code C++ qui a été utilisé pour spécifier ce comportement. Il est alors nécessaire de retrouver ces informations par des méthodes inspirées de l'approche statique si l'on veut extraire un modèle SystemC qui soit humainement lisible.

Quiny Quiny[?] est un front-end SystemC développé dans le cadre du projet européen *Interface and Communication based Design of Embedded Systems* (ICODES). Quiny exécute un code SystemC afin de récupérer la hiérarchie du design et les informations comportementales du modèle. Techniquement, le code SystemC est compilé et lié avec une librairie interne, qui remplace la librairie SystemC. Lorsque le code est exécuté, Quiny construit à la volée des AST (Arbre Syntaxique Abstraite) pour chaque expression qu'il rencontre. Pour cela, Quiny s'appuie sur le fait que les types SystemC peuvent être facilement "détournés" en utilisant des surcharges d'opérateurs. Pour les types primitifs, cette technique devient inapplicable. Les types composés de plusieurs mots clé tels que unsigned int ne peuvent être gérés par cette méthode. Ceci oblige alors le programmeur à utiliser des types spéciaux pour aider l'outil (Q_UINT dans ce cas). Le même problème se pose pour les pointeurs et tableaux, obligeant le programmeur à utiliser les types Array et Pointer.

PINAPA *Is Not A PArser* (PINAPA)[?] est un front-end SystemC open-source faisant partie intégrante de l'outil LusSy[?]. LusSy a pour but de permettre l'analyse des Systèmes sur puce (SoC) décrits au niveau transactionnel (TLM). PINAPA stocke les informations sous la forme d'un AST qui contient la hiérarchie du design SystemC issue de la phase d'élaboration. PINAPA se présente sous la forme d'un patch pour la librairie SystemC (versions 2.1.1 et 2.0.1) et un

patch pour la version 3.4.1 du compilateur GCC. La version modifiée de GCC produit l'AST du code SystemC et la version modifiée de la librairie SystemC est utilisée pour exécuter la phase d'élaboration jusqu'à la fin de la phase d'élaboration et récupérer les informations hiérarchiques. On peut noter les embryons d'un traducteur SystemC vers IP-Xact.

PinaVM PinaVM[?] est le successeur de PINAPA. PinaVM utilise LLVM-GCC[?] pour compiler le code source SystemC en bitcode LLVM (*Low Level Vitual Machine*). Il utilise alors ce bitcode pour exécuter la phase d'élaboration qui révèle la hiérarchie d'un design SystemC. Puis, durant la phase de simulation, PinaVM lie les informations comportementales à la hiérarchie du design. Pour cela, PinaVM implémente une reconnaissance des primitives read, write et wait. Lorsque PinaVM identifie un de ces appels de fonction, il essaie de relier les paramètres d'appel à la hiérarchie du design SystemC. Par exemple, lors d'un appel à la fonction d'écriture sur un port de sortie, le paramètre qui spécifie ce port doit être identifié. La valeur de ce paramètre peut être le résultat d'un calcul annexe. L'idée de PinaVM est d'identifier les bitcodes qui sont utilisés pour calculer ce paramètre et d'ensuite construire une nouvelle fonction LLVM qui contient ces bitcodes et produit la valeur du paramètre. Une fois cette construction faite, cette fonction est exécutée et la valeur du paramètre peut être reliée à l'objet approprié dans la hiérarchie du design. Selon les auteurs, cette approche est limitée aux modèles dans lesquels les ports qui sont utilisés dans la description comportementale peuvent être déterminés statiquement.

3.2.2.3 Bilan sur les outils d'analyse de programmes SystemC

Chacun des outils présentés se heurte aux inconvénients de son approche (statique *vs.* dynamique). Une analyse syntaxique permet d'identifier de nombreuses relations. Mais sans exécution, récupérer les informations de structure d'un design qui n'a pas encore été instancié devient impossible sans l'adoption d'un style de codage/utilisation de nouveaux types bien définis. La complexité d'écrire un analyseur C++ vient s'ajouter à l'ajout de toutes les primitives dédiées à la reconnaissance de programme SystemC. D'un autre côté, attendre la fin de la phase d'élaboration, et ainsi récupérer un design bien instancié sans conserver un lien avec le code SystemC, limite l'interprétation de la vue structurelle que l'on en retire. Celle-ci est difficilement analysable par un concepteur de systèmes car elle a perdu tout concordance avec le code.

3.3 L'approche choisie

L'étude des travaux existants nous a convaincu de la nécessité de combiner et d'articuler au mieux les capacités des deux approches, statiques et dynamiques. D'autre part, et ce pour coller au mieux aux standards, nous désirons produire la structure du design extrait du programme original SystemC/C++ au format IP-Xact, lui-même dédié à l'assemblage ultérieur de

composants SystemC dans des systèmes plus larges. Et enfin nous souhaitons pouvoir opérer ces transformations et extractions aussi bien au niveau TLM qu'au niveau RTL plus bas. Tout ceci forme le cahier des charges de notre approche.

3.3.1 Description générale de l'approche

Partant du constat que ni l'approche statique ni l'approche dynamique n'est suffisante pour récupérer assez d'informations toute seule afin de nourrir un modèle architectural utilisable clairement d'un point de vue utilisateur, nous avons dû nous poser la question suivante : *comment extraire ces informations le plus génériquement possible et les exporter dans le formalisme* IP-X*act ?*. Les inconvénients présents dans une approche devront pouvoir être surmontés dans l'autre approche et vice-versa. Notre problème d'origine se décompose en trois questions indépendantes et plus facilement traitables :

- Comment réconcilier les informations issues des deux approches ?
- Comment stocker ces informations ?
- Comment exporter ces informations en IP-Xact ?

La figure 3.6 donne un aperçu des étapes du flot souhaité. L'objectif à réaliser est la traduction du bloc "Programme SystemC" vers le bloc "Modèle IP-Xact". Nous la divisons en deux étapes par l'introduction d'un bloc "Modèle structurel", sorte de format pivot afin de séparer deux sortes de considérations : depuis le "Programme SystemC " nous parlerons d'extraction (d'un modèle depuis un programme), la deuxième phase étant elle qualifiée de transformation de modèles (en fait la traduction entre notre modèle interne et la syntaxe IP-Xact).

On voit ici l'importance de définir un méta-modèle adéquat, afin que les modèles structurels exprimables dans le langage de ce méta-modèle aient justement à la fois la capacité de représenter les informations nécessaires à la transformation de modèles ultérieure tout en autorisant la production de ces informations de manière simple et efficace par la phase antérieure d'extraction. Ce méta-modèle est le résultat d'un affinement progressif basé d'une part sur les expériences précédentes (cf. notre étude bibliographique de la section 3.2.1), d'autre part sur des expérimentations conduites avec des versions préliminaires sur des bibliothèques disponibles (cf. section 3.5).

La phase d'extraction est raffinée pour intégrer les approches statiques et dynamiques d'analyse de la structure du design sous le code SystemC. La phase de transformations s'appuie sur les techniques de transformations de modèles décrites au niveau des méta-modèles. Il faut ici insister sur le fait que si ces trois aspects :

- Définition d'un méta-modèle structurel pivot
- Définition d'une méthode d'extraction de code SystemC vers ce modèle
- Définition d'une transformation de ce modèle vers IP-Xact

sont décrits de manière séquentielle et largement autonome, dans la pratique le choix de ces définitions a été largement interdépendantes pour permettre la meilleure articulation de ces étapes. Dans la suite nous décrirons dans l'ordre :

- Le méta-modèle structurel pivot en 3.3.2
- La transformation de ce modèle vers IP-Xact en 3.3.3
- L'extraction depuis le code SystemC en 3.3.4

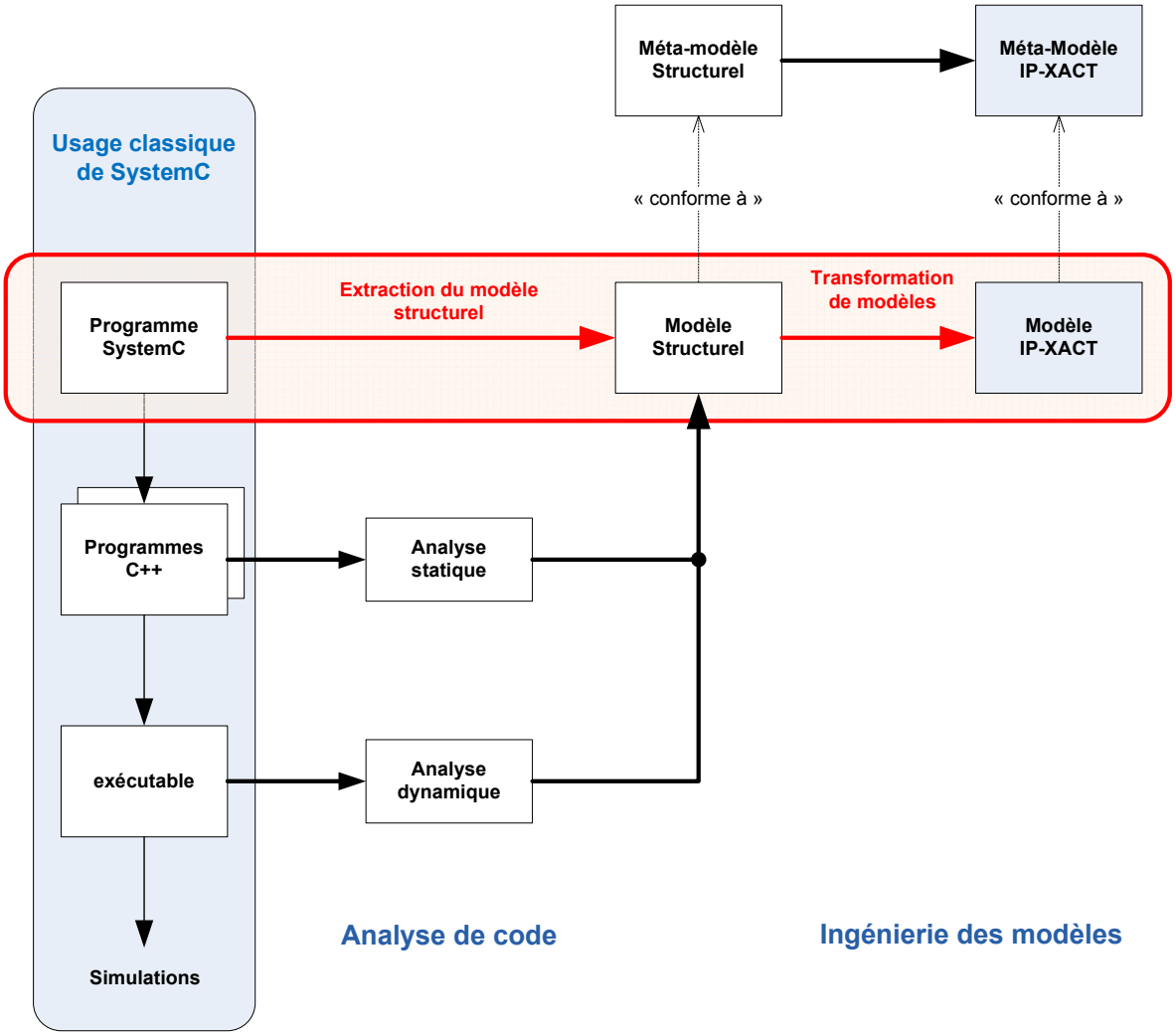


Figure 3.6 – Flot général

3.3.2 Un méta-modèle structurel de design

Afin de pouvoir décrire un modèle structurel, plusieurs concepts sont essentiels. La description d'un modèle d'architecture commence en premier lieu par la séparation claire entre les concepts de types et d'instances comme pour une approche orientée objet générique. Nous considérons un design comme un assemblage d'objets qui sont eux-mêmes des instances de types/classes. Chaque objet doit pouvoir être identifié de manière sûre et unique. C'est pourquoi le lien entre un objet et son type doit être préservé. La notion de composants nous permettra d'abstraire certain blocs complexes en de simples blocs en encapsulant les structures internes de ceux-ci. Afin de permettre le dialogue de ces blocs à leur niveau d'instanciation, la notion de point de connexion présent à la surface des composants est nécessaire elle aussi conjointement avec la notion de lien de connexion pour pouvoir construire réellement une architecture tout en conservant l'encapsulation permise par le concept de composant.

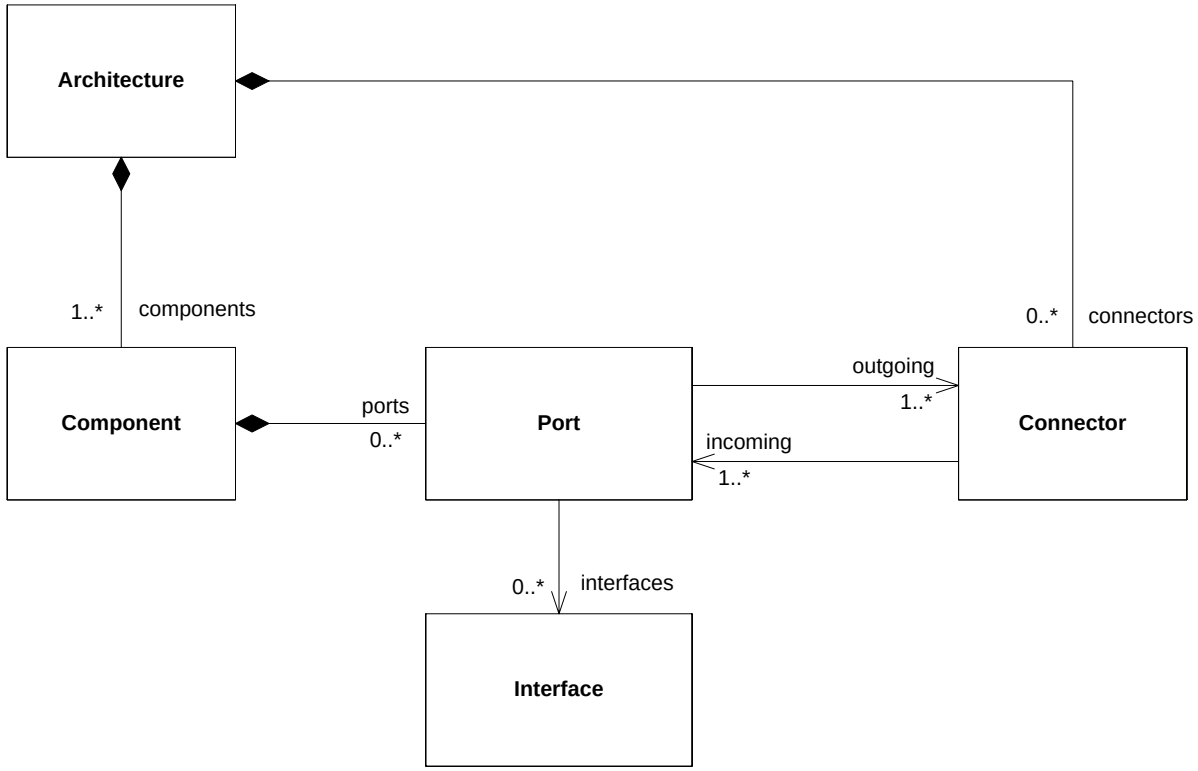


Figure 3.7 – Méta-modèle structurel

Les noms diffèrent des mots clefs utilisés dans la bibliothèque SystemC mais représentent les mêmes concepts. L'Architecture représente le niveau le plus haut de hiérarchie comme on peut le décrire dans le sc_main d'un programme SystemC. Elle contient des instances de Component et des

instances de Connector. Un Component représente un type générique de composants. Différentes spécialisations nous permettent alors de manipuler les différents types de composants dérivés de la classe sc_module définis dans du code SystemC. Un Component peut contenir des instances de points de connexions que nous nommons Port (semblables aux sc_port et sc_export présents en SystemC). Ceux-ci possèdent des références à leurs canaux de communication ou Connector et inversement. Il n'est ici question que d'extraire la structure d'un design, c'est pourquoi cette simple représentation suffit.

Il apparaît que la notion de flot orienté ait un rôle important, et cela à différents niveaux de représentation. Au niveau RTL, proche de la représentation électronique, il est nécessaire de pouvoir distinguer un point de connexion produisant de l'information (output) d'un point de connexion ne faisant que subir l'arrivée d'informations (input). Effectivement, pouvoir identifier les outputs permet alors de vérifier que ceux-ci ne tentent pas de propager des informations en même temps sur le même canal (*i.e.,*dans ce cas-ci un simple fil). Cela est critique d'un point de vue électronique si les informations ne sont pas les mêmes (la majeure partie du temps) pouvant mener à un endommagement du système physique (composants électroniques). Au niveau TLM, les points de connexion de type RTL restent présents et utilisés pour des communications dites critiques telles que les interruptions, mais il apparaît un autre type de point de connexion. Ces nouveaux points de connexion ne sont plus une simple liaison unidirectionnelle comme au niveau RTL. Des interfaces de communications complexes peuvent être utilisées pour accéder à un canal de communication encapsulant plusieurs liaisons ainsi qu'un protocole de communication. Là encore, les flots sont orientés mais d'une manière différente. Les directions représentent une relation de dominance entre un *maître* et un *esclave*. Plusieurs allers et retours d'informations peuvent être engendrés par une requête initiée par un *maître* à son *esclave*. Cette fois-ci le sens du flot est utile afin de prévenir d'un éventuel accès concurrent de deux maîtres à une même cible à un niveau d'abstraction plus élevé mais qui découle du même problème qu'au niveau RTL. Proposer deux différents méta-modèles dédiés à chaque niveau de représentation aurait été possible, cependant on constate que des connexions de type RTL subsistent dans des modélisations au niveau TLM. C'est pourquoi ces différents types de connexions sont regroupées sous le nom de *Port* dans notre méta-modèle. Ainsi, dans un modèle structurel conforme à notre méta-modèle, une connexion entre maître et esclave au niveau TLM sera interprétée comme une simple connexion entre un port d'entrée et un port de sortie du niveau RTL. Une connexion TLM sera vue comme un simple fil bien qu'elle en embarque plusieurs.

3.3.3 Les règles de transformation vers IP-Xact

Nos règles de transformations convertiront un modèle conforme au méta-modèle présenté ci-avant en un modèle conforme au méta-modèle d'IP-Xact tel que décrit dans la figure 2.8 de la section 2.4. Pour cela, les règles de transformations pour le modèle IP-Xact commencent par la

définition des types de composants qui seront utilisés dans le design. Tout composant sc_module découvert dans le design SystemC produit un composant librairie IP-Xact. Le champ *name* du composant (représentant son type) est alors la concaténation de son type au sens SystemC/C^{++} agrémenté des informations relatives aux parties (dynamique dans le code SystemC) devenues statiques par instanciation qui caractérise une version et une seule d'un composant.

Pour chaque composant, un champ *model* sera alors généré et contiendra une liste des vues du composant (dans notre cas une seule vue est créée)

ainsi qu'une liste des ports disponibles pour un composant dans le champ *ports*. Le port contiendra son nom, sa direction, son type instancié dans le programme SystemC et une référence vers la vue du composant définie précédement.

Les règles de transformations pour le modèle IP-Xact du design (cf. méta-modèle de la figure 2.9) sont les suivantes. Tout composant sc_module découvert dans le design SystemC produit une instance de composant (ComponentInstance) dans le modèle IP-Xact liée au composant IP-Xact précédemment défini. Le design contient la liste de tous ces composants ainsi que le réseau d'interconnexion.

Le nom du composant SystemC est associé au nom de l'instance IP-Xact. Le nom du type réel du composant SystemC instancié est utilisé pour l'instanciation du composant IP-Xact. La liste des canaux sc_channel est convertie en connexion ad-hoc IP-Xact et ceux-ci référencent les ports qu'ils connectent. Un aperçu de code XML généré est donné dans l'annexe A.

3.3.4 La production de notre modèle structurel à partir d'un code SystemC

La partie réellement technique, voire délicate, de la transformation globale consiste en l'extraction des informations structurelles et topologiques du design à partir de code SystemC, à répartir et organiser entre celles qui sont récupérées par analyse statique ou par exécution symbolique de la phase d'élaboration. Nous aborderons désormais cette phase importante, en décomposant bien les effets respectifs des analyses statiques et dynamiques, et leur réconciliation éventuelle pour intégration. Nous avons dû pour cela considérer la récupération des informations suivantes :

- le type d'un objet
- le nom d'un objet
- la hiérarchie d'un design
- la contenance d'un objet
- l'héritage d'un type
- les connexions entre entre objets

Lors d'une première lecture, il est possible de passer directement à la partie réalisation (section 3.4), réservant les détails des solutions apportées pour une lecture plus approfondie.

3.3.4.1 Les informations relatives aux types

Statique Une analyse statique du code SystemC permet initialement de récupérer des informations partielles de types de toutes les classes et structures définies. Dans un souci de réutilisation de code, celui-ci est souvent écrit de manière modulaire. L'utilisation des *templates* dans la définition de certaines classes (la plupart des classes de la librairie SystemC) rend l'information sur le type réellement instancié complexe à retrouver en théorie. Cependant, à partir du moment où une spécialisation apparaît dans le code analysé, qu'elle soit utilisée ou non durant l'exécution du programme, il est possible de la récupérer sous forme de type statique (Type < Template >).

Dynamique L'utilisation de la bibliothèque RTTI (*Run Time Type Information*) nous permet de récupérer le type instancié d'un objet sous la forme d'une chaîne de caractères encodée par le compilateur (*mangling* fournissant un identificateur unique du type de l'objet instancié). Cette chaîne de caractères n'étant pas humainement lisible, nous y associons l'utilisation de la fonction __cxa_demangle(). Cette fonction fait partie de l'API standard <cxxabi.h>. Nous récupérons ainsi une chaîne de caractères de manipulation plus aisée qui contient le type réellement instancié durant l'exécution. Une analyse des différents modes de déclaration de variable dans un programme C++ nous a conduit à la conclusion que les informations relatives à un type instancié que nous pouvons récupérer sont d'une forme particulière, formé d'un *Type* concaténé le cas échéant avec un paramètre *template* identifiable par le fait qu'il est compris entre les symboles < et >.

Conciliation Les informations issues des deux approches se recoupent en un point, le type statique. Connaissant le type instancié d'un objet (de la forme Type < Template > $[('*')^*|_{\sqcup}([n])^*]$), on peut alors dériver le type statique (Type < Template >). Ainsi, un objet dont nous récupérons le type instancié peut être lié à sa déclaration dans le code SystemC. Nous pourrons donc relier les informations statiquement extraites à un objet durant la simulation par ce biais.

3.3.4.2 Les informations relatives à la hiérarchie

Statique La figure 3.8 représente les informations récupérées par une analyse statique dans le cas où les points de connexion sont statiquement définis dans les composants. Malgré cela, dans le cas d'un instanciation dynamique, certaines parties d'un design ne pourront pas être récupérées. L'analyse révèle qu'une structure/classe contient un pointeur (dont on connaît le nom) vers un certain type (en général non totalement définis comme indiqué dans la section précédente) ; mais rien ne permet de savoir si un objet y est réellement référencé. Un tel style de codage peut s'appliquer aussi bien à la déclaration des composants.

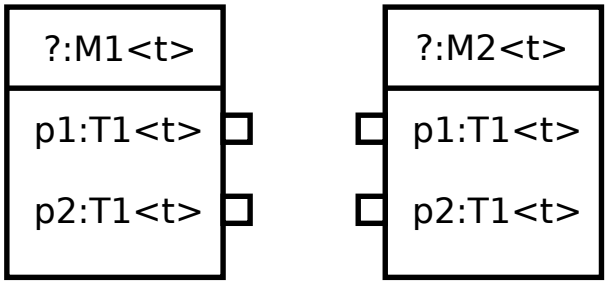


Figure 3.8 – Design simple vue statiquement

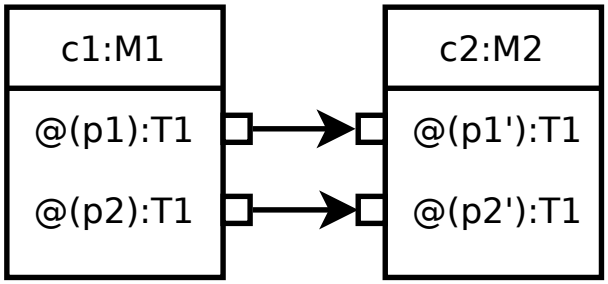


Figure 3.9 – Design simple vue dynamiquement

Dynamique La hiérarchie après l'exécution de la phase d'élaboration de SystemC est contenue dans l'objet sc_simcontext. Par le parcours de celui-ci, il est possible de la récupérer. Les informations récupérées sont des listes de pointeurs d'objets de types sc_object dans le cas général. Certains de ces objets font aussi partie de listes de pointeurs spécifiques telles que des listes de composants sc_module, points de connexion sc_port/export ou encore les connexions génériques sc_prim_channel comme déjà décrit dans la section 2.2.1. A partir de ces listes, il est possible de construire un arbre représentant la hiérarchie grâce à une fonction de la classe sc_object. Cette fonction se nomme get_parent() et retourne un pointeur sur un sc_object. Il faut n'effectuer ce traitement qu'après la fin de la phase d'élaboration pour récupérer la hiérarchie d'un design instancié de manière sûre (c'est le moment où le design devient figé).

3.3.4.3 Les informations relatives aux noms des objets

Statique Une analyse statique de code C^{++} classique permet de récupérer le noms de tous les attributs d'une structure/classe. On peut ainsi savoir la composition d'une structure/classe de manière sûre, ceux-ci étant figés par la définition de la structure/classe. Cependant, le nom d'une instance de cette même structure/classe ne peut être récupéré facilement. Il faut pouvoir identifier un élément racine.

Dynamique Certains éléments tels que les sc_module embarquent un champ spécial obligatoire contenant leur nom. Celui-ci est requis à la construction de l'objet. Il nous suffit alors de consulter celui-ci pour récupérer le nom d'un composant. Comme énoncé dans la section 3.1.2, de manière plus générale, les objets n'embarquant pas le champ de nom sont en quelque sorte devenus anonymes. C'est le cas des éléments de type Port ou encore Connector. La seule chose qui puisse différencier ces objets est leur adresse mémoire. Celle-ci pourra être récupérée facilement en interrogeant la valeur du pointeur.

Réconciliation Nous avons dû établir une manière de déduire de ces adresses le nom des points de connexion afin de pouvoir les différencier. Partant du fait que nous pouvons récupérer les informations relatives à l'adresse d'un composant en mémoire et qu'une spécification SystemC supplémentaire nous indique que les *Ports* sont contenus dans un composant (sc_module), nous pouvons récupérer la différence d'adresse entre un composant et l'un de ces points de connexion. Plus généralement, nous pouvons récupérer la différence d'adresse entre tous les attributs d'une structure/classe et celle-ci à partir du moment où nous possédons une référence à chacun d'eux.

D'autre part, il existe une macro C^{++} offsetof() qui permet de récupérer le décalage d'adresse d'un champ dans une structure. Pour utiliser cette macro, deux informations cruciales sont nécessaires, le type instancié de la structure et le nom de l'attribut à tester. C'est pourquoi, il nous faut entremêler analyse statique et dynamique de la manière décrite par la figure 3.10 afin de récupérer les informations nécessaires à son appel. Le type réellement instancié d'un composant est alors récupéré durant l'analyse dynamique pour être "généralisé" comme décrit dans la section 3.3.4.1. Ce type généralisé nous permet ensuite d'interroger le résultat de l'analyse statique qui lie les structures et classes généralisées aux noms de leurs membres.

Nous sommes alors en possession d'assez d'informations pour appeler la macro offsetof() mais sa signature prend en entrée des paramètres et non des chaînes de caractères. Nous avons donc été amené à instrumenter le programme pour l'utiliser. Nous générons alors un code C^{++} pour interroger tous les couples de composants/membres et récupérer ce décalage d'adressage. Après compilation et exécution, le résultat obtenu est une base de données liant les composants, le nom de leurs membres et leur décalage d'adresse. Une comparaison entre ces décalages d'adresse et ceux issus des objets réels nous permet alors d'associer le nom d'un membre et l'adresse d'un objet contenu dans un composant. Cependant, cette méthode doit faire appel à une exécution externe (*i.e.,*en dehors du programme principal utilisé pour l'analyse dynamique) car elle nécessite une nouvelle phase de compilation du code généré.

Bien qu'adaptée dans la majeure partie des cas qui se présentent, il est à noter que l'utilisation de cette macro est limité à l'interrogation des membres publics d'une classe/structure.

Nous avons été amené à étendre la démarche pour traiter les pointeurs. En partant du type instancié, nous pouvons déterminer si une variable est de type statique ou dynamique

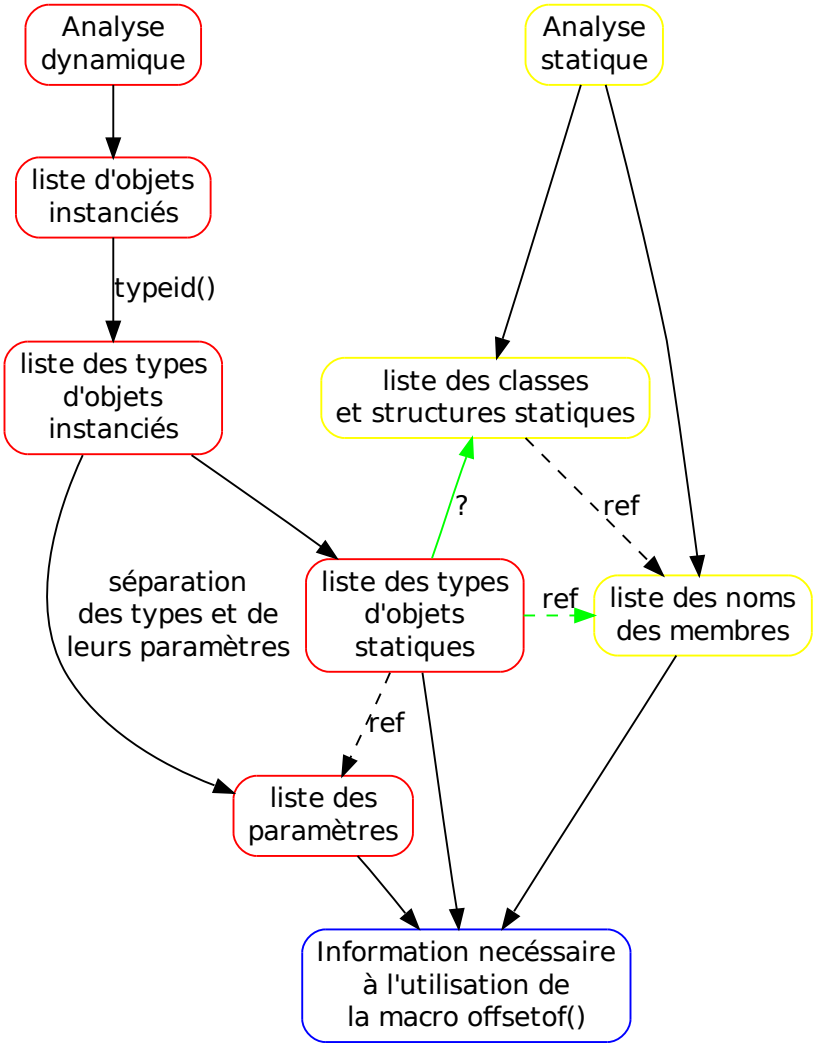


Figure 3.10 – Récupération du nom des membres

grâce à la forme de la chaîne de caractères retournée (section 3.3.4.1). Nous avons départagé la reconnaissance de la manière suivante. Si le type instancié est de la forme Type < Template > l'objet est alors dit statique simple et la méthode décrite ci-avant s'applique directement afin de réconcilier le nom à l'objet. Si le type est de la forme Type < Template >$_{\sqcup}([n])^*$, il suffit de consulter la valeur de l'adresse de chaque case du tableau statique et ainsi générer les différents noms correspondants à chaque case du tableau statique à partir du nom du tableau. Pour les cas ou nous rencontrons la forme Type < Template > $('*')^*$, aucune indication quant à la borne du

potentiel vecteur n'est disponible. Dans le cas le plus simple où il n'y aurait qu'une seule *, on teste tout d'abord si le pointeur référence bien un objet et le cas échéant on applique la même méthode que pour le cas Type < Template > mais cette fois-ci en effectuant la réconciliation sur l'objet pointé et non sur le pointeur. Si ce pointeur n'est que le début d'un tableau il faut alors mémoriser cette information et étendre la réconciliation aux valeurs d'adresse du début du tableau auxquelles on ajoute la taille en mémoire du type instancié du tableau. Nous n'avons cependant pas exploré le cas où le nombre d'étoiles est supérieur à un, ce qui constitue en soit une limitation de notre approche.

3.3.4.4 Identification des spécialisations de ports

Détection par héritage Tout point de connexion de la librairie SystemC dérive des classes sc_port_base et sc_export_base comme le décrit le diagramme d'héritage de la figure 3.11. C'est le seul moyen pour qu'il subsiste une trace de ces objets durant l'exécution. Sont alors dérivés de cette classe primitive les différents types de Ports RTL tels que les *entrées/sorties* ou encore les *files* (contenus dans le rectangle rouge).

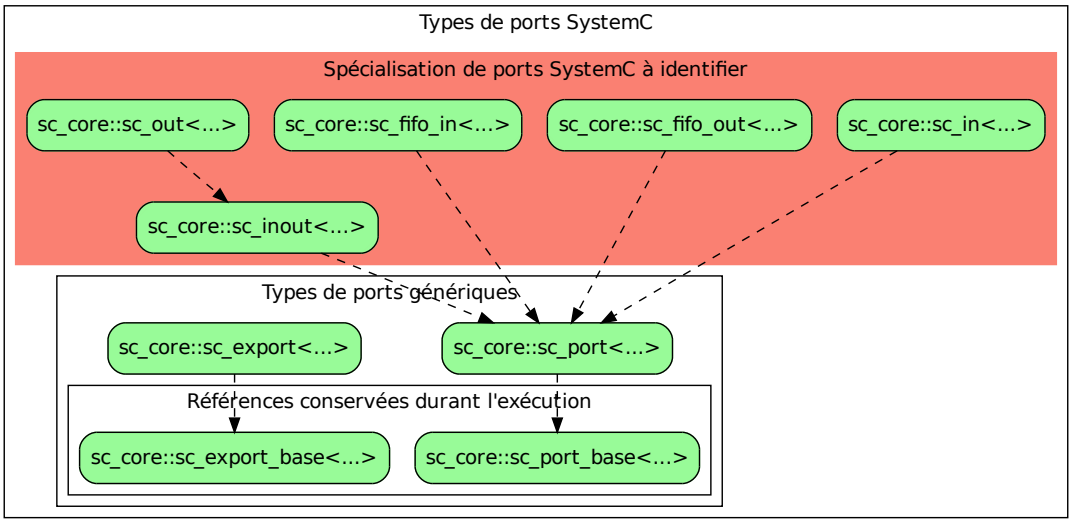


Figure 3.11 – Diagramme d'héritage des Ports SystemC

La version officielle de la librairie TLM définit les types de connexions à haut niveau comme héritant des classes sc_port_base ou sc_export_base comme nous l'avons expérimenté avec nos tentatives préliminaires de traduction sur des bibliothèques existantes (cf. section 3.5). Certaines

librairies telles que GreenSocs ou AMBA-pv implémentent certains types de Ports transactionnels comme dérivant de ces types (figure 3.12). De ce fait, la détection des types de bases semblent suffisante pour pouvoir identifier les dérivés.

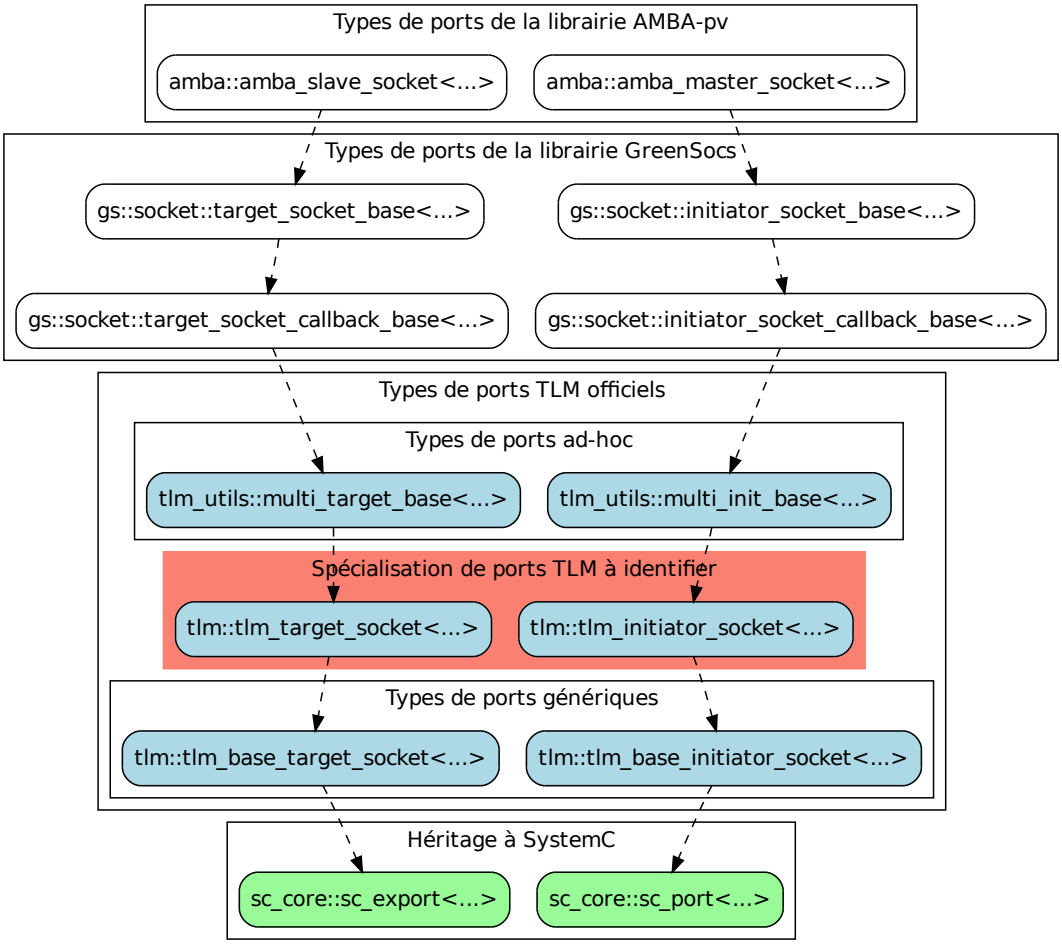


Figure 3.12 – Diagramme d'héritage des Ports TLM

Nous avons donc élaboré une manière de distinguer les différences entre ces points d'interconnexion de manière à rester compatible avec les différentes librairies. Nous nous sommes basés sur la structure de donnée interne de SystemC contenant les informations génériques concernant les sc_port_base et sc_export_base instanciés durant la phase d'élaboration d'un design. Il s'agit d'une liste de références de tous les objets dérivants de sc_port_base et d'une autre pour les

objets qui dérivent de sc_export_base. Cependant, un *dynamic_cast* vers le type de port spécialisé ne peut être utilisé car le type d'arrivée requiert des paramètres template. L'information dont nous disposons concernant le type véritable d'un sc_port/export_base est le type instancié. Cette vue du type rend l'interprétation du paramètre *template* trop complexe pour être traitée de la sorte. Là encore, le recours aux informations issues de l'analyse statique devient nécessaire. Il est possible de parcourir les informations statiques dans le but de savoir si un type statique est une version template particulière d'un autre type ainsi que de savoir si le type non spécialisé dérive d'un autre et cela sans avoir à spécifier le paramètre *template*.

Détection de contenance Un dernier obstacle subsiste : aucune règle n'empêche un développeur de créer ses propres types de Ports à partir de ceux fournis par SystemC. Pour cela il dispose de plusieurs possibilités telles que l'héritage, la contenance ou encore un construction hybride à partir de n'importe quel type dérivant de sc_port_base ou sc_export_base. C'est pourquoi nous ne pouvons nous limiter à la détection des seuls types contenus dans les rectangles rouges. D'autres librairies de composants telles que SoCLib, GreenSocs, et certains exemples de la librairie SystemC officielle car plus anciennes, implémentent d'autres types de port transactionnels. Jusqu'à présent, nous avons supposé que les ports étaient contenus directement dans leurs composants, ce qui est juste d'un point de vue modèle mais peut avoir différent encodage en C++. D'un point de vue dynamique, le niveau de contenance du port ne pourra être récupéré directement. En d'autres termes, en utilisant la méthode décrite à la section 3.3.4.2, nous ne pouvons pas déterminer si un sc_port/export est un attribut du composant au sens C++ ou bien si il est contenu dans une structure qui l'encapsule, elle-même contenue dans le composant. Le cas échéant, cela rend inutilisable le calcul du décalage d'adresse. Dans le cas de la librairie GreenSoCs, il existe une autre définition des ports transactionnels (différente de celle décrite dans la figure 3.12). L'analyse manuelle du code d'un design l'utilisant nous a permis de constater que les ports transactionnels ne faisaient pas partie des structures reconnues par la méthode utilisant les héritages comme mode de reconnaissance. La figure 3.13 met en évidence l'encodage du design au sens C++ d'un des exemples utilisant ce genre de ports. Cette exemple contient au niveau le plus haut de la hiérarchie deux composants (sillysort et simplememory). Ces deux composants contiennent des ports transactionnels bien particuliers GenericInitiatorBlockingAPI< ... > (équivalent des tlm::initiator_socket) et GenericTargetBlockingAPI< ... > (équivalent des tlm::target_socket). En interrogeant les attributs des composants, nous pouvons récupérer les décalages d'adresses de ces ports transactionnels par la macro offsetof() mais ces valeurs ne correspondront en aucune manière au décalage d'adresse entre les objets de type composants et les objets de type sc_port_base. Pour cela il suffit d'étendre la recherche des sc_port_base et sc_export_base aux différents attributs des attributs des composants et cela de manière récursive car aucune contrainte de profondeur n'est imposée en terme de style de codage C++.

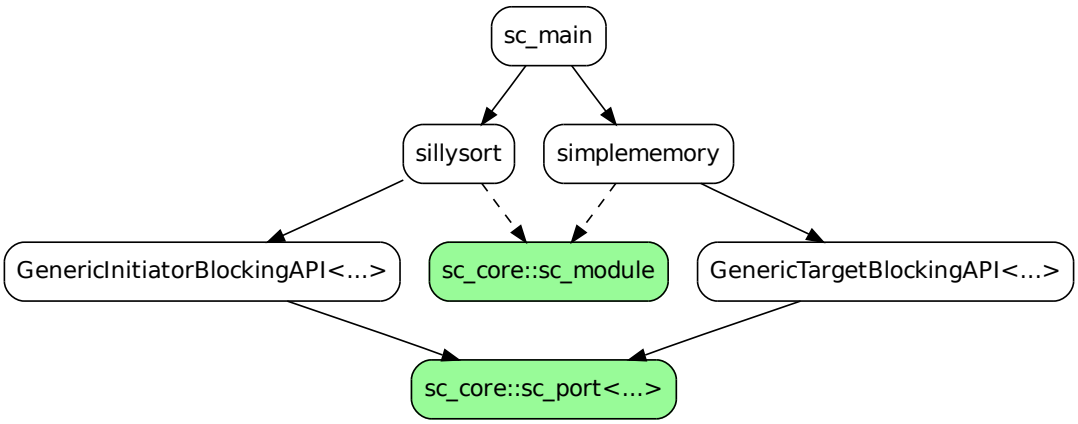


Figure 3.13 – Structure d'un exemple GreenSoCs

3.3.4.5 Les connexions entre Ports

Cette dernière étape est cruciale afin de tisser le réseau d'interconnexion entre les différents Ports de composants. Elle n'est cependant pas la plus compliquée car l'information nécessaire est contenue dans les Ports eux-mêmes. Effectivement ceux-ci nécessitent des références vers les canaux de communication qu'ils vont utiliser durant la phase de simulation. De la même manière que pour l'instanciation des composants et des Ports, nous ne pouvons être certains de l'existence d'un lien que durant la phase active du programme SystemC. L'analyse statique est donc à proscrire mais une simple consultation d'un Port nous permet de lier celui à un canal. Ensuite, si deux Ports sont reliés à un même canal, cela signifie qu'il existe un lien entre ces Ports.

3.4 Mise en œuvre : l'outil SCiPX

3.4.1 Présentation

Nous avons développé un environnement de traduction, nommé SCiPX (pour *SystemC to IP-Xact*), qui combine et articule les analyses statique et dynamique telles que décrites en section 3.3. Son flot de traitement général est représenté dans la figure 3.14. Il prend en entrée du code SystemC (en haut à gauche) et produit en sortie un modèle IP-Xact (en bas à droite). Tout d'abord, nous utilisons un analyseur de code source C^{++} sur le code SystemC à analyser. Il en résulte plusieurs fichiers XML (case jaune sur la figure) qui contiennent toutes les défini-

tions de classes, leur héritage, leur attributs, etc... Ces fichiers XML peuvent ensuite être filtrés pour recueillir la définition des divers composants génériques (*i.e.,*héritant d'un sc_module). Le programme original SystemC est alors compilé et exécuté jusqu'à la fin de l'élaboration par un analyseur dynamique. Lorsque ce point est atteint, l'analyseur dynamique fournit un accès à la représentation du réseau des composants (case rouge sur la figure). Nous combinons alors ces deux types d'informations afin de générer d'une part une bibliothèque de composants utilisés par le design, c'est-à-dire les composants génériques augmentés des types réellement instanciés (jaune orangé sur la figure) et d'autre part nous complétons les informations dynamiques afin de générer le modèle structurel du design analysé (en rouge orangé).

Partant de cette vue du design, on construit une représentation IP-Xact liées aux définitions de composants appropriées (en vert pâle sur la figure). La description IP-Xact est alors une vue structurelle de la description originale SystemC.

3.4.2 Méta-modèle et transformation

Nous avons utilisé comme briques de base les élément fournis par le méta-modèle ECore présent dans le projet EMF (*Eclipse Modeling Framework*) pour construire un modèle ECore du méta-modèle structurel présenté dans la partie précédente (figure 3.7). Notre méta-modèle a ensuite été converti en méta-modèle générique (XML *Metadata Interchange* ou XMI qui est un standard pour l'échange d'informations de méta-données UML basé sur XML.) nous donnant un schéma XML (XSD) reflétant ce dernier. Ce schéma a ensuite été traduit par l'outil CodeSynthesis XSD xsdtool en classes et structures C++ pour chaque élément du méta-modèle. Il intègre dans ces structures les contraintes liées aux constructions permises par notre méta-modèle. Cela nous permet de nous assurer de la validité d'une construction. En plus de générer ces conteneurs, l'outil CodeSynthesis XSD génère une structure permettant de sérialiser les objets C++ créés dans un fichier XML ainsi que le chemin inverse permettant de construire les objets C++ d'un modèle à partir d'une description contenue dans un fichier XML. La figure 3.15 donne un aperçu global de l'interface entre les modèles et le code C++.

Afin de réaliser la transformation entre SystemC et IP-Xact, nous nous sommes appuyés sur la technologie Java. Nous avons implémenté les règles de transformation énoncées dans la section 3.3.3. Nous avons ensuite rendu le code Java exécutable indépendamment d'eclipse afin de pouvoir le lancer par ligne de commande et ainsi réaliser le flot décrit à la figure 3.16. Nous passons le modèle structurel (qui doit être conforme à son méta-modèle) en paramètre d'appel pour en récupérer un modèle IP-Xact (qui est conforme à son méta-modèle) en sortie.

3.4.3 Extraction de modèle de design

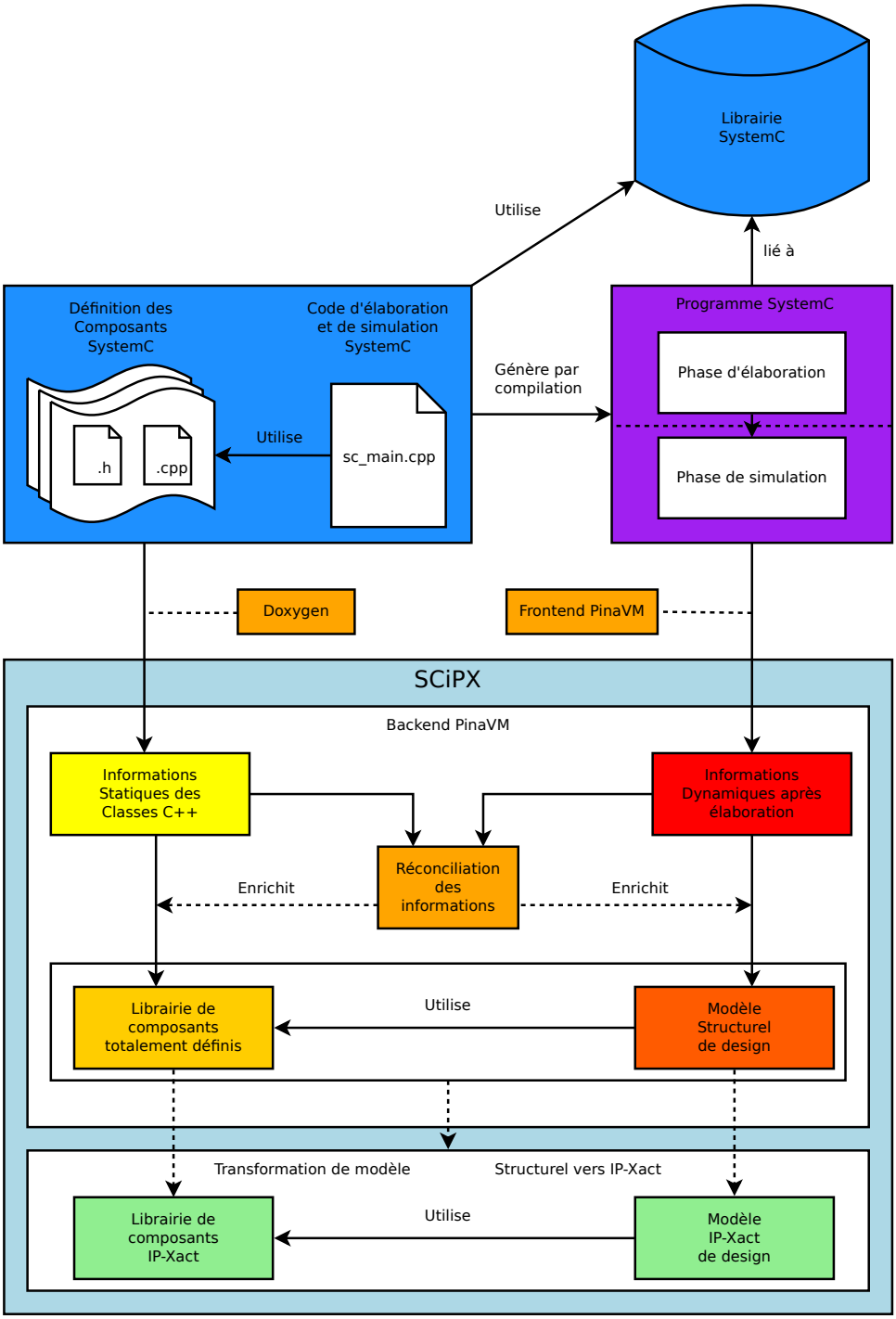


Figure 3.14 – Flot de traitement global de SCiPX

Analyseur statique Pour la phase d'analyse statique, nous avons utilisé Doxygen doxy. Doxygen est à l'origine un générateur de documentation, mais peut aussi être utilisé comme un analy-

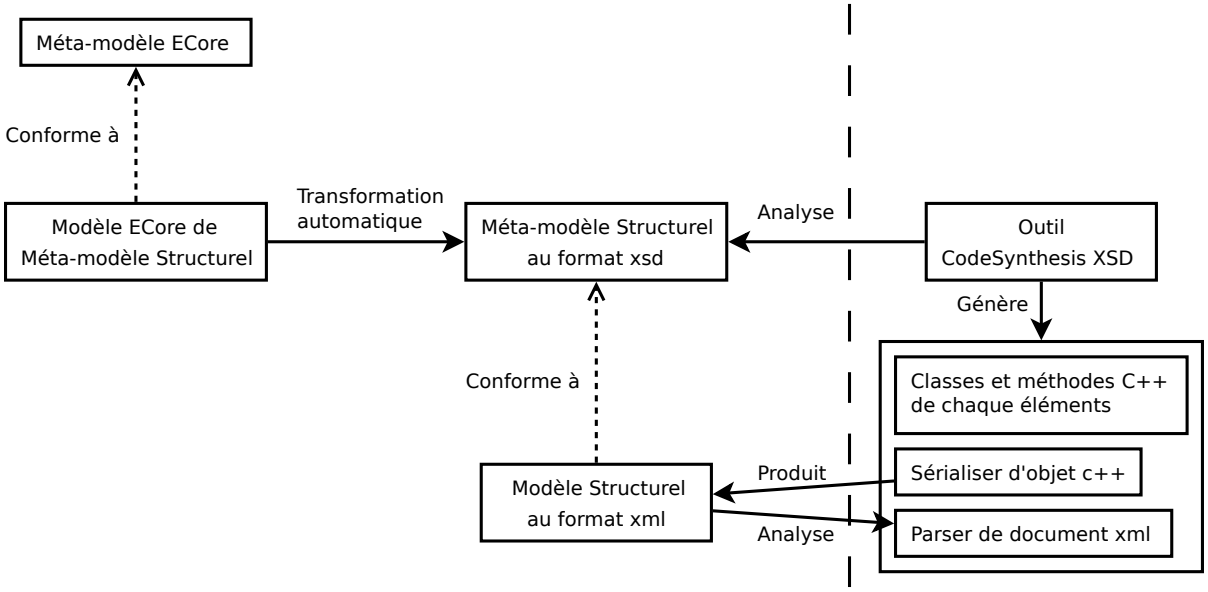


Figure 3.15 – Interface modèle/C^{++}

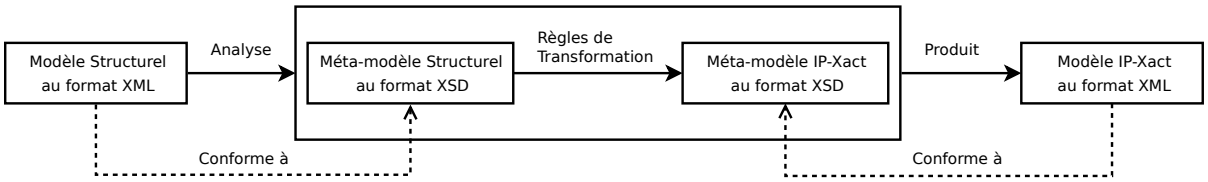


Figure 3.16 – Transformation

seur de code source statique. Il nous permet de récupérer des informations sur la définition des composants comme décrit dans la section précédente. Dans la plupart des cas, Doxygen ne peut donner toutes les informations pertinentes concernant le réseau de composants qui sera instancié à l'exécution. Cependant, Doxygen nous permet de générer un ensemble de fichiers XML représentant tous les éléments du code SystemC qu'il aura analysé à partir du fichier contenant la fonction sc_main. Nous utilisons dans SCiPX Doxygen-1.7.3 [1]. Ce logiciel produit des fichiers XML qui contiennent les résultats d'une analyse de code. Cette version du logiciel embarque également un analyseur syntaxique de ces fichiers XML permettant une interrogation simplifiée des informations recueillies.

1. disponible à l'adresse : `ftp://ftp.stack.nl/pub/users/dimitri/doxygen-1.7.3.src.tar.gz`.

Analyseur dynamique Pour cette phase, notre choix s'est porté sur l'outil PinaVM déjà décrit dans la section 3.2.2.2. Il nous fournit une représentation abstraite d'un programme SystemC après la phase d'élaboration (il facilite l'accès à une vue abstraite du réseau des composants). PinaVM utilise une version légèrement modifiée de la librairie SystemC qui permet de récupérer la main lorsque la phase d'élaboration est totalement achevée. On a alors accès un pointeur vers la structure interne de SystemC (sc_simcontex) ainsi que les informations produites par le compilateur LLVM. La représentation abstraite fournie par PinaVM est adaptée à la vérification/validation d'un point de vue comportemental et peut se contenter de la vue anonyme d'un design afin d'effectuer ses analyses. Cependant, la traduction de la structure dans une vue modèle (IP-Xact ou UML) sans l'ajout d'informations devient alors très lointaine de la version code. Néanmoins, PinaVM procure une structure de projet extensible permettant le développement de back-end spécifique. Nous avons choisi de profiter de cette structure de projet et avons développer un back-end pour réconcilier (au sens de la section 3.3.4) les informations issues de Doxygen et fournies par PinaVM.

Notre extension à PinaVM "XMLbackend" Nous avons implémenté les différentes méthodes nécessaires pour produire un modèle IP-Xact telles que décrites en sections 3.3.4. Nous commençons par une analyse statique des sources SystemC et stockons ces informations. Nous compilons ensuite le programme SystemC à l'aide de la librairie fournie par le front-end PinaVM et récupérons les listes d'objets de la simulation.

La première étape est de construire la cartographie d'un composant (et plus généralement d'un type instancié) en termes de nom d'attribut, type d'attribut et décalage d'adresse relatif. La figure 3.17 décrit la mécanique sous-jacente mise en œuvre. Les informations statiques sont représentées en jaune et afin de limiter la recherche (et le temps de traitement), nous partons de l'ensemble des composants instanciés contenus dans la liste d'objet sc_module que nous fournit le front-end PinaVM. Pour chaque composant, nous récupérons son type instancié (ou spécialisation d'un composant). A l'aide de ce type, nous déduisons le type statique et interrogeons la base de donnée d'informations générée par Doxygen. Grâce à une API que nous avons développée, nous récupérons les noms des attributs du type statique interrogé. Nous générons un code C++ appelant successivement la fonction offsetof() sur tous attributs publics du type instancié. Après compilation et exécution, nous récupérons la valeur du décalage d'adresse de chacun des attributs. On notera, la présence d'une boucle de retour sur la figure 3.17. Elle permet d'explorer en profondeur un composant en réitérant l'opération sur chaque attribut. A partir de ce traitement, on obtient une cartographie de toutes les adresses où l'on peut trouver des objets accessibles dans un composant ainsi que les noms des attributs.

Ensuite, comme le décrit la figure 3.18, en parcourant la liste des objets de type Port (comprendre ici la liste des sc_port et sc_export), nous récupérons par la méthode get_parent() (héritée

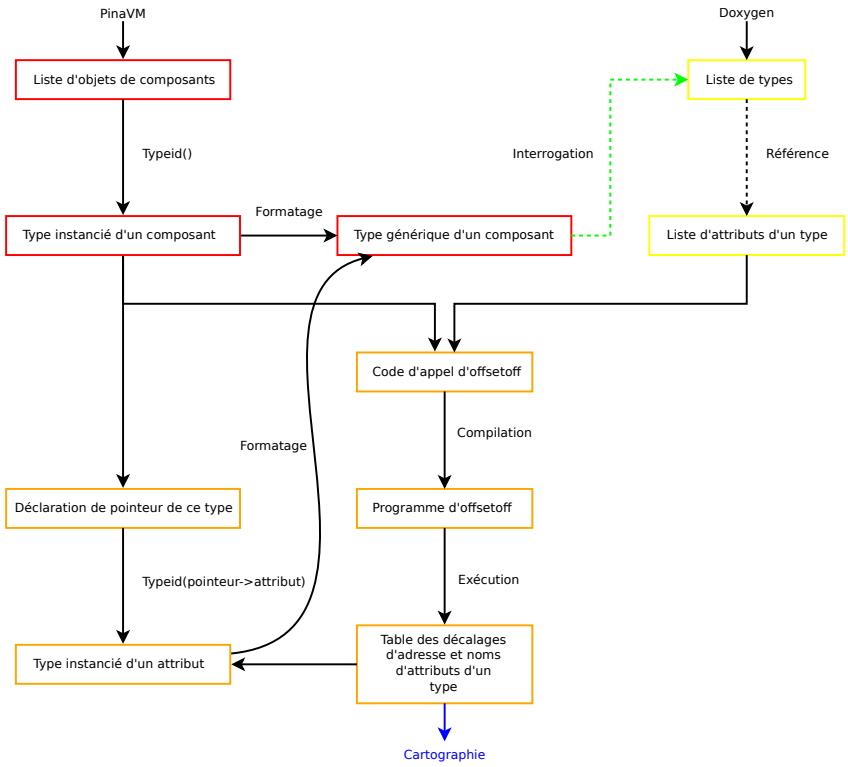


Figure 3.17 – Cartographie des composants

de la classe sc_object) l'appartenance de ces Ports aux différents composants instanciés. En consultant leur adresse et celle de leur composant parent, nous calculons le décalage d'adresse. Nous pouvons dès lors consulter la cartographie du composant précédemment construite afin de récupérer et ré-associer, le cas échéant, le nom d'un port à un objet de type Port. Tous les Ports non identifiés sont alors stockés en vue d'un traitement ultérieur. Effectivement, lors de la détection d'un socket initiateur TLM, le sc_export qu'il contient ne sera pas identifié par la présente passe car celui-ci est un attribut protégé du socket. Cela se fera lors de la phase de détection des types (ou spécialisations) de Ports.

Lorsque les phases précédentes sont terminées, nous possédons deux listes d'objets Ports. La spécialisation des Ports reconnus se fait alors dans l'ordre indiqué dans la figure 3.19. Cet ordre doit être respecté à cause de l'imbrication des différentes spécialisations décrites dans la section 3.3.4.4. L'implémentation des sockets TLM occulte certains sc_port et sc_export vis à vis des phases précédentes. Cependant, celle-ci définit les méthodes get_base_port() et get_base_export(). Afin de pouvoir les utiliser il faut donc au préalable identifier les spécialisations des Ports reconnus pour déterminer si ils sont des sockets. Une suite de tests est alors appliquée sur le type du Port. Si il est spécialisé en socket TLM nous utilisons les méthodes précédentes afin de récupérer une référence vers les sc_port ou sc_export qu'ils contiennent. Ceux-ci viennent alors

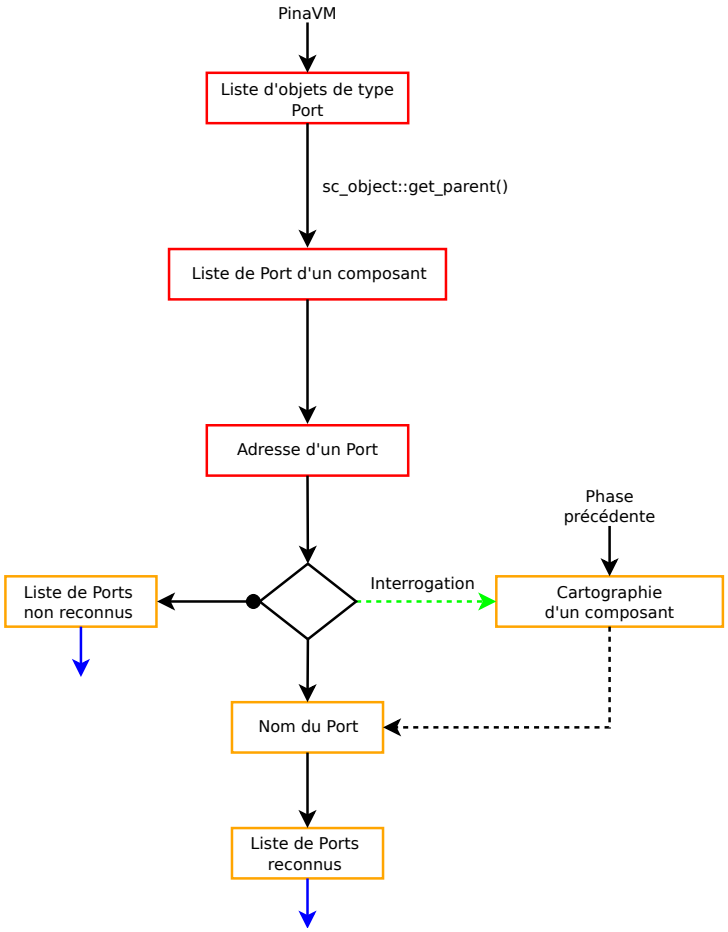


Figure 3.18 – Récupération des noms de Ports

grossir la liste des Ports reconnus lorsqu'une concordance est trouvée. Si tout ce passe bien, à la fin de l'analyse des spécialisations de Ports, la liste des Ports non reconnus est vide. Si ce n'est pas le cas, on attribue un nom généré aux Ports encore anonymes. Ils suivent alors le même traitement que les autres.

Le modèle complet du design est construit à l'aide de toutes les informations recueillis lors des traitements successifs. Il peut alors être validé structurellement et transformé en modèle IP-Xact.

3.5 Expérimentations et test

Nous avons utilisé SCiPX sur des exemples SystemC (*i.e.*,RTL et TLM) extraits des bibliothèques existantes du domaine public et propriétaire. Nous allons donner les résultats en soulignant les spécificités de chaque bibliothèque, les difficultés qui ont été rencontrées et les

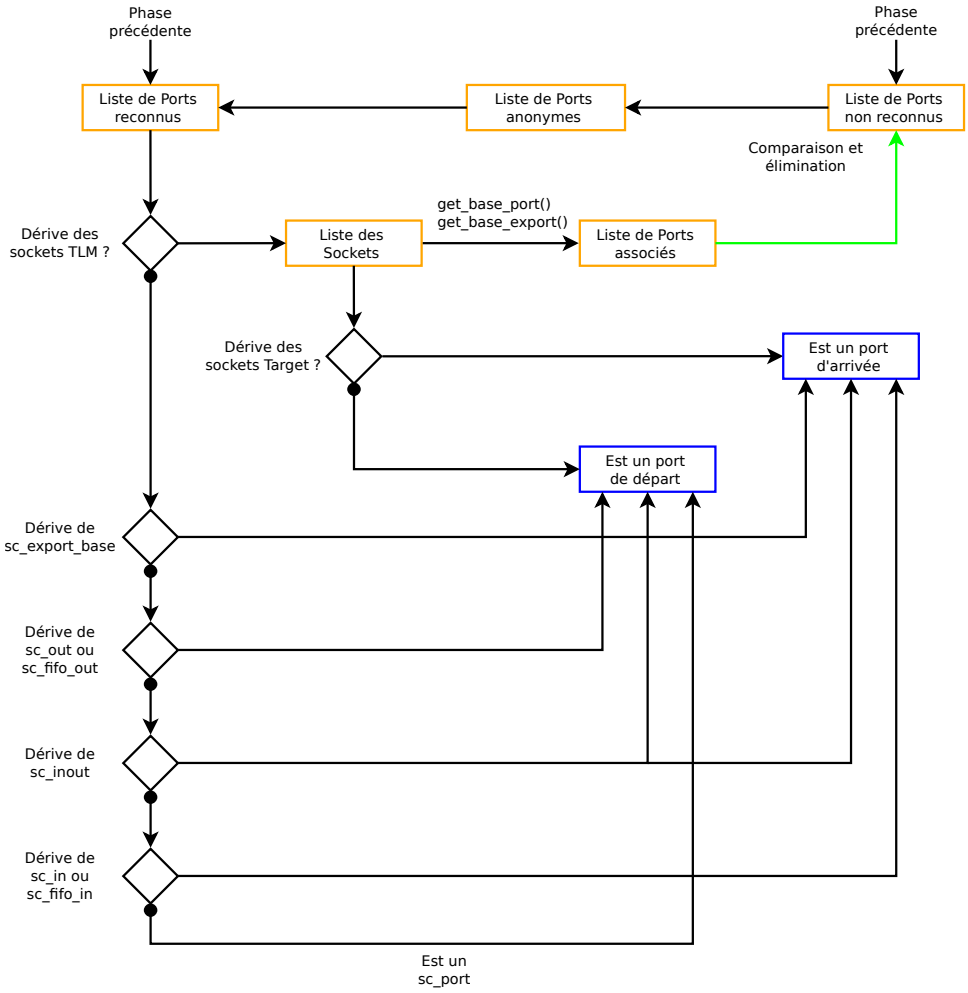


Figure 3.19 – Récupération des types de Ports

améliorations apportées à SCiPX pour les résoudre.

3.5.1 Librairie SystemC officielle

La librairie de référence du site de OSCI (*Open SystemC Initiative*) contient un certain nombre d'exemples standards qui nous ont servi de cas de test initiaux. La majorité de ces exemples sont des modèles de niveau RTL et ont été analysés correctement par notre outil. Ne souhaitant pas tous les détailler, nous présentons, en annexe A, un exemple de programme SystemC regroupant des constructions de type RTL ainsi que le résultat de l'extraction de modèle IP-Xact.

Un exemple de cette bibliothèque en particulier laisse entrevoir les prémices d'une modélisation TLM bien avant l'apparition de la librairie TLM officielle. Il représente un design générique composé de trois composants maîtres connectés à deux composants esclaves par l'intermédiaire

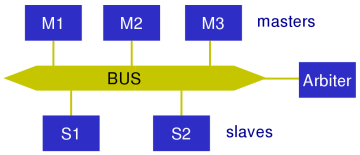


Figure 3.20 – Exemple simple_bus

d'un composant bus. La figure 3.20 en donne une illustration tirée de la documentation SystemC. Les connexions entre un maître vers le bus(esclave) et le bus (maître) vers un esclave ne sont pas standard mais tout à fait valide d'un point de vue SystemC/C++. Le composant maître embarque un Port lui permettant de communiquer avec un esclave. Cependant, l'esclave ne déclare pas de Port d'arrivée susceptible d'être le point de raccordement à son maître. La connexion existe mais est occultée par l'utilisation directe de pointeurs vers l'objet composant esclave. Pour cette raison, SCiPX n'est pas en mesure de traiter cet exemple et d'en générer un modèle IP-Xact correct. Cependant, n'étant pas l'implémentation officielle des ports transactionnels, elle n'est que peu utilisée.

3.5.2 Librairie TLM officielle

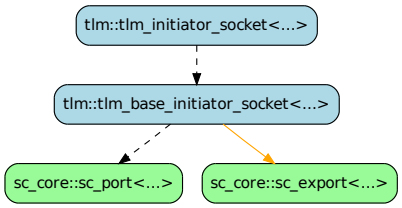


Figure 3.21 – TLM initiator socket

Cette librairie implémente de manière standardisée les Ports transactionnels (section 2.2.3). Elle permet de fixer des bases communes pour différentes implémentations de socket TLM (figure 3.12, page 57) adaptées à différents protocoles. La spécification TLM impose de plus l'utilisation de ceux-ci pour la conception d'un design ce qui élimine le problème rencontré dans l'exemple précédent de la librairie SystemC "non TLM". L'implémentation de ces connexions TLM est décrite dans les figures 3.21 et 3.22.

Les tlm::tlm_initiator_socket< ... > dérivent publiquement de tlm::tlm_base_initiator_socket< ... >. Ce dernier contient publiquement un sc_core::sc_port par héritage et un sc_core::sc_export par contenance en mode protégé. Cette contenance protégée ne nous permet pas d'accéder à cet sc_core::sc_export par la cartographie d'un composant. Néanmoins, nous contournons cette li-

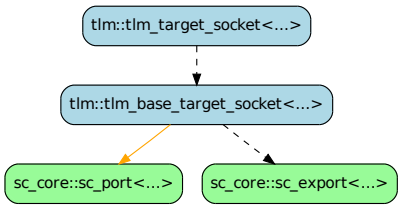


Figure 3.22 – TLM target socket

mitation grâce à l'utilisation d'une méthode déclarée dans tlm::tlm_base_initiator_socket< ... > : virtual sc_core::sc_export<BW_IF> & get_base_export() { return m_export; }. De la même manière pour un tlm::tlm_base_target_socket< ... >, il existe une méthode pour accéder sc_core::sc_port qu'il contient de manière protégée : virtual sc_core::sc_port_b< BW_IF > & get_base_port(). Nous avons sélectionné dans cette librairie un exemple utilisant différents types de connexions. Le schéma du design en question est décrit dans la figure 3.23.

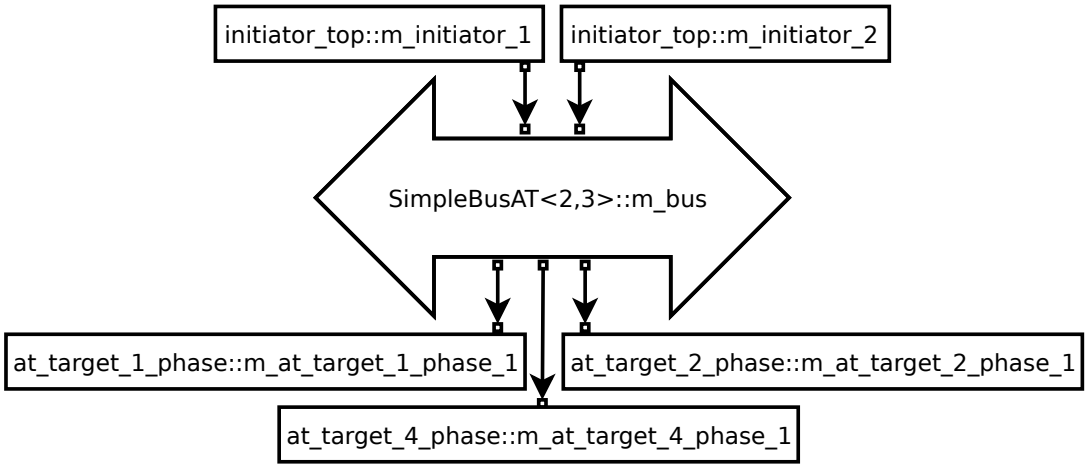


Figure 3.23 – Exemple at_mixed_targets_example

Les informations que notre outil est en mesure de retrouver peuvent se résumer par le graphe de la figure 3.24. Il est la superposition des informations récupérées par l'analyse d'héritage (liens en pointillés) et de la cartographie obtenue par recherche récursive à l'exécution après la phase d'élaboration (lien en trait plein). Le design analysé contient des objets de types SimpleBusAT$< 2, 3 >$, at_target_1_phase, at_target_2_phase, at_target_4_phase et initiator_top. Ceux-ci dérivent de sc_module et sont donc des composants. Ils contiennent différents types de port en tant qu'attributs.

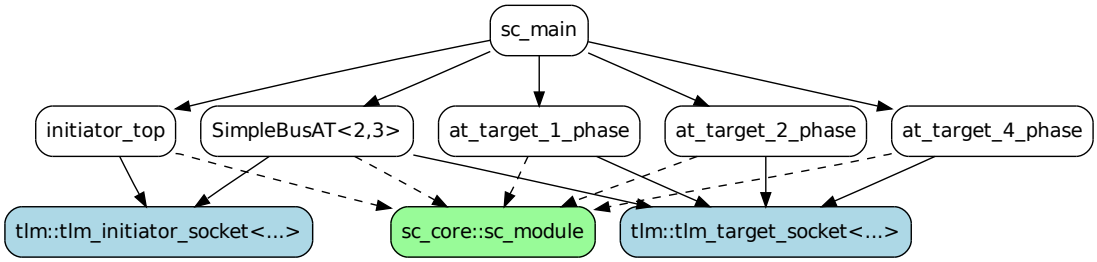


Figure 3.24 – Graphe de l'exemple at_mixed_targets_example

3.5.3 Librairie SoClib

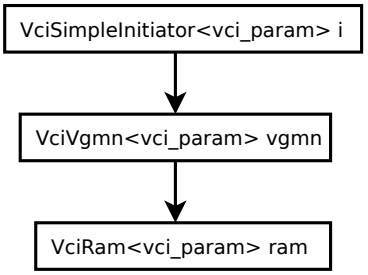


Figure 3.25 – Design extrait de la SoCLib

SoCLib[7] est une plate-forme de développement pour le prototypage virtuel des systèmes sur puce. Le projet a commencé en tant que projet ANR. Il est désormais maintenu au LIP6. Le noyau de la plate-forme est une bibliothèque de modèles de simulation SystemC de composants virtuels. La principale préoccupation est l'interopérabilité entre les différentes IPs disponibles. Tous les modèles de simulation sont écrits en SystemC, et peuvent être simulés avec l'environnement de simulation SystemC standard. Deux types de modèles sont disponibles pour chaque IP, le niveau CABA (*Cycle Accurate / Bit Accurate*) et le TLM-DT (*Transaction Level Modeling with Distributed Time*). Tous les modèles de simulation et les outils sont distribués en tant que logiciels libres.

Cette bibliothèque ayant été développée avant l'arrivée officielle de la librairie TLM, elle définit ses propres connexions TLM. L'exemple analysé (figure 3.25), se compose d'un composant maître communiquant avec un composant esclave par l'intermédiaire d'un médium de communication. Le graphe de la figure 3.26 montre le design réel. SCiPX n'a cependant pas pu récupérer les liens colorés en rouge. La raison est analysée ci-dessous.

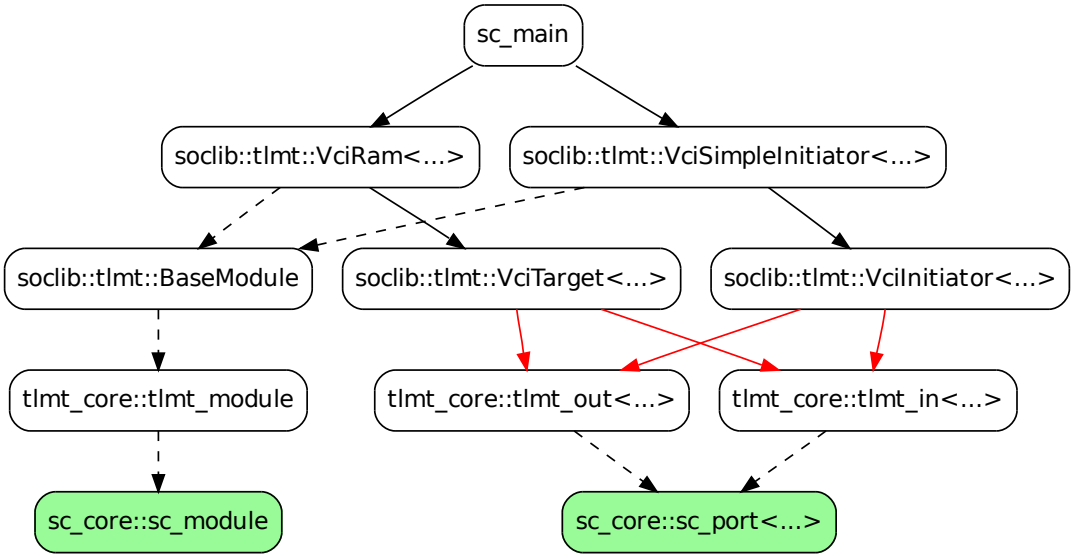


Figure 3.26 – Graphe de contenance

Les composants, bien qu'encapsulés dans divers namespaces propres à la SoCLib, dérivent tous de sc_core::sc_module. La reconnaissance des composants de la SoClib est donc assurée grâce à la consultation du graphe l'héritage d'un type. Pour les interfaces maîtres, des soclib::tlmt::VciInitia -tor< ... > sont utilisés. Ils redéfinissent un équivalent de tlm::tlm_initiator_socket sans en hériter. Ils contiennent de manière privée un tlmt_core::tlmt_out et un tlmt_core::tlmt_in et ceux-ci dérivent de sc_port. Cependant, la contenance privé rend impossible l'accès aux objets de type sc_port par notre outil.

Nous avons renoncé à étendre SCiPX pour résoudre ce problème car l'interface utilisée n'est pas standard. Cependant, une solution est envisageable mais demande de modifier le code source de la librairie (rendre public les liens en rouge).

3.5.4 Librairie GreenSoC

GreenSocs greensocweb est une plate-forme de développement dont le but est d'accélérer l'écriture des modèles et des outils. Il comporte des blocs IPs simples servant de briques de base à l'élaboration de système plus complexe. Dans l'exemple considéré (figure 3.27), le composant maître m est connecté directement à un composant esclave s. Comme pour l'exemple précédent, des sockets spécifiques sont utilisés.

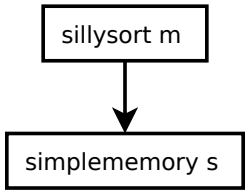


Figure 3.27 – Design fourni par GreenSocs

Les informations que SCiPX est capable de récupérer sont décrites dans le graphe de la figure 3.28. Les composants dérivent publiquement d'un sc_core::sc_module. D'un point de vue des connexions utilisées, les composants contiennent publiquement des sc_core::sc_port facilement accessible par SCiPX grâce à la cartographie récursive des types qu'il construit. Cependant, l'essence même de la connexion TLM est perdue. SCiPX récupère ainsi les liens qui existent entre les composants mais le port TLM n'apparaît pas dans le modèle IP-Xact. Il génère alors autant de port que de sc_port possédés par GenericInitiatorBlockingAPI< ... > et GenericTargetBlockingAPI< ... >.

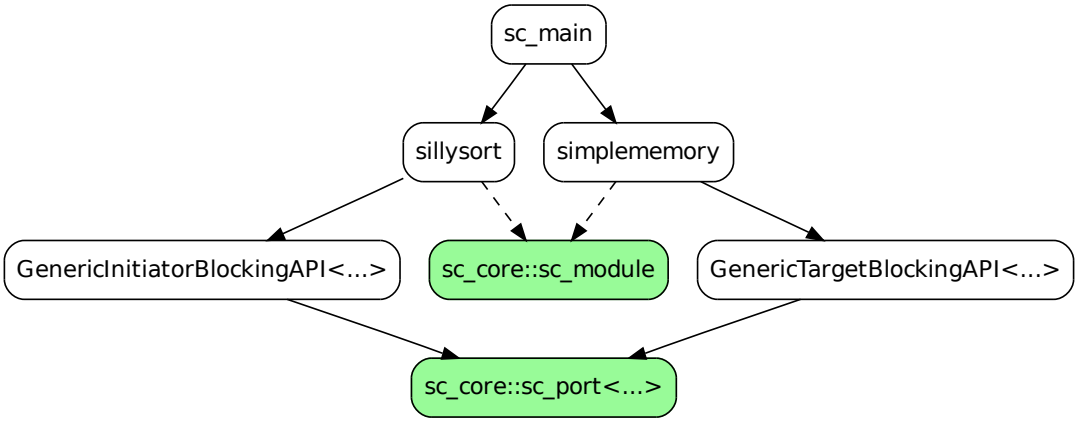


Figure 3.28 – Graphe de l'exemple GreenSoCs

3.5.5 Librairie AMBA-pv

Cette librairie est disponible gratuitement sur le site de ARM armweb. Elle implémente les différentes variantes du protocole AMBA ambaweb (*i.e.,*axi, ahb et apb) à un niveau *programmer view*. L'exemple étudié est celui de la figure 3.29. Il représente un composant TLM2_M[0] connecté directement en tant que maître à un composant AHB_M[0] par des sockets TLM officiels. Ce

dernier est lui-même connecté à deux composants esclaves AHB_S[i] au travers d'un bus AHB_Bus. Ces dernières connexions utilisent des sockets représentant des connexions de type AMBA au niveau TLM. Elles utilisent en interne, comme briques de bases, un type plus récent de connexions définit dans la librairie GreenSocs (basée sur la librairie officielle TLM).

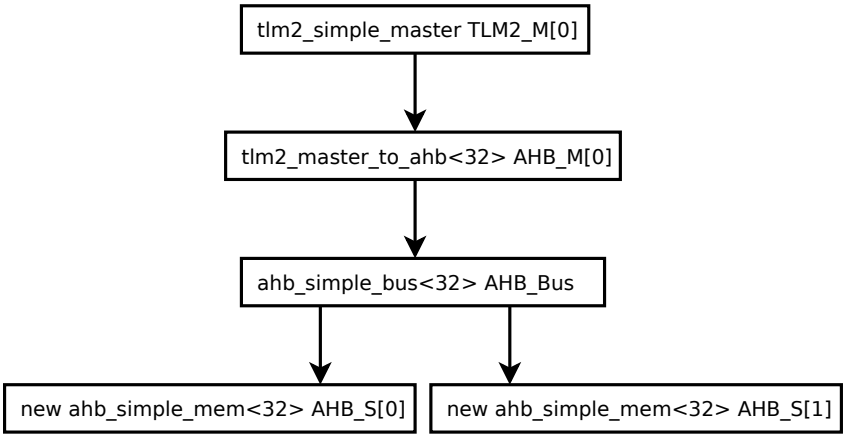


Figure 3.29 – Design issue de la librairie AMBA

Les deux librairies utilisées étant supportées par notre outil, le traitement de cet exemple conduit à une extraction complète du modèle (figure 3.30).

3.5.6 Bilan

Le tableau 3.1 dresse le bilan des types de design que notre outil est en mesure d'extraire et ceux sur lesquels il échoue. Ce tableau différencie deux types de programme TLM, l'un officiel suivant la norme et les versions non standard mais de ce niveau de modélisation. Le symbole ✔signifie que la détection et d'extraction est prise en compte par SCiPX. Le symbole X indique ce qui ne peut être reconnu sans modification du code source de la bibliothèque en question.

	RTL	TLM (official)	TLM (handmade)
SystemC 2.2	✔	N/A	X
SoCLib	✔	N/A	X
GreenSocs	✔	✔	✔
TLM 2.0	✔	✔	N/A
AMBA-pv	✔	✔	N/A

Table 3.1 – Détection des différentes librairies

Notre but était de produire un modèle structurel à partir du plus grand nombre de librairies

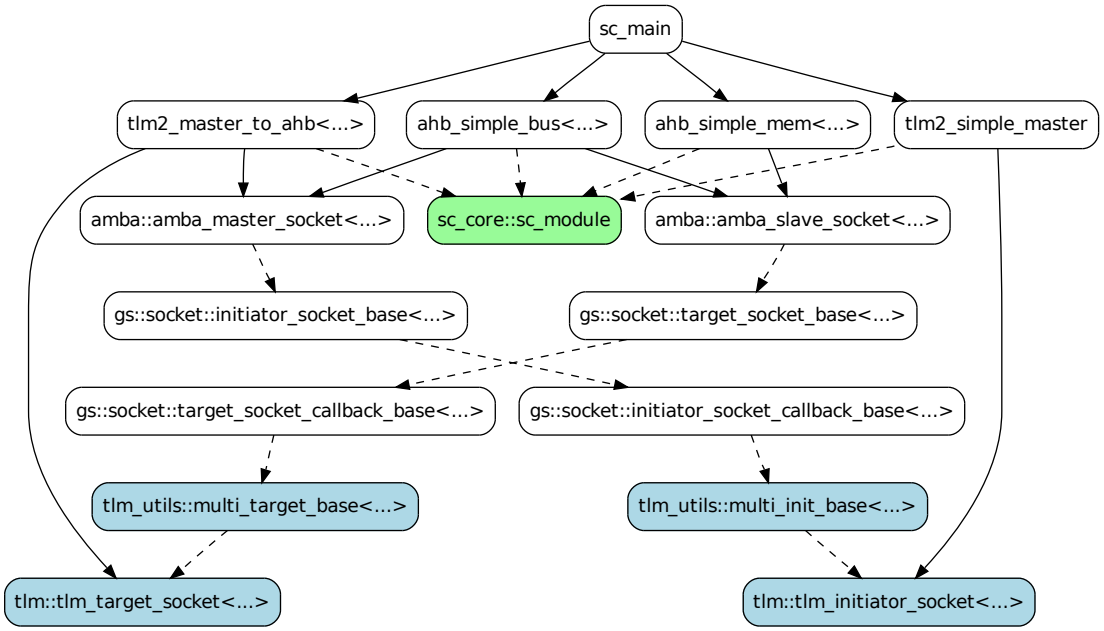


Figure 3.30 – graphe de l'exemple Amba-pv

possibles afin de faciliter la réutilisation de blocs d'IP déjà existants. Cependant, cette étude a mis en évidence, divers problèmes rendant la tâche ardue voire même impossible dans certains cas. Les limitations révélées peuvent être séparées en deux catégories.

La première catégorie étant bien évidemment les limitations de notre outil de par les techniques d'extractions utilisées alliées à la non réflexivité du langage C++. SCiPX, bien que conçu pour être le plus général possible, reste limité. La macro offsetof() ne peut être utilisée que sur des objets publiquement accessibles rendant impossible l'analyse de type restreignant l'accès à leurs attributs. La cartographie récursive d'un composant est, elle, coûteuse en temps et restreinte par la macro offsetof(). L'analyse de type que nous avons mise en place se voit limitée par les déclarations dynamiques. Une déclaration de pointeurs est analysable tant que celle-ci ne dépasse pas le pointage simple (un objet pointé ou vecteur d'objets).

Toutes ces limitations imposent donc des règles de style de codage appropriées afin que notre outil soit en mesure d'extraire un design. Cependant, nous considérons ce style de codage assez général pour ne pas trop contraindre un designer.

La deuxième catégorie de limitation est plus conceptuelle au sens où les informations de structure exprimables en SystemC peuvent être capturées dans le standard IP-Xact mais non l'inverse. Certains concepts présents en IP-Xact ne trouvent pas de mécanismes permettant

leur identification automatique dans la librairie SystemC. Par exemple, il est question dans IP-Xact de décrire les différents registres définis dans un composant, ceux-ci mêmes utilisés afin de piloter/configurer ou d'interroger l'état du composant. En SystemC, aucun mécanisme standard n'est prévu et le designer utilise la majeure partie du temps des types primitifs tels que les unsigned int. Cependant, cette utilisation des types primitifs rend impossible la distinction entre une variable temporaire et un véritable registre au sens IP-Xact que l'on souhaiterait capturer dans un design. Une solution permettant de combler ces limitations est la réécriture de parties de la librairie SystemC. Il faut ajouter les mécanismes permettant de contenir les informations manquantes à l'exécution. Il faut de plus, s'assurer qu'un designer fournira ces informations lors de la conception de la même manière que celui-ci doit renseigner le nom d'un composant ou encore la connectivité des Ports. Pour les concepts manquants, des extensions simples peuvent être facilement implémentées par simple héritage à la classe *sc_object*, assurant par là-même de conserver une trace d'un type durant l'exécution du programme SystemC. Pour plus d'informations le lecteur est convié à consulter à l'annexe B.

Chapitre 4

Modélisation de protocoles et CCSL

Sommaire

4.1 Introduction

La modélisation d'un système par un MoCC apporte une séparation claire des calculs et des communications. Un système peut être représenté par son architecture qui se compose d'unités de calcul et les interconnexions entre ces unités. Dans le chapitre précédent, nous nous sommes intéressé à l'extraction des aspects structurels dans le cas bien particulier de code C^{++} SystemC. Nous nous intéressons maintenant à l'aspect *interactions* entre ces différents blocs. Les communications ou interactions entre ces unités de calcul suivent souvent un ensemble bien établi de règles, connu sous le nom de "protocole". La librairie TLM permet de considérer à différents niveaux d'abstraction ces communications. Elles sont alors bien plus complexes que de simples passages de valeurs sur un fil électrique mais néanmoins représentées par un lien unique de dominance (maître/esclave). Généralement, différentes entités sont mises en concurrence pour l'accès au média de communication. Le simulateur SystemC, par sa nature séquentielle dans le traitement de sa file d'événement, présente des difficultés pour la modélisation d'interactions conccurentes (cf. annexe C). C'est pourquoi nous nous tournons, dans ce chapitre, vers une approche alternative. Nous explorerons, dans la section 4.2, la modélisation de protocoles à l'aide du langage CCSL en nous focalisant sur l'aspect contrôle. Nous partirons d'une vue très abstraite de la communication (lien unique de dominance) et la raffinerons jusqu'à obtenir les détails suffisants pour la modélisation abstraite d'un protocole industriel de type AMBA. Nous mettrons alors en évidence les règles nécessaires à la résolution de conflit (*i.e.*,priorités) lorsque plusieurs maîtres tentent d'accéder à un même bus de communication. Puis, fort de cette expérience de modélisation, nous proposerons, dans la section 4.3, une approche alternative à la résolution de conflit par l'ajout d'un mécanisme de prise en compte de ces priorités durant la simulation d'une spécification CCSL.

4.2 Modélisation de protocoles en CCSL

Dans cette section, nous utilisons les mécanismes et représentations issus de l'IDM (plus particulièrement issus d'UML et de MARTE) afin de modéliser de manière abstraite un protocole AMBA AHB largement répandu dans le domaine des systèmes embarqués. Nous commencerons par identifier les informations de contrôle nécessaires dans le cas simple où seulement deux composants (un maître et un esclave) communiquent par l'intermédiaire d'une connexion point à point. Puis nous complexifierons les exemples, jusqu'à considérer des configurations plus proches des systèmes embarqués réels. Nous nous intéresserons alors à la problématique qu'est l'accès à une ressource de communication partagée entre différents acteurs. Nous dégagerons alors des règles génériques permettant de spécifier le comportement de tels acteurs lors de l'apparition de conflit à l'aide du langage de spécification CCSL.

4.2.1 Transactions et diagrammes de séquence

En UML, on utilise généralement des diagrammes de séquence afin de décrire des interactions entre différents objets. Chaque ligne verticale représente la *ligne de vie* de l'objet identifié au sommet de la ligne. Le temps s'écoule vers le bas. Les échanges d'information (ou messages) sont visualisés par des flèches entre lignes de vie (figure 4.1). Sur ces diagrammes, on peut également identifier les événements liés à l'envoi et à la réception de messages. Lors d'une interaction, il peut y avoir plusieurs occurrences du même événement (figure 4.2). En UML on peut les nommer par la notation @<occurrence de l'événement>.

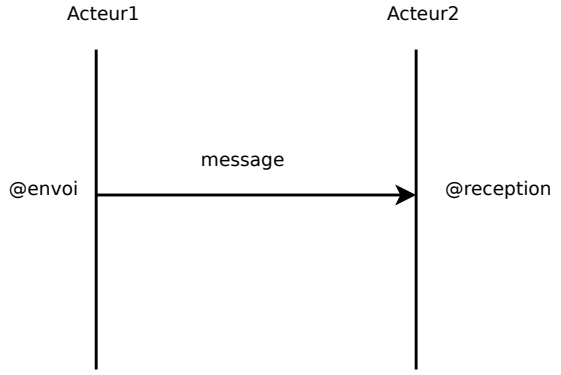


Figure 4.1 – Interaction élémentaire

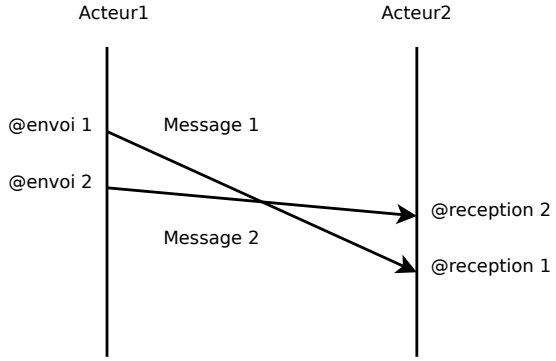


Figure 4.2 – Interactions multiples

Lorsqu'on utilise le profil UML-MARTE, on peut associer une horloge logique à un événement et par conséquent associer les occurrences de cet événement aux instants de cette horloge. Des contraintes entre les occurrences d'événements peuvent alors être spécifiées à l'aide de contraintes d'horloges. Par exemple, le fait que l'envoi d'un message doive précéder sa réception s'exprime par la relation 4.1 qui utilise la précédence faible (section 2.5.2).

$$envoi \boxed{\preccurlyeq} réception \tag{4.1}$$

Il est clair que cette relation n'apporte rien de plus que ce qui est exprimé par la flèche associée au message dans le diagramme de séquence. En revanche, les contraintes d'horloge s'avèrent très utiles quand on veut exprimer des contraintes *entre* messages ou entre occurrences d'événements. C'est ce que nous allons analyser à présent. Au préalable, notons que dans la suite on s'intéresse plus particulièrement aux *transactions* qui sont des messages particuliers. Les événements d'envoi et de réception sont alors nommés begin et end.

4.2.2 Transactions concurrentes

Un inconvénient majeur des diagrammes de séquence est qu'ils ne représentent que des évolutions particulières. UML 2 les a enrichi avec les *fragments* qui permettent de représenter

plusieurs évolutions (boucles, conditionnelles, références) sur un même diagramme. La lisibilité de ces diagrammes n'est pas toujours aisée. Nous préférons nous en tenir aux diagrammes de séquence simples auxquels on adjoint des contraintes, essentiellement des contraintes d'horloges.

La figure 4.3 modélise deux échanges d'un message de même type. Le diagramme de droite représente le cas où le deuxième message "double" le premier.

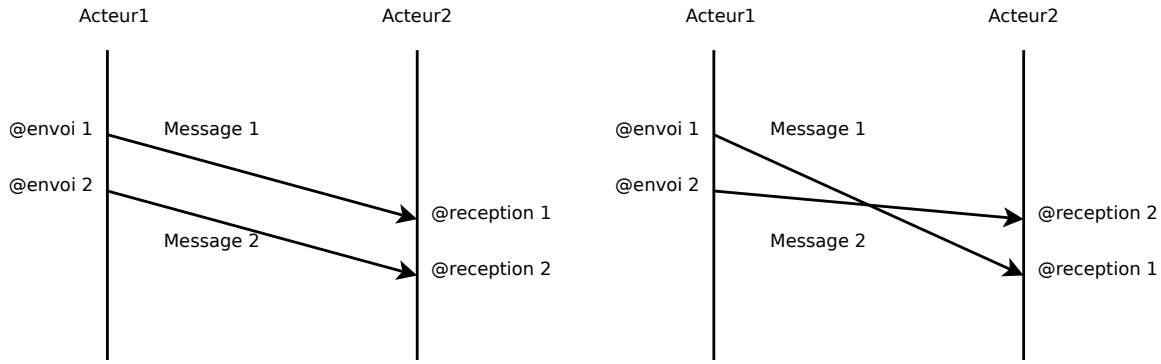


Figure 4.3 – Interaction avec deux messages de même type

Lorsque les messages modélisent des transactions, l'inversion de l'ordre de terminaison peut être problématique. La relation 4.1 ne contraint pas l'ordre des réceptions ; des contraintes supplémentaires doivent être imposées si on veut interdire ou borner les inversions de réception. Il est cependant possible de solutionner cela à l'aide de VSL (Value Specification Language) qui fait partie du profile MARTE et permet d'exprimer des contraintes entre les dates de réception. Les expressions font alors intervenir explicitement des indices associés aux différentes occurrences. On fait alors implicitement référence à un *temps chronométrique*. Dans un souci de généralité, nous préférons des modélisations en *temps logique* et la spécification d'invariants temporels sous forme de contraintes d'horloges.

Si on veut que l'ordre des réceptions respecte l'ordre des émissions, le plus simple est d'imposer une stricte séquentialité : un nouveau message ne peut être envoyé qu'après la réception du message précédent. Ceci s'exprime aisément en CCSL avec la relation d'alternance (section 2.5.2) : *envoi* $\boxed{\simeq}$ *réception*. Une nouvelle occurrence $k+1$ de l'événement *envoi* est alors interdite par cette contrainte tant que l'occurrence k de l'événement *réception* ne s'est pas produite. Cette solution a l'inconvénient d'interdire le chevauchement de transactions et en particulier des fonctionnements en mode *pipe-line* (à gauche de la figure 4.3).

Cependant, le recours à $n-1$ horloges logiques auxiliaires h_k, avec h_k $\boxed{\simeq}$ h_{k+1} permet de spécifier un comportement pipe-line à n étages (figure 4.4). La transaction entre l'acteur 1 et l'acteur N est alors décomposée en transactions (flèches noires) entre les acteurs de 1 à N. Chaque transaction reçue par un acteur déclenche l'envoie d'une transaction à l'acteur suivant jusqu'à atteindre l'acteur N. L'occurrence k de l'événement *réception* de l'acteur N en bout

de chaîne correspond bien à l'occurrence k de l'événement *envoi* de l'acteur 1. Cependant, l'occurrence $k+1$ de l'événement *envoi* de l'acteur 1 n'est plus interdite par l'occurrence k de l'événement *réception* de l'acteur N mais par celle de l'acteur 2 (relations de précédence en pointillés orangés). Notons que ces horloges auxiliaires ne correspondent pas nécessairement à des événements existant réellement dans le système modélisé.

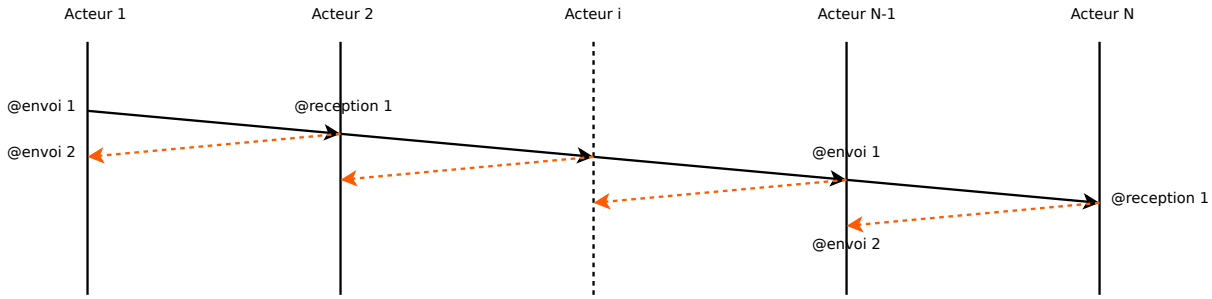


Figure 4.4 – Alternance avec plusieurs intermédiaires

CCSL permet également de spécifier des comportements temporels autorisant un nombre borné d'inversions. De tels comportements sont décrits dans une étude sur un processeur d'images spécifié en CCSL[?]. Notre objectif étant plus limité, nous allons étudier plus spécialement les modélisations d'interactions "Maîtres-Esclaves".

4.2.3 Système 1 maître-1 esclave

Nous commençons par l'étude de la configuration la plus simple : un seul maître et un seul esclave (figure 4.5).

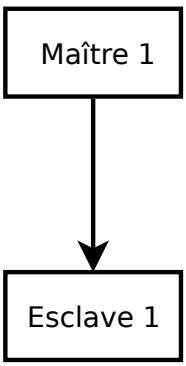


Figure 4.5 – Un maître-un esclave (point à point)

Au niveau le plus abstrait une transaction, considérée comme atomique, peut être représentée par un simple passage de message comme dans les figures 4.1 et 4.2. Nous avons discuté dans

la sous-section précédente du problème de respecter ou non l'ordre des envois (*begin* pour une transaction) au niveau des réceptions (*end* pour une transaction). Dans tous les cas la relation de précédence (equation 4.1) impose des évolutions temporelles telles que celles présentées dans la figure 4.6. Cette figure représente une évolution possible de cette équation sous la forme d'un chronogramme. Les flèches en traits interrompus indiquent chaque précédence entre instants. Cette relation ne reflète pas nécessairement la relation begin-end (cas où les réceptions ne sont pas dans l'ordre des émissions).

Le style de codage *Loosely-Timed* (LT) de SystemC TLM peut très bien se contenter de cette modélisation. En effet, le style LT n'est pas une représentation fidèle des évolutions temporelles. Un de ses avantages est de permettre au maître de simuler *plusieurs* transactions sans commutation de contexte lorsque son process s'exécute. Il réalise alors une transaction par un simple appel de fonction. La transaction se termine lorsque la fonction retourne son résultat au maître. Il peut en résulter une distorsion temporelle (*warp*) acceptable dans la mesure où elle reste limitée.

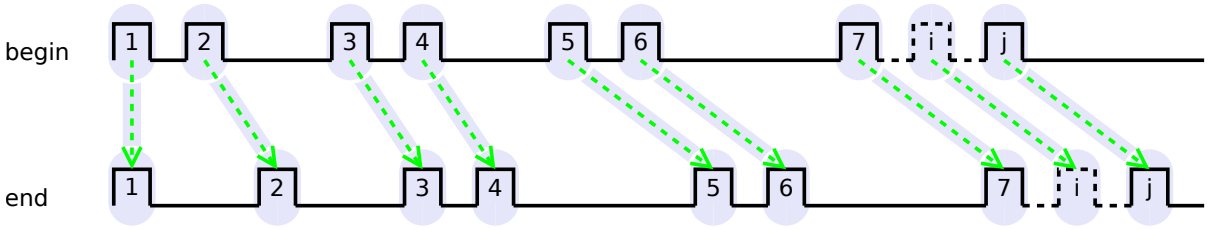


Figure 4.6 – Début et fin de transactions

Un premier *raffinement* d'une transaction consiste à distinguer une *phase de réquête* et une *phase de réponse* (figure 4.7). Cette modélisation est plus réaliste vis à vis des comportements temporels car elle permet de prendre en compte la disponibilité des acteurs. C'est ce que permet le style de codage *Approximately-Timed* (AT) de SystemC TLM.

Pour exprimer ce modèle en CCSL, il convient d'utiliser une contrainte de précédence faible entre la réception de la requête et l'envoi de la réponse. Il faut alors lui adjoindre les alternances entre les deux sous-messages de la transaction (requête et réponse) (relation 4.2).

$$\begin{array}{rcl} begin\_req & \boxed{\simeq} & end\_req \\ end\_req & \boxed{\preccurlyeq} & begin\_resp \\ begin\_resp & \boxed{\simeq} & end\_resp \end{array} \tag{4.2}$$

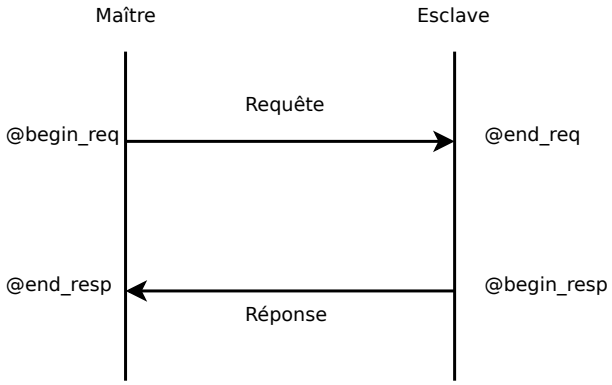


Figure 4.7 – Transaction avec requête et réponse

Pour les bus les plus simples les transactions ne se chevauchent pas : on ne peut commencer une nouvelle transaction qu'après avoir terminé la précédente. La simple relation d'alternance 4.3 exprime alors ce comportement. On retombe alors sur la version de la transaction abstraite détaillée précédemment.

$$begin\_req \boxed{\simeq} end\_resp \tag{4.3}$$

Un bus comme le bus AMBA permet un chevauchement limité des transactions. Les contraintes CCSL 4.2 spécifient un comportement dans lequel deux transactions au plus peuvent être simultanément en cours. La figure 4.8 représente une évolution possible. Chaque flèche en traits interrompus représente ici encore des relations de précédences entre instants d'horloges. Les flèches rouges illustrent ici les précédences entre instants issues des relations d'alternances spécifiées dans les équations 4.2. Les vertes représentent les relations de précédences dues à la spécification explicite de la précédence.

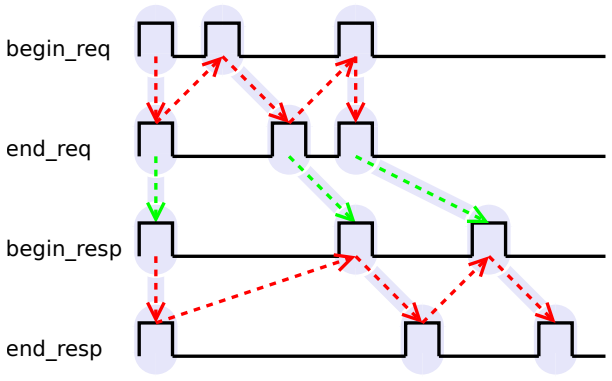


Figure 4.8 – Transactions sur bus permettant deux transactions en cours

En CCSL il est également possible de décrire une transaction pour bus AMBA au niveau *cycle-*

accurate[?]. Toutefois ces modélisations sortent du cadre “haut-niveau d’abstraction” étudié dans cette thèse.

4.2.4 Système 1 maître-2 esclaves

Nous avons considéré jusqu’à présent que notre système ne comportait qu’un seul maître et un seul esclave afin de détailler l’essence même des raffinements possibles pour une transaction. Cependant, cette configuration limite les comportements permis évitant certains problèmes dus à la concurrence. Nous considérons maintenant un système possédant un maître et deux esclaves (figure 4.9). Néanmoins, même si la figure s’apparente à une diffusion de la part du maître, nous considèrerons qu’un seul esclave peut être sollicité à la fois. Il nous faut alors distinguer les transactions telles qu’elles sont vues par les différents acteurs du système.

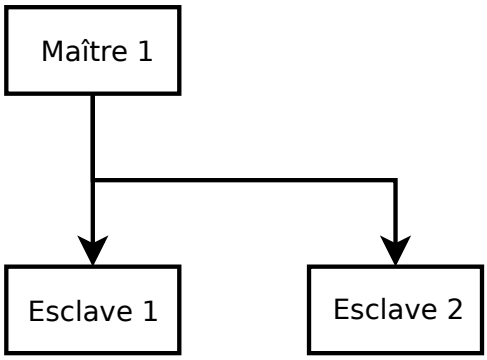


Figure 4.9 – Un maître-deux esclaves (démux)

Les contraintes exprimées par les équations 4.2 restent nécessaires mais ne représentent que le flot global de transactions. Il faut introduire des horloges auxiliaires aux interfaces de nos différents composants. Pour le cas du seul maître, ces horloges ne sont qu’une simple copie des horloges globales. C’est pourquoi nous les assimilerons, dans ce cas bien particulier, aux horloges globales du média de transport. Pour les esclaves, les horloges de leurs interfaces résultent du démultiplexage des horloges globales. A priori, le système semble pouvoir s’exprimer par les contraintes d’horloges détaillées ci-après.

$$
\begin{aligned}
Slave1\_begin\_req &\ \boxed{\simeq}\ Slave1\_end\_req \\
Slave1\_end\_req &\ \boxed{\preccurlyeq}\ Slave1\_begin\_resp \\
Slave1\_begin\_resp &\ \boxed{\simeq}\ Slave1\_end\_resp \\
Slave2\_begin\_req &\ \boxed{\simeq}\ Slave2\_end\_req \\
Slave2\_end\_req &\ \boxed{\preccurlyeq}\ Slave2\_begin\_resp \\
Slave2\_begin\_resp &\ \boxed{\simeq}\ Slave2\_end\_resp
\end{aligned}
\tag{4.4}
$$

Les six contraintes décrites par les équations 4.4 représentent le flux vu des esclaves numéro un et deux indépendamment du flux global.

$$
\begin{aligned}
begin\_req &\ \boxed{=}\ Slave1\_begin\_req + Slave2\_begin\_req \\
end\_req &\ \boxed{=}\ Slave1\_end\_req + Slave2\_end\_req \\
begin\_resp &\ \boxed{=}\ Slave1\_begin\_resp + Slave2\_begin\_resp \\
end\_resp &\ \boxed{=}\ Slave1\_end\_resp + Slave2\_end\_resp
\end{aligned}
\tag{4.5}
$$

Les quatre contraintes décrites par les équations 4.5 lient le flux global aux flux des esclaves.

$$
\begin{aligned}
Slave1\_begin\_req &\ \boxed{\#}\ Slave2\_begin\_req \\
Slave1\_begin\_resp &\ \boxed{\#}\ Slave2\_begin\_resp
\end{aligned}
\tag{4.6}
$$

Les contraintes des équations 4.6 assurent l'accès exclusif aux deux esclaves.

Nous considérons, ici encore, une relation de précédence comme comportement émergent mais faisons l'hypothèse que l'ordre entre les occurrences d'événements end_req et begin_resp est conservé. Ainsi, nous évitons de considérer les comportements de la trace décrite à la figure 4.10, dans laquelle une inversion d'ordre dans la réception des réponses provenant des esclaves (Slave1_begin_resp et Slave2_begin_resp) peut apparaître bien que l'ordre de chaque esclave soit respecté indépendamment. On remarque dans cette figure que la deuxième transaction débutée par le maître (deuxième occurence de begin_req) est bien prise en compte par

l'esclave 1 (leur deuxième occurrence de begin_req et end_req sont synchrones). Cependant, le début de réponse que le maître obtient (deuxième occurence de begin_resp) ne coïncide pas avec le début de réponse initié par l'esclave 1 mais coïncide avec le début de la troisième transaction qu'il initie à destination de l'esclave 2.

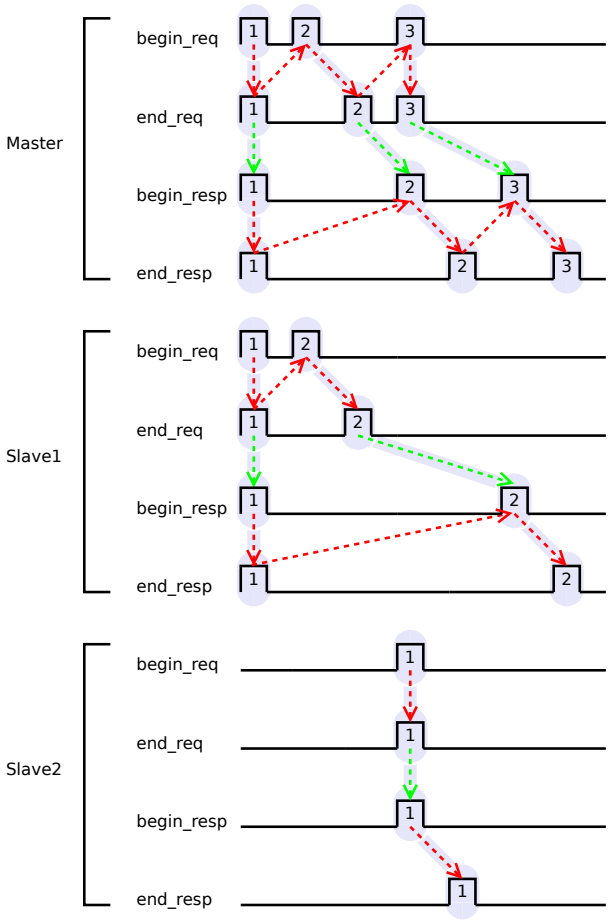


Figure 4.10 – Inversion d'ordre de terminaison de transactions

Quoi qu'il en soit, nous basant sur les spécifications du protocole AMBA ambaweb, nous nous sommes aperçus que ce problème était résolu par le forçage d'un comportement bien particulier qui contrôle l'ordre de retour. A l'analyse de la spécification de ce protocole, il apparaît d'une part qu'un maître ne peut débuter une nouvelle transaction que lorsque la précédente a été prise en compte. D'autre part la réponse d'un esclave n'est valide qu'à partir d'un certain moment qu'il notifie au maître. Ces deux phénomènes ont un point en commun : ils sont notifiés par le même événement (à savoir le signal *hready*). Lorsque cet événement se produit, cela signifie à la fois que la requête précédente a bien été prise en compte (permettant de débuter une nouvelle) et que la réponse est disponible sur le média de communication (garantissant que la réponse est

bien celle de la requête précédente). D'un point de vue relation, cela revient à considérer une forme plus rudimentaire de la transaction (décrite par les équations 4.7). Nous considérerons donc par la suite une modélisation de la transaction composée de trois horloges contraintes par les relations décrites par les équations 4.7

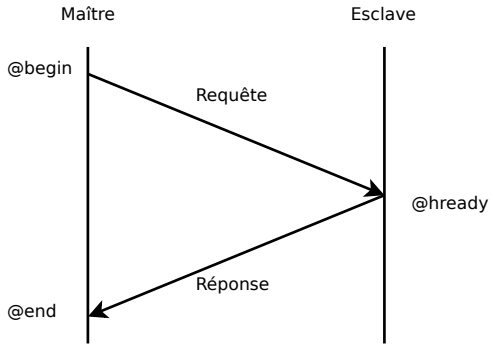


Figure 4.11 – Transaction simplifiée

$$\begin{array}{r} begin \boxed{\simeq} hready \\ hready \boxed{\simeq} end \end{array} \tag{4.7}$$

où begin (resp. end) est l'équivalent de l'horloge begin_req (resp. end_resp) initiale et hready l'horloge forçant la synchronisation entre end_req et begin_resp (et interdisant par la même occasion les inversions d'ordre de retour des réponses). Ainsi, une transaction s'en retrouve simplifiée de la manière décrite dans la figure 4.11. Comparativement à la version "AT", elle ne contient plus que deux phases. Pour un système équivalent à la figure 4.9, une trace d'exécution est donnée dans figure 4.12. L'ordre de passage des messages est forcé (à l'inverse du comportement de la figure 4.10). Le problème d'inversion ne peut plus apparaitre, les relations de précédences héritées de la relation d'alternance l'interdisant.

Une généralisation des contraintes à un ensemble fini d'esclaves est donnée dans l'annexe C.

4.2.5 Système 2 maîtres-2 esclaves

Nous considérons maintenant un système comportant plusieurs maîtres (figure 4.13). Cette fois-ci il est alors nécessaire de distinguer les événements aux interfaces des différents maîtres alors que les horloges logiques concernant événements aux interfaces des esclaves ne changent pas. Afin de s'affranchir des problèmes liés à la concurrence entre les deux maîtres, nous faisons provisoirement l'hypothèse qu'ils tentent d'accéder à un esclave de manière exclusive. Il en résulte les contraintes décrites par les équations 4.8 pour chaque interface des maîtres et les

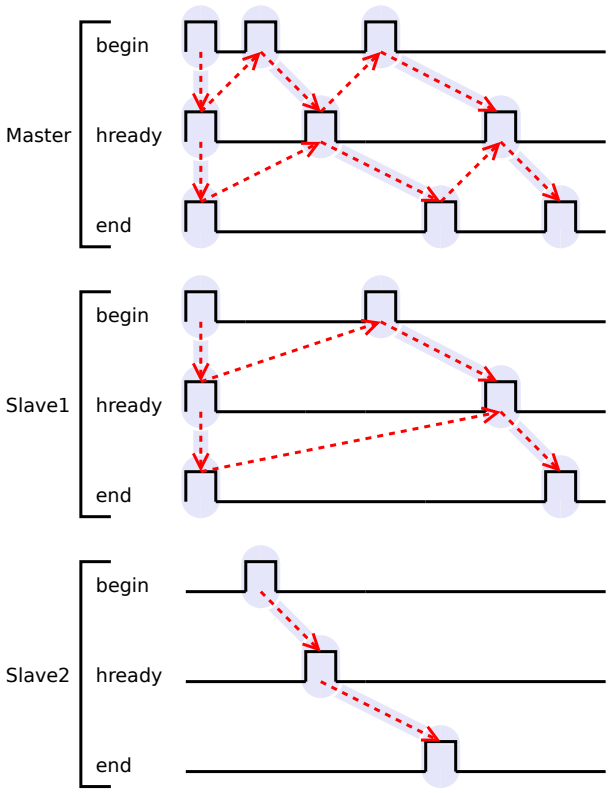


Figure 4.12 – Trace de transaction simplifiée

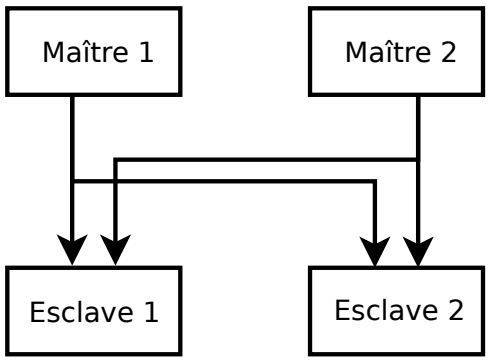


Figure 4.13 – Deux maîtres-deux esclaves

contraintes (équations 4.9) les liant (comme pour les cas des esclaves) au flux global ainsi que les contraintes d'exclusion (équations 4.10).

$$\begin{aligned} Master1\_begin &\boxed{\simeq} Master1\_hready \\ Master1\_hready &\boxed{\simeq} Master1\_end \\ Master2\_begin &\boxed{\simeq} Master2\_hready \\ Master2\_hready &\boxed{\simeq} Master2\_end \end{aligned} \tag{4.8}$$

$$\begin{aligned} begin &\boxed{=} Master1\_begin + Master2\_begin \\ hready &\boxed{=} Master1\_hready + Master2\_hready \\ end &\boxed{=} Master1\_end + Master2\_end \end{aligned} \tag{4.9}$$

$$\begin{aligned} Master1\_begin &\boxed{\#} Master2\_begin \\ Master1\_hready &\boxed{\#} Master2\_hready \end{aligned} \tag{4.10}$$

Grâce à ces seules contraintes, on est alors en mesure de représenter différents passages de messages entre différents maîtres et esclaves. Le premier maître ayant fait une demande étant le premier servi tout en n'interdisant pas une demande de l'autre maître (bien que celle-ci soit retardée d'autant que le nécessite la réponse au précédent). La figure 4.14 donne un exemple d'évolution possible caractérisée par de telles contraintes. On peut y distinguer les transactions provenant du premier maître (en bleu clair) de celles provenant du deuxième maître (en vert). Le begin du deuxième maître ne peut se produire que lorsque le begin global le peut. La deuxième occurrence du hready globale contraint donc implicitement la première occurrence du begin du deuxième maître 2 en contraignant la troisième occurrence du begin global. L'inversion des transaction n'est alors pas permis non plus. Une généralisation des contraintes à un ensemble fini de maîtres et d'esclaves est présentée dans l'annexe C.

4.2.6 Émulation de priorité statique

Dans les parties précédentes, nous avons fait une hypothèse forte en nous affranchissant de certains problèmes liés aux accès concurrents. Nous considérons ici une vue plus réaliste des contraintes précédentes afin de modéliser au mieux les passages de message sur un média commun de communication. Les contraintes d'exclusion entre maîtres (équation 4.11) sous-entendent soit que les maîtres sont au courant des activités des autres maîtres et adaptent leur

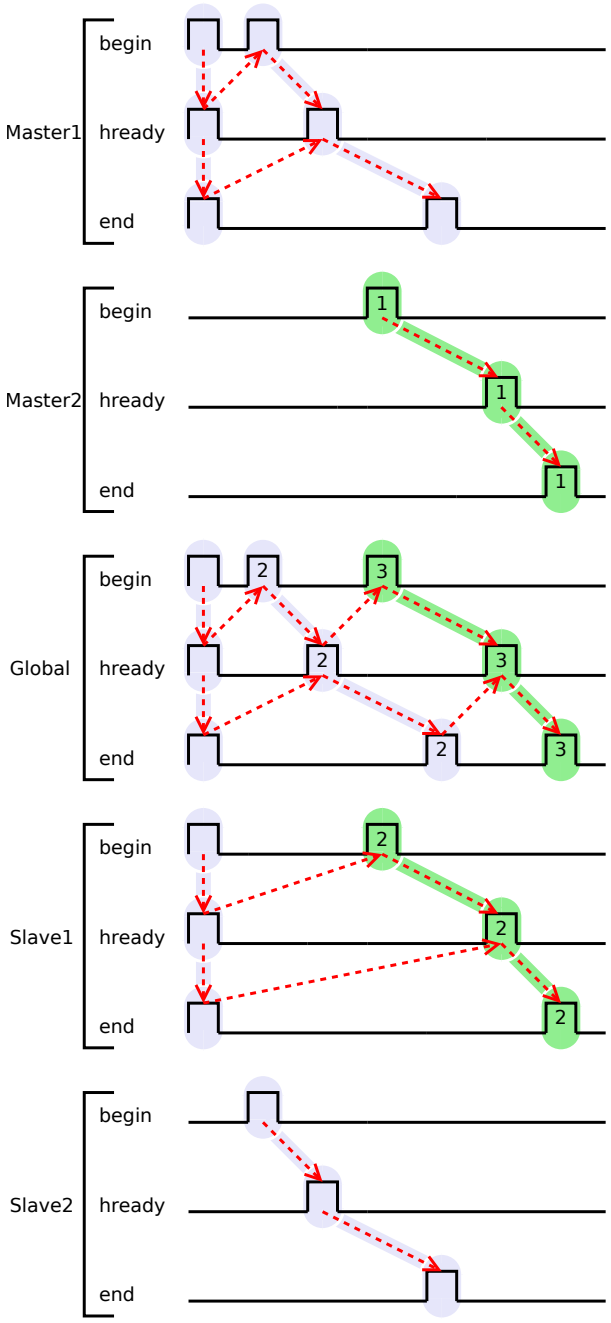


Figure 4.14 – Mélange des transactions

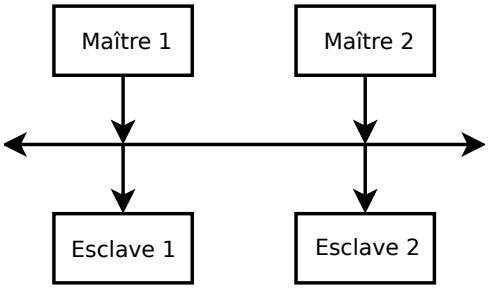


Figure 4.15 – Deux maîtres-deux esclaves (mux/démux)

envois en fonction de l'occupation du média de transport partagé, soit que l'on dispose d'un chef d'orchestre global (ou arbitre) qui autorise un maître à communiquer avec un esclave.

$$\forall i \in M, Master_i\_begin \boxed{\#} \bigcup_{\forall j \in M \setminus i} Master_j\_begin \tag{4.11}$$

Cependant, d'un point de vue local, un maître peut commencer une transaction quand bon lui semble. Cela pose un problème lorsque deux maîtres tentent de débuter une transaction simultanément sur le même média de transport. Ce n'est qu'à ce moment qu'une procédure de résolution de conflits s'engage qui règle le problème par un choix selon certains critères de décision (ou priorités). Nous considérons donc ici, que chaque maître possède une priorité statique unique imposant un ordre entre eux que l'on associera (pour des raisons pratiques) à l'indice de chaque maître de manière décroissante (*i.e.,*plus l'indice sera petit et plus la priorité sera important). Nous devons alors introduire de nouvelles horloges intermédiaires qui permettront de laisser évoluer les horloges $Master_i\_begin$ de chaque maître i pour éventuellement tenter des accès concurrents). L'émulation de la priorité est alors possible si l'on considère qu'un début de transaction peut être placé dans une file d'attente si elle est en conflit avec une autre. Pour spécifier cela, il est nécessaire de séparer explicitement les requêtes d'un maître en plusieurs catégories, celles sans conflit (que l'on post-fixera par CF pour "Conflict Free"), qui se dérouleront alors normalement, celles en conflits qui devront être décalées dans le temps (que l'on post-fixera par C pour "Conflict"), celles qui représenteront la demande décalées (que l'on post-fixera par D pour "Delayed") et qui respecteront d'éventuelles autres demandes plus prioritaires et pour terminer celles qui représenteront les demandes effectives telles qu'elles apparaissent sur le bus (que l'on post-fixera par E pour "Effective"). La figure 4.16 illustre un tel fonctionnement du point de vue du maître d'indice i (sous-entendu ici, il existe des maîtres de plus hautes priorités). La première horloge ("begin effectif global") représente ici l'union de toutes les demandes effectives de plus hautes priorités. Ainsi, la première requête du maître i (en vert) qu'il effectue est considérée comme la plus prioritaire au moment où elle apparaît et doit donc être traitée normalement car aucune requête de plus hautes priorités n'apparaît sur l'horloge des

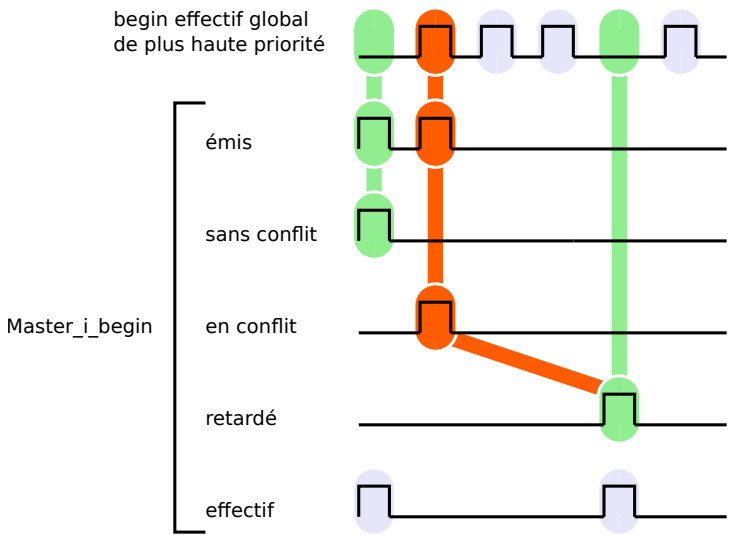


Figure 4.16 – Délai de prise en compte

demandes effectives globale. Elle génère alors l'horloge sans conflit en coïncidence. La seconde requête en rouge est considérée en conflit avec une requête de plus haute priorité et doit donc être différée. Elle génère cette fois-ci l'horloge avec conflit en coïncidence. De plus, une requête équivalente mais translatée dans le temps doit alors être générée de manière à ne pas interférer avec d'autres requêtes de plus hautes priorités (l'horloge "delayed"). Au final, il est alors possible de construire l'horloge effective des requêtes de ce maître comme l'union des requêtes sans conflits et des requêtes différées.

Dans le cas bien particulier où il n'y a que deux maîtres, les relations à spécifier sont simplifiées. Effectivement, le maître de plus haute priorité ne peut être retardé, ses requêtes émises sont alors identiques à ses requêtes effectives. Cela implique aussi qu'il n'aura aucune requête décalée dans le temps. Le maître de priorité plus faible devra tenir compte des requêtes émises (effectives) par le premier pour pouvoir faire passer les siennes. On retrouvera alors le même type de décomposition que celui décrit à la figure 4.16. Un tel comportement est spécifié par les équations 4.12.

$$
\begin{array}{rcl}
Master_2\_begin_{EG} & = & Master_1\_begin \\
Master_2\_begin_{E} & = & Master_2\_begin_{CF} + Master_2\_begin_{D} \\
Master_2\_begin_{EG} & \# & Master_2\_begin_{E} \\
Master_2\_begin & = & Master_2\_begin_{C} + Master_2\_begin_{CF} \\
Master_2\_begin_{C} & \# & Master_2\_begin * Master_2\_begin_{EG} \\
Master_2\_begin_{C} & \# & Master_2\_begin_{CF} \\
Master_2\_begin_{C} & \simeq & Master_2\_begin_{D}
\end{array}
\tag{4.12}
$$

Ici encore, le lecteur pourra trouver la version généralisée des relations à spécifier pour un ensemble fini de maîtres dans l'annexe C.

4.2.7 Bilan

Dans cette section, nous avons modélisé le comportement abstrait d'un protocole de communication standard en nous focalisant sur l'aspect contrôle. Nous avons exploré les possibilités permises par le langage de spécification de contraintes CCSL. Nous avons ainsi poussé notre modélisation jusqu'à considérer des systèmes matériels où la concurrence impose de pouvoir résoudre des conflits lors de courses critiques. Cependant, cette résolution de conflit, bien que possible par l'ajout de contraintes CCSL et d'horloges auxiliaires, alourdit considérablement la spécification originale. Il est, quoi qu'il en soit, possible de générer ces contraintes de manière automatique pour un système figé (ordre total sur la relation de priorité) grâce aux généralisations décrites dans l'annexe C.

4.3 Prise en compte des priorités dans CCSL

Nous détaillons, dans cette partie, une approche différente de la gestion des priorités. Cette notion de priorité, manquante en CCSL, est fortement liée aux systèmes que nous modélisons. La notion d'entrée/sortie présente dans les systèmes matériels est elle aussi absente en CCSL. Un développeur peut vouloir distinguer des horloges comme étant des entrées ou des sorties d'un système, elles n'en restent pas moins liées par des relations CCSL *acausales* (*i.e.,*qu'elles ne distinguent pas les entrées des sorties). Autrement dit, lorsque l'on tente de générer une trace d'exécution conforme à une spécification CCSL, on ne tient pas compte de la direction des flots.

Il en résulte qu'une horloge considérée comme étant une entrée d'un système peut très bien être contrainte à ne pas s'activer si d'autres horloges l'en empêchent. Or, les entrées/sorties d'un composant matériel sont des interfaces spéciales généralement non symétriques que le composant doit subir ou contrôler. Plus généralement, on souhaite privilégier le choix de certaines horloges plutôt que d'autres. Dans cette section, nous expliquons tout d'abord le passage d'une spécification CCSL (représenté par sa formule booléene d'un instant logique) à une représentation du choix des solutions possibles sous la forme d'un graphe particulier. Puis nous définirons une notion de priorité adaptée au langage CCSL et nous l'illustrerons à l'aide d'exemples de complexité croissante. Pour terminer, nous donnerons un algorithme permettant de l'appliquer durant l'exécution d'une spécification CCSL.

4.3.1 Représentation des solutions

Problème à résoudre Pour un système contraint par une spécification CCSL, considéré à un instant logique courant, nous savons caractériser l'ensemble des évolutions autorisées à cet instant : il s'agit de l'ensemble des valuations qui satisfont la formule propositionnelle ϕ issue de la traduction des contraintes CCSL. Imposer une relation de priorité revient à ne retenir qu'un sous-ensemble des solutions précédentes. Notre objectif est de caractériser ce sous-ensemble.

Pour résoudre ce problème, nous proposons de construire un *graphe de décision*. Cette approche nécessite un minimum de notations que nous introduisons maintenant.

4.3.1.1 Notations

Définition 4.1 *Soit $\mathcal{V}$ l'ensemble des variables propositionnelles en bijection avec l'ensemble des horloges logiques définies dans une spécification* CCSL. *Par la suite, nous noterons $\top$ (resp. $\bot$) la valuation à "vrai" (resp. "faux") d'une variable propositionnelle (ou formule portant sur les variables de $\mathcal{V}$).*

Définition 4.2 *Soit $\phi \neq \bot$ une formule propositionnelle sur $\mathcal{V}$. ϕ induit une partition de $\mathcal{V}$ en trois sous-ensembles $\mathcal{V}_{\phi_{F^+}}$, $\mathcal{V}_{\phi_{F^-}}$ et $\mathcal{V}_{\phi_U}$ tels que*

- *$\mathcal{V}_{\phi_{F^+}}$ est l'ensemble des variables de $\mathcal{V}$ **fixées** à vrai pour ϕ, c'est-à-dire que dans toutes les solutions qui satisfont ϕ, v est à vrai ;*
- *$\mathcal{V}_{\phi_{F^-}}$ est l'ensemble des variables de $\mathcal{V}$ **fixées** à faux pour ϕ, c'est-à-dire que dans toutes les solutions qui satisfont ϕ, v est à faux ;*
- *$\mathcal{V}_{\phi_U}$ est l'ensemble des variables de $\mathcal{V}$ **non fixées** pour ϕ, c'est-à-dire que pour tout $v \in \mathcal{V}_{\phi_U}$, il existe des solutions satisfaisant ϕ avec $v =$ faux et des solutions avec $v =$ vrai.*

Remarque 4.1 *Une variable qui n'apparaît pas explicitement dans ϕ est nécessairement dans $\mathcal{V}_{\phi_U}$.*

Dans la suite, nous utilisons souvent la notion de co-facteur. Les co-facteurs permettent entre autre de déterminer aisément si une variable est fixée ou non dans une formule propositionnelle (propriétés 4.1 et 4.2).

Définition 4.3 *(Co-facteur) Soit f une fonction booléenne sur $X = x_1, \cdots, x_n$, on appelle i^e co-facteurs de f les fonctions booléennes $f_{|x_i=\mathsf{faux}}$ et $f_{|x_i=\mathsf{vrai}}$.*

- *$f_{|x_i=\mathsf{faux}}$ est noté $f_{\neg x_i}$ et appelé co-facteur négatif de f pour x_i ;*
- *$f_{|x_i=\mathsf{vrai}}$ est noté f_{x_i} et appelé co-facteur positif de f pour x_i ;*

On peut alors pour tout i de 1 à n écrire la *décomposition de Shannon* sous la forme

$$f(x_1, \cdots, x_i, \cdots, x_n) = (x_i \wedge f_{x_i}) \vee (\neg x_i \wedge f_{\neg x_i}) \tag{4.13}$$

Proposition 4.1 *v appartient à $\mathcal{V}_{\phi_{F^+}}$ si et seulement si $(\phi \neq \bot)$ et $(\phi_{\neg v} = \bot)$.*

La première clause élimine le cas trivial de la fonction $\phi = \bot$ qui ne dépend d'aucune variable. La deuxième clause dit que le co-facteur négatif de ϕ est $\bot$. Cela signifie qu'aucune valuation satisfaisant ϕ n'affecte faux à v. Pour toute valuation satisfaisant ϕ, v est donc forcément à vrai et appartient à $\mathcal{V}_{\phi_{F^+}}$.

Proposition 4.2 *v appartient à $\mathcal{V}_{\phi_{F^-}}$ si et seulement si $(\phi \neq \bot)$ et $(\phi_v = \bot)$.*

Cette proposition est duale de la précédente.

4.3.1.2 Représentation par graphe de décision

Pour une formule propositionnelle ϕ sur $\mathcal{V}$, nous voulons déterminer les valuations de $\mathcal{V}$ qui satisfont ϕ. Ces solutions vont être trouvées par la construction incrémentale d'un graphe de décision. Cette approche s'apparente aux techniques utilisées par les outils SAT. A l'inverse de ces derniers qui tentent de trouver une solution, nous souhaitons considérer la totalité des solutions d'une formule ϕ. De plus, nos décisions vont privilégier les horloges qui *tiquent* dans une réaction car elles ont une influence majeure sur les contraintes de priorité et l'évolution d'un système décrit en CCSL (un instant logique où aucune horloge ne tique ne fait pas évoluer la configuration). La construction se fait donc en ajoutant à un sommet un arc étiqueté par une variable v de l'ensemble des variables non fixées à ce sommet. Cet arc correspond au choix de la valeur vrai pour v.

Définition 4.4 *(Graphe de décision) Pour une formule propositionnelle ϕ sur $\mathcal{V}$ on définit un graphe orienté acyclique :*

$$\mathcal{G}_\phi = \langle S, A, \lambda_S, \lambda_A \rangle$$

où S est l'ensemble des sommets, $A \subset S \times S$ l'ensemble des arcs, $\lambda_A : A \to \mathcal{V}$ est une fonction d'étiquetage des arcs et λ_S une fonction d'étiquetage des sommets.
L'étiquette d'un sommet est un triplet $\langle \psi, D, L \rangle$ qui contient les informations relatives au point de décision représenté :

- *ψ est une formule propositionnelle sur $\mathcal{V}$;*
- *D est l'ensemble des variables pour lesquelles on a pris la décision de les fixer à vrai ;*
- *L est l'ensemble des variables dont la valeur n'est pas encore fixée.*

Proposition 4.3 *Les ensembles S et A de $\mathcal{G}_\phi$ sont définis inductivement comme les ensembles minimaux tels que :*

1. *Le sommet initial s_0 étiqueté par $\langle \phi, \emptyset, \mathcal{V}_{\phi_U} \rangle$ est dans S*
2. *Pour tout sommet s dans S étiqueté $\langle \psi, D, L \rangle$, pour tout $v \in \mathcal{V}$, si les conditions 4.14 sont satisfaites, alors le sommet s' étiqueté $\langle \psi', D', L' \rangle$ est dans S et il existe un arc (s, s') dans A étiqueté par v.*

Condition d'existence d'un arc Dans le graphe $\mathcal{G}_\phi$, on a un arc étiqueté v du sommet s étiqueté $\langle \psi, D, L \rangle$ au sommet s' étiqueté $\langle \psi', D', L' \rangle$ si et seulement si les conditions 4.14 sont satisfaites.

$$
\begin{aligned}
&1)\ v \in L \\
&2)\ \psi' = \psi_v \\
&3)\ D' = D \cup \{v\} \\
&4)\ L' = L \setminus \{v\} \setminus \{x \in L \mid (\psi'_x = \bot) \vee (\psi'_{\neg x} = \bot)\}
\end{aligned}
\tag{4.14}
$$

La condition 1 indique que la décision porte sur une variable dont la valeur n'est pas encore fixée. La condition 2 exprime le choix de v à vrai. La formule résultante est alors le cofacteur positif de ψ pour v. La condition 3 ajoute v à l'ensemble des variables qu'on a décidé de fixer à vrai. La dernière condition actualise l'ensemble des variables non encore fixées. On supprime la variable v et toutes celles dont la valeur se trouve fixée à vrai ($\{x \in L \mid \psi'_{\neg x} = \bot\}$) ou a faux ($\{x \in L \mid (\psi'_x = \bot)\}$) par la décision de mettre v à vrai.

Proposition 4.4 *Pour un ensemble $\mathcal{V}$ fini, $\mathcal{G}_\phi$ est fini.*

C'est une conséquence de la condition 4 dans l'équation 4.14 qui implique que l'ensemble des variables non fixées est strictement décroissant sur tout chemin du graphe.

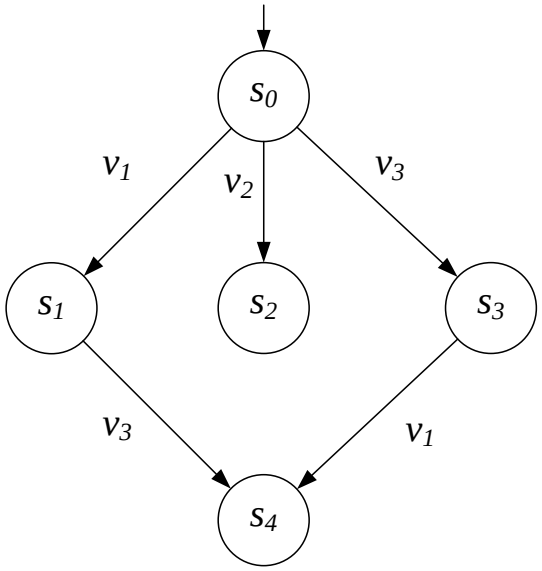
Définition 4.5 *(Sommet acceptant) Pour un sommet s de $\mathcal{G}_\phi$ étiqueté $\langle \psi, D, L \rangle$, soit f la valuation de $\mathcal{V}$ qui affecte vrai à toutes les variables de $D \cup \mathcal{V}_{\psi_{F^+}}$ et faux aux autres. Le sommet s est dit acceptant si f satisfait ϕ.*

Tous les sommets de $\mathcal{G}_\phi$ ne sont pas nécessairement acceptants comme le montre le contre-exemple suivant :
Soit $\mathcal{V} = \{v_1, v_2, v_3\}$ et $\phi = (\neg v_1 \vee v_2 \vee v_3) \wedge (\neg v_2 \vee \neg v_3)$. En prenant la décision d'affecter vrai à v_1, on obtient le sommet étiqueté par $\langle (\neg v_2 \wedge v_3) \vee (v_2 \wedge \neg v_3), \{v_1\}, \{v_2, v_3\}\rangle$. Pour ce sommet, $\kappa = \neg v_2 \wedge \neg v_3$ et $\psi_\kappa = \bot$, ce qui correspond à la valuation $f(v_1) = \mathsf{vrai}$ et $f(v_2) = f(v_3) = \mathsf{faux}$ qui ne satisfait pas ϕ.

Proposition 4.5 *(Complétude) $\mathcal{G}_\phi$ accepte toutes les valuations satisfaisant ϕ.*

La preuve donnée dans l'annexe D.

Exemple Pour la formule $\phi = (\neg v_1 \vee \neg v_2) \wedge (\neg v_2 \vee \neg v_3)$ définie sur $\mathcal{V} = \{v_1, v_2, v_3\}$, on obtient le graphe de la figure 4.17.



S	ψ	D	L
s_0	$(\neg v_1 \vee \neg v_2) \wedge (\neg v_2 \vee \neg v_3)$	$\varnothing$	$\{v_1, v_2, v_3\}$
s_1	$\neg v_2$	$\{v_1\}$	$\{v_3\}$
s_2	$\neg v_1 \wedge \neg v_3$	$\{v_2\}$	$\varnothing$
s_3	$\neg v_2$	$\{v_3\}$	$\{v_1\}$
s_4	$\neg v_2$	$\{v_1, v_3\}$	$\varnothing$

Figure 4.17 – Graphe de décision pour $\phi = (\neg v_1 \vee \neg v_2) \wedge (\neg v_2 \vee \neg v_3)$

Dans ce cas très simple, tous les sommets sauf le sommet initial donnent des solutions :

- $s_1 \rightarrow \{v_1\}$ (*i.e.*,la valuation $v_1 = \mathsf{vrai}, v_2 = \mathsf{faux}, v_3 = \mathsf{faux}$)
- $s_2 \rightarrow \{v_2\}$ (*i.e.*,la valuation $v_1 = \mathsf{faux}, v_2 = \mathsf{vrai}, v_3 = \mathsf{faux}$)
- $s_3 \rightarrow \{v_3\}$ (*i.e.*,la valuation $v_1 = \mathsf{faux}, v_2 = \mathsf{faux}, v_3 = \mathsf{vrai}$)
- $s_4 \rightarrow \{v_1, v_3\}$ (*i.e.*,la valuation $v_1 = \mathsf{vrai}, v_2 = \mathsf{faux}, v_3 = \mathsf{vrai}$)

4.3.2 La priorité

4.3.2.1 Définitions

Définition 4.6 *(Relation de priorité) Une spécification de priorité $\mathcal{P}$ sur un ensemble de variables $\mathcal{V}$ est une relation binaire d'ordre partiel strict sur $\mathcal{V}$.*

$(v_1, v_2) \in \mathcal{P}$ est noté $v_1 \dashrightarrow v_2$ et se lit v_1 est prioritaire sur v_2.

Par définition d'un ordre partiel strict, la relation $\mathcal{P}$ est irréflexive, asymétrique et transitive. Cette définition laisse une grande liberté de choix au niveau de la structure de priorité qui peut être

- une chaîne (ou ordre linéaire), *e.g.,*$v_1 \dashrightarrow v_2 \dashrightarrow v_3$;
- un arbre, *e.g.,*$v_1 \dashrightarrow v_2$ et $v_1 \dashrightarrow v_3$;
- une forêt, *e.g.,*$v_1 \dashrightarrow v_2$ et $v_1 \dashrightarrow v_3$ et $v_4 \dashrightarrow v_5$;
- un graphe orienté acyclique (DAG), *e.g.,*$v_1 \dashrightarrow v_2$ et $v_1 \dashrightarrow v_3$ et $v_4 \dashrightarrow v_2$.

Ces structures ont été indiquées dans l'ordre de complexité croissante vis à vis de leur prise en compte.

Approche du traitement des priorités Le principe du traitement est de prendre les décisions quant aux variables à mettre à vrai dans un ordre qui respecte $\mathcal{P}$. Ainsi, si on a $v_1 \dashrightarrow v_2$, on prendra d'abord la décision sur v_1. Cette décision pourra très bien avoir pour conséquence d'exclure la possibilité pour v_2 d'être sélectionnée ultérieurement. Une telle stratégie est d'application immédiate pour un ordre linéaire. Pour un arbre ou une forêt, l'ordre des décisions est moins évident. On peut faire un tri topologique sur les variables contraintes par $\mathcal{P}$ et suivre cet ordre pour les décisions. Dans le cas d'un DAG le choix paraît beaucoup plus délicat.

Dans la section précédente, nous avons expliqué que pour une formule ψ les décisions portaient sur les variables de $\mathcal{V}_{\psi_U}$. L'idée est donc de limiter le choix des variables afin de respecter les priorités. On introduit pour cela le concept de variables *candidates à une décision.*

Définition 4.7 *L'ensemble $\mathcal{C}(\psi, \mathcal{P})$ des variables candidates à une décision pour la formule ψ sous la priorité $\mathcal{P}$ par*

$$\mathcal{C}(\psi, \mathcal{P}) = \{v \in \mathcal{V}_{\psi_U} \mid (\nexists v' \in \mathcal{V}_{\psi_U}) v' \dashrightarrow v\}$$

$\mathcal{C}(\psi, \mathcal{P})$ est donc un sous ensemble de $\mathcal{V}_{\psi_U}$ qui ne contient que les variables non contraintes par $\mathcal{P}$ et les variables contraintes par $\mathcal{P}$ les plus prioritaires.

4.3.2.2 Graphe de décision avec priorité

Le graphe de décision avec priorité est un graphe de décision (définition 4.4) et une relation de priorité. Dans les informations portées par les sommets il faut ajouter l'ensemble des variables

candidates.

Définition 4.8 *Un graphe de décision avec priorité est une paire $\mathcal{G}_{\phi,\mathcal{P}} = \langle \mathcal{G}_\phi, \mathcal{P} \rangle$ où $\mathcal{G}_\phi$ est le graphe de décision pour une formule propositionnelle ϕ sur $\mathcal{V}$ et $\mathcal{P}$ une relation de priorité sur $\mathcal{V}$. L'étiquette d'un sommet est un quadruplet $\langle \psi, D, L, C \rangle$. Les champs ψ, D, L sont hérités de $\mathcal{G}_\phi$. Le champ additionnel C est l'ensemble des variables candidates pour ce sommet.*

Proposition 4.6 *Les ensembles S et A de $\mathcal{G}_{\phi,\mathcal{P}}$ sont définis inductivement comme les ensembles minimaux tels que :*

1. *Le sommet initial s_0 étiqueté par $\langle \phi, \emptyset, \mathcal{V}_{\phi_U}, \mathcal{C}(\phi, \mathcal{P}) \rangle$ est dans S*
2. *Pour tout sommet s dans S étiqueté $\langle \psi, D, L, C \rangle$, pour tout $v \in \mathcal{V}$, si les conditions 4.15 sont satisfaites, alors le sommet s' étiqueté $\langle \psi', D', L', C' \rangle$ est dans S et il existe un arc (s, s') dans A étiqueté par v.*

Condition d'existence d'un arc en présence de priorité Dans le graphe $\mathcal{G}_{\phi,\mathcal{P}}$, on a un arc étiqueté v de $\langle \psi, F^+, U, C \rangle$ à $\langle \psi', F'^+, U', C' \rangle$ si et seulement si les conditions 4.15 sont satisfaites.

$$
\begin{aligned}
&1)\ v \in C \\
&2)\ \psi' = \psi_v \\
&3)\ D' = D \cup \{v\} \\
&4)\ L' = L \setminus \{v\} \setminus \{x \in L \mid (\psi'_x = \bot) \vee (\psi'_{\neg x} = \bot)\} \\
&5)\ C' = \mathcal{C}(\psi', \mathcal{P})
\end{aligned}
\tag{4.15}
$$

Comparée à l'équation 4.14, la condition 1 a été modifiée et la condition 5 a été ajoutée pour prendre en compte les priorités.

Proposition 4.7 *$\mathcal{G}_{\phi,\mathcal{P}}$ est un sous-graphe de $\mathcal{G}_\phi$.*

Pour chaque sommet la condition 1 de l'équation 4.15 impose de ne choisir qu'un sous-ensemble (l'ensemble des variables candidates) des choix qui étaient possibles dans $\mathcal{G}_\phi$.

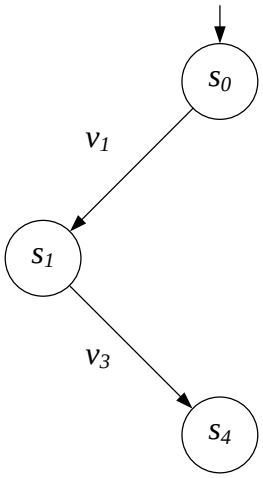
4.3.2.3 Exemples

Nous reprenons l'exemple de la page 97 en considérant différentes priorités.

Piorité 1 $v_1 \dashrightarrow v_2 \dashrightarrow v_3$

La figure 4.18 est le graphe de décision correspondant. On constate qu'il est effectivement un sous graphe de celui de la figure 4.17. Les solutions sont données par les sommets s_1 et s_4. $s_1 \rightarrow v_1 \wedge \neg v_2$; $s_4 \rightarrow v_1 \wedge \neg v_2 \wedge v_3$

Notons que les solutions $\neg v_2 \wedge v_3$, qui étaient possibles sans priorité, ont été rejetées par $\mathcal{P}$. En effet elles donneraient une priorité à v_3 sur v_1, ce qui viole $\mathcal{P}$. La solution $\neg v_1 \wedge v_2 \wedge \neg v_3$ a également été rejetée puisqu'elle aurait donné priorité à v_2 sur v_1.

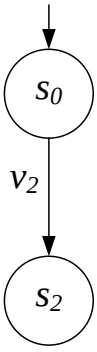


priorité: $v_1 \rightarrow v_2 \rightarrow v_3$

S	ψ	D	L	$C(\psi,\mathcal{P})$
s_0	$(\neg v_1 \vee \neg v_2) \wedge (\neg v_2 \vee \neg v_3)$	$\varnothing$	$\{v_1, v_2, v_3\}$	$\{v_1\}$
s_1	$\neg v_2$	$\{v_1\}$	$\{v_3\}$	$\{v_3\}$
s_4	$\neg v_2$	$\{v_1, v_3\}$	$\varnothing$	$\varnothing$

Figure 4.18 – Graphe de décision avec priorité v_1 --→ v_2 --→ v_3

Piorité 2 v_2 --→ v_1 et v_2 --→ v_3



priorité: $v_2 \rightarrow v_1$ et $v_2 \rightarrow v_3$

S	ψ	D	L	$C(\psi,\mathcal{P})$
s_0	$(\neg v_1 \vee \neg v_2) \wedge (\neg v_2 \vee \neg v_3)$	$\varnothing$	$\{v_1, v_2, v_3\}$	$\{v_2\}$
s_2	$\neg v_1 \wedge \neg v_3$	$\{v_2\}$	$\varnothing$	$\varnothing$

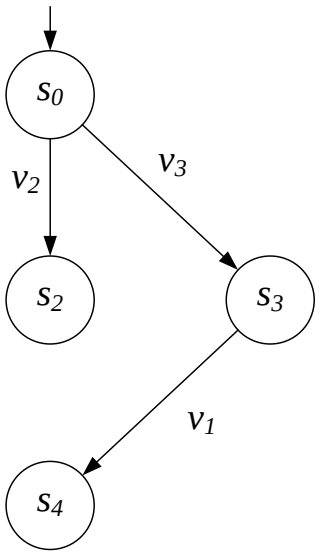
Figure 4.19 – Graphe de décision avec priorité v_2 --→ v_1 et v_2 --→ v_3

La figure 4.19 est le graphe de décision correspondant. Dans ce cas il n'y a que la solution donnée par le sommet s_2 : $\neg v_1 \wedge v_2 \wedge \neg v_3$.

Piorité 3 v_2 --→ v_1

Le graphe de décision est donné dans la figure 4.20.

Les sommets s_2, s_3 et s_4 fournissent respectivement les solutions suivantes : $\neg v_1 \wedge v_2 \wedge \neg v_3$, $v_3 \wedge \neg v_2$ et $v_1 \wedge \neg v_2 \wedge v_3$. Notons que v_1 était exclu des décisions pour le sommet s_0, mais



priorité: $v_2 \rightarrow v_1$

S	ψ	D	L	$C(\psi,\mathcal{P})$
s_0	$(\neg v_1 \vee \neg v_2) \wedge (\neg v_2 \vee \neg v_3)$	$\varnothing$	$\{v_1, v_2, v_3\}$	$\{v_2, v_3\}$
s_2	$\neg v_1 \wedge \neg v_3$	$\{v_2\}$	$\varnothing$	$\varnothing$
s_3	$\neg v_2$	$\{v_3\}$	$\{v_1\}$	$\{v_1\}$
s_4	$\neg v_2$	$\{v_1, v_3\}$	$\varnothing$	$\varnothing$

Figure 4.20 – Graphe de décision avec priorité $v_2 \dashrightarrow v_1$

il a pu être considéré pour le sommet s_3. En effet la priorité choisie permet d'avoir v_1 et v_2 simultanément à vrai, mais elle exclut v_1 à vrai sans v_2 à vrai.

4.3.3 Algorithme simplifié d'application des priorités

En pratique, construire et manipuler le graphe de décision peut devenir très coûteux. Nous proposons d'exprimer les restrictions imposées par les priorités sur la formule ϕ par une autre formule booléenne $\Psi(\phi, \mathcal{P})$. La conjonction $\phi \wedge \Psi(\phi, \mathcal{P})$ représentera alors les solutions qui satisfont ϕ et respectent $\mathcal{P}$.

Comme le graphe de décision avec priorité, $\Psi(\phi, \mathcal{P})$ est définie récursivement. La condition d'arrêt est cependant différente. Si on considère le graphe de la figure 4.18, dans le sommet s_1, $U = C$, c'est-à-dire que la priorité n'intervient plus à partir de cette décision. Toutes les décisions pour les successeurs de s_1 seront les mêmes et ϕ n'aura plus à être modifiée.

Définition 4.9 *Formellement, la fonction $\Psi(\psi, \mathcal{P})$ est définie par*

$$\Psi(\psi,\mathcal{P}) = \begin{array}{ll} \top & \textit{si } \mathcal{V}_{\psi_U} = \mathcal{C}(\psi,\mathcal{P}) \\ \left(\bigvee_{v\in\mathcal{C}(\psi,\mathcal{P})} \left(v \wedge \Psi(\psi_v,\mathcal{P})\right)\right) \vee \left((\bigwedge_{v\in\mathcal{V}_{\psi_U}} \neg v) \wedge \psi\right) & \textit{sinon.} \end{array}$$

La première ligne exprime la condition d'arrêt de la récursion (l'ensemble des variables candidates est égal à l'ensemble des variables non fixées). On retourne alors $\top$ qui est l'élément neutre de la conjonction. Dans le pire des cas, cette condition est atteinte lorsque les deux ensembles deviennent vides. La seconde ligne est une disjonction de deux expressions. Celle de

gauche $\bigvee_{v\in\mathcal{C}(\psi,\mathcal{P})}(v \wedge \Psi(\psi_v,\mathcal{P}))$ étudie récursivement les contraintes induites par la décision d'affecter vrai aux variables de $\mathcal{C}(\psi,\mathcal{P})$. L'expression de droite $\bigwedge_{v\in\mathcal{V}_{\psi_U}} \neg v \wedge \psi$ exprime le fait qu'il est tout à fait possible d'affecter faux à toutes les variables de $\mathcal{V}_{\psi_U}$ et qu'il convient donc de préserver cette possibilité.

4.3.4 Bilan de la section

Dans cette section consacrée aux priorités, nous avons étudié une manière alternative de résoudre les conflits entre événements concurrents. Plutôt que d'émuler le comportement d'un système à priorités à l'aide de contraintes CCSL, nous avons proposé de traiter des priorités lors de la simulation. Notre objectif était double : 1) alléger la spécification d'un tel type de système ; 2) séparer clairement ce que CCSL est en mesure de modéliser et la sémantique de simulation d'une spécification. Une spécification de priorité est alors possible et celle-ci n'impacte en aucun cas la spécification initiale du système. L'intérêt principal d'une telle approche est de pouvoir aiguiller la simulation sur certains scénarii plutôt que d'autres. Dans le cas où ces scénarii seraient souhaités pour toutes simulations, il reviendra au concepteur de repenser sa spécification CCSL pour les intégrer en "dur" dans son design. Dans le cas contraire, la spécification des blocs restera inchangée et seule une nouvelle spécification de priorités sera requise.

La priorité considérée ici était une priorité sur l'instant, cependant il est possible d'envisager la définition de politiques de simulation plus complexes (*i.e.,*au sens ordonnancement) et ainsi pouvoir analyser le comportement d'un système vis-à-vis de ces différentes politiques de simulations sans avoir à changer la description du système.

4.4 Bilan du chapitre

Dans ce chapitre, nous avons exploré les possibilités de modélisation offertes par le langage CCSL en appliquant notre étude à des systèmes communicants. Nous avons identifié, à haut niveau d'abstraction, les aspects contrôles nécessaires à la modélisation transactionnelle et avons donné les briques de base permettant une spécification plus aisée de tels systèmes. La présence quasi systématique de conflits d'accès au média de communication nous a mené à considérer une façon plus directe de modéliser/résoudre ces conflits. Nous avons proposé une manière de spécifier un système de priorité en complément à la spécification CCSL, sans modifier cette dernière. Elle se contente d'ajouter des contraintes de priorité qui ont pour effet de restreindre l'ensemble des évolutions possibles. Une technique pour trouver ces évolutions contraintes a été présentée.

Chapitre 5

Conclusions et perspectives

Contributions : Nous avons étudié les problèmes liés à la modélisation de systèmes électroniques numériques. Ceci nous a conduit, dans le chapitre 2, à évaluer les possibilités de modélisation de la librairie SystemC, du format IP-Xact et du profil UML-MARTE. Les résultats de cette analyse peuvent se résumer ainsi :

SystemC facilite la modélisation d'applications comprenant aussi bien du matériel que du logiciel. Cette modélisation peut être conduite à différents niveaux d'abstraction. Le système modélisé peut être ensuite simulé de façon très efficace en temps de calcul. Pour ces raisons SystemC est devenu incontournable dans le domaine de l'ESL (*Electronic System Level*). Lorsqu'on désire aller au-delà de l'aspect simulation, par exemple pour faire de la vérification de propriétés, l'avantage est moins évident car un design SystemC est encodé en C^{++}, rendant son interprétation et ses analyses complexes.

IP-Xact est un langage de description d'architecture largement diffusé auprès des industriels. Il se concentre sur la modélisation de composants/IPs et leur assemblage au sein d'un design. Il se révèle particulièrement efficace pour les systèmes construits autour de bus standards comme AMBA ou OCP-IP. Si les aspects structurels sont très bien pris en compte par IP-Xact, les aspects comportementaux sont son point faible.

L'utilisation du profil UML-MARTE en ESL s'inscrit plus dans une vision *ingénierie des modèles*. Ce profil bénéficie de la puissance de modélisation d'UML pouvant couvrir à la fois les aspects structurels et comportementaux. En tant que profil dédié aux systèmes embarqués temps réel, il bénéficie en plus d'éléments de modèles spécialisés (usage de stéréotypes) répondant à des besoins au niveau logiciel et matériel. MARTE étant d'introduction récente, les exemples industriels sont encore peu nombreux.

Nous nous sommes donc intéressé à une manière de profiter de la pléiade de designs existants en SystemC mais dans le format UML-MARTE. Nombreuses sont les passerelles allant vers SystemC, mais la passerelle effectuant le chemin inverse fut introuvable. Des outils permettant la navigation entre IP-Xact et UML-MARTE existant déjà, nous avons limité notre champ d'action à la création d'une passerelle entre SystemC et IP-Xact. Nous avons donc proposé, dans le chapitre 3, un outil (SCiPX) permettant de surmonter la barrière des formats. Notre outil permet d'extraire les aspects structurels d'un design SystemC pour en produire une version IP-Xact. Cet outil reste limité à un sous ensemble de programme SystemC. Cependant, les règles de codage acceptées ont été voulues les plus larges possibles. Ainsi, IP-Xact est promu au rang de format pivot pour une transformation supplémentaire dans le format UML-MARTE. Il en résulte une vue architecturale d'un programme SystemC au format UML-MARTE.

Dans le chapitre 4, nous avons exploité les possibilités offertes par le profil UML-MARTE, son modèle de temps et le langage de spécification de contraintes temporelles (CCSL). En UML-MARTE, des éléments de modèle structurels, comme les ports, peuvent être stéréotypés de façon à les lier explicitement à des horloges logiques. Des contraintes d'horloges, exprimées en CCSL permettent alors de spécifier des comportement attendus. C'est l'approche que nous avons adoptée pour modéliser des protocoles de communication entre composants à différents niveaux d'abstraction. Les contraintes sont imposées de façon incrémentale, leurs effets sont simulés et analysés à l'aide de l'outil TimeSquare. Il est même possible de générer automatiquement des observateurs en langages de description de matériel[?,?] afin de pouvoir vérifier des propriétés dans l'environnement de départ. Notre contribution porte plus particulièrement sur la modélisation de protocoles à l'aide d'horloges logiques. Nous avons défini des règles génériques de générations de contraintes CCSL dans le but d'annoter automatiquement les éléments d'interfaces d'un modèle structurel.

Nous avons ensuite considéré des problèmes rencontrés lors de la modélisation de comportements concurrents. Nous avons mis en évidence un manque d'expressivité de CCSL en ce qui concerne la spécification de choix (priorités) dans l'instant logique. Nous avons donc proposé une méthode d'enrichissement de CCSL pour lui permettre de manipuler ce concept de priorités. Ce choix doit rester dans l'ensemble des évolutions possibles d'un système et ne pas occasionner de blocages superflus. Nous avons ensuite proposé un algorithme permettant de traiter les priorités de manière orthogonale à la spécification des contraintes CCSL d'un design.

Perspectives : Dans l'optique d'intégration d'un composant propriétaire extérieur, une information nécessaire (et fournie) est l'interface de celui-ci. Le fournisseur ne donne généralement qu'une version compilée du comportement d'un composant. C'est la raison pour laquelle seul l'aspect structurel a été considéré dans cette thèse. Le format IP-Xact convient parfaitement à cette vue boîte noire (ou encore "IP") d'un design. Ce choix présente tout de même un in-

convénient : IP-Xact délègue la modélisation des comportements à des langages de description de matériel (*e.g.*,SystemC, VHDL, Verilog). L'extraction des comportements d'un composant ou d'une IP est en revanche très pertinente lorque le modèle cible est un modèle UML-MARTE. Une piste d'étude serait de considérer une passerelle directe de SystemC vers MARTE. En plus de pouvoir annoter un design, il serait également possible de vérifier le comportement d'un système vis-à-vis de contraintes CCSL en restant au niveau UML-MARTE.

Pour résoudre les problèmes de priorité nous avons choisi une approche décisionnelle qui s'appuie sur une séquence de choix. L'utilisation de l'algorithme est "notre" manière de résoudre le problème. D'autres approches sont envisageables. Par exemple, on pourrait considérer ce problème comme un simple calcul d'accessibilité dans le graphe priorisé. Quoi qu'il en soit, cet enrichissement permettra à terme, de construire différentes politiques de simulations plus complexes. La gestion des conflits a été considérée ici au niveau de l'instant logique. Des priorités changeant en fonction de l'évolution du système sont tout à fait concevables. Il serait intéressant de les considérer en relation avec la spécification de politiques de simulation telles qu'on peut les trouver dans la théorie de l'ordonnancement.

Annexe A

Résultats de l'exécution de SCiPX sur des constructions RTL

Objet :

Cette annexe présente le résultat d'une exécution de l'outil SCiPX sur un design SystemC générique de niveau RTL. Nous détaillons tout d'abord le code SystemC du design utilisé ainsi que les difficultés d'analyses qui peuvent en découler. Nous terminons en fournissant le modèle IP-Xact généré par notre outil ainsi qu'une vue graphique de ce modèle après conversion au format UML-MARTE

Le code source ci-dessous illustre un certain nombre de déclarations que notre outil SCiPX est capable de traiter. Le premier morceau de code est une implémentation d'un composant *Source*. La même description pour un composant *cible*, étant le miroir de la première, sera omise.

Les lignes 2 à 4 illustrent différentes manières de déclarer des objets en C^{++} (la ligne 2 pour une déclaration statique, la 3 pour dynamique et finalement la 4 pour la déclaration de tableaux). Étant donné que SystemC ne permet pas l'ajout de nouveaux objets (architecturaux) aux designs après la phase d'élaboration, tous les ports dynamiques doivent avoir été créés lorsque celle-ci se termine. Connaissant l'adresse de ces objets en mémoire (le port et la référence de pointeurs de port dans un composant), une simple comparaison entre l'adresse du port et de la valeur du pointeur de port nous permet de retrouver les informations relatives aux noms des ports dynamiques. Le cas des tableaux de port est très similaire à la méthode utilisée pour un port dynamique. L'ensemble des ports du tableau sont instanciés d'un bloc. Le premier port du tableau est détecté comme un simple port dynamique. Une simple comparaison entre les adresses de port non cartographiées et la taille du port nous permet de relier les ports non cartographiées au premier port du tableau.

Listing A.1 – Definition du composant Source en SystemC

```
1 SC_MODULE(Source){
2   sc_out<statType> statOUT;
3   sc_out<dynType> * dynOUT;
4   sc_out<arrayType> * arrayOUT;
5   SC_HAS_PROCESS(Source);
6   Source(sc_module_name name, unsigned arraySize){
7     dynOUT = new sc_out<dynType>;
8     arrayOUT = new sc_out<arrayType>[arraySize];}};
```

Le code du listing A.2 considère l'instanciation des deux composants Source et Target dans un sc_main. Une telle déclaration est valide en SystemC car il s'agit de simple code C^{++}. Mais là encore, les interfaces et les interconnexions sont créées conformément à certains paramètres et qui ne peuvent être analysés par analyse statique classique (*i.e.,*sans exécution symbolique) car ils contiennent des boucles et branchements conditionnelles. Combiner les approches dynamiques et statiques évite cette limitation. Cette dynamicité est néanmoins limitée à la phase d'élaboration où les objets de la simulation sont créés et liés. La vraie partie dynamique (l'exécution du programme/simulation) aura lieu après la construction de la hiérarchie du design.

Listing A.2 – Instanciation des deux composants en SystemC

```
1 int sc_main(int argc,char** argv){
2   unsigned arraySize=2;
3   bool invertArray=true;
```

```
4   Source sourceInst("sourceInst", arraySize);
5   Target targetInst("targetInst", arraySize);
6   sc_signal<statType> statSIG;
7   sc_signal<dynType> dynSIG;
8   sc_signal<arrayType> * arraySIG;
9   arraySIG = new sc_signal<arrayType>[arraySize];
10  sourceInst.statOUT.bind(statSIG);
11  sourceInst.dynOUT->bind(dynSIG);
12  for(unsigned i=0;i<arraySize;i++)
13    sourceInst.arrayOUT[i].bind(arraySIG[i]);
14  targetInst.statIN.bind(statSIG);
15  targetInst.dynIN->bind(dynSIG);
16  if(!invertArray) for(unsigned i=0;i<arraySize;i++)
17    targetInst.arrayIN[i].bind(arraySIG[i]);
18  else for(unsigned i=0;i<arraySize;i++)
19    targetInst.arrayIN[i].bind(arraySIG[arraySize-i-1]);
20  sc_start();
21  return 0;}
```

Après le traitement du code source de SystemC, le modèle IP-Xact produite par SCiPX est disponible. Les équivalents IP-Xact sont alors produits et sont disponibles dans les listing A.3 et A.4. Le premier représente le code correspondant pour le composant source et pour le second le design correspondant. Ce modèle peut être traité à nouveau afin de produire un modèle UML-MARTE équivalent, fournissant une description visuelle que l'on peut voir sur la capture d'écran de la figure A.1.

Listing A.3 – Définition du composant Source en IP-Xact

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <spirit:component xmlns:spirit="http://www.spiritconsortium.org/XMLSchema/
       SPIRIT/1.4" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.spiritconsortium.org/XMLSchema/SPIRIT
       /1.4 http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.4/index.xsd ">
3  <spirit:vendor>vendor</spirit:vendor>
4  <spirit:library>lib</spirit:library>
5  <spirit:name>Source_b1_-1__dynOUT_0__statOUT_-1__arrayOUT_1__</spirit:name>
6  <spirit:version>version</spirit:version>
7  <spirit:model>
8  <spirit:views>
9  <spirit:view>
10 <spirit:name>Source_b1_-1__dynOUT_0__statOUT_-1__arrayOUT_1___view</
       spirit:name>
11 <spirit:envIdentifier>systemc:timesquare:null</spirit:envIdentifier>
```

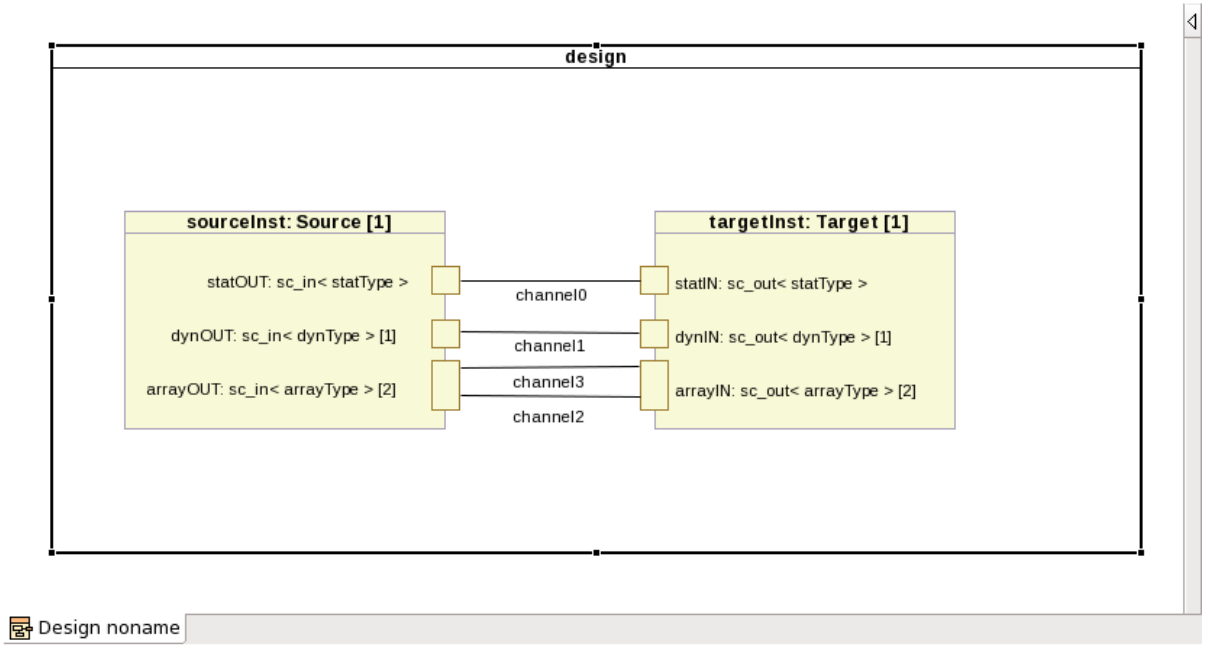


Figure A.1 – Diagramme UML-MARTE du design SystemC

```
12 </spirit:view>
13 </spirit:views>
14 <spirit:ports>
15 <spirit:port>
16 <spirit:name>statOUT</spirit:name>
17 <spirit:description>sc_out&lt;statType&gt;@ Source::statOUT --&gt;
       classsc__core_1_1sc__out</spirit:description>
18 <spirit:wire spirit:allLogicalDirectionsAllowed="false">
19 <spirit:direction>out</spirit:direction>
20 <spirit:wireTypeDefs>
21 <spirit:wireTypeDef>
22 <spirit:typeName spirit:constrained="false">sc_core::sc_out&lt;bool&gt;</
       spirit:typeName>
23 <spirit:viewNameRef>Source_b1_-1__dynOUT_0__statOUT_-1__arrayOUT_1___view</
       spirit:viewNameRef>
24 </spirit:wireTypeDef>
25 </spirit:wireTypeDefs>
26 </spirit:wire>
27 </spirit:port>
28 <spirit:port>
29 <spirit:name>dynOUT</spirit:name>
```

```
30 <spirit:description>sc_out&lt;dynType&gt;*@ Source::dynOUT —&gt;
       classsc__core_1_1sc__out</spirit:description>
31 <spirit:wire spirit:allLogicalDirectionsAllowed="false">
32 <spirit:direction>out</spirit:direction>
33 <spirit:vector>
34 <spirit:left spirit:dependency="" spirit:format="long" spirit:maximum=""
       spirit:minimum="" spirit:order="0.0" spirit:prompt="" spirit:rangeType=
       "float" spirit:resolve="immediate">0</spirit:left>
35 <spirit:right spirit:dependency="" spirit:format="long" spirit:maximum=""
       spirit:minimum="" spirit:order="0.0" spirit:prompt="" spirit:rangeType=
       "float" spirit:resolve="immediate">0</spirit:right>
36 </spirit:vector>
37 <spirit:wireTypeDefs>
38 <spirit:wireTypeDef>
39 <spirit:typeName spirit:constrained="false">sc_core::sc_out&lt;bool&gt;</
       spirit:typeName>
40 <spirit:viewNameRef>Source_b1_-1__dynOUT_0__statOUT_-1__arrayOUT_1___view</
       spirit:viewNameRef>
41 </spirit:wireTypeDef>
42 </spirit:wireTypeDefs>
43 </spirit:wire>
44 </spirit:port>
45 <spirit:port>
46 <spirit:name>arrayOUT</spirit:name>
47 <spirit:description>sc_out&lt;arrayType&gt;*@ Source::arrayOUT —&gt;
       classsc__core_1_1sc__out</spirit:description>
48 <spirit:wire spirit:allLogicalDirectionsAllowed="false">
49 <spirit:direction>out</spirit:direction>
50 <spirit:vector>
51 <spirit:left spirit:dependency="" spirit:format="long" spirit:maximum=""
       spirit:minimum="" spirit:order="0.0" spirit:prompt="" spirit:rangeType=
       "float" spirit:resolve="immediate">0</spirit:left>
52 <spirit:right spirit:dependency="" spirit:format="long" spirit:maximum=""
       spirit:minimum="" spirit:order="0.0" spirit:prompt="" spirit:rangeType=
       "float" spirit:resolve="immediate">1</spirit:right>
53 </spirit:vector>
54 <spirit:wireTypeDefs>
55 <spirit:wireTypeDef>
56 <spirit:typeName spirit:constrained="false">sc_core::sc_out&lt;bool&gt;</
       spirit:typeName>
57 <spirit:viewNameRef>Source_b1_-1__dynOUT_0__statOUT_-1__arrayOUT_1___view</
       spirit:viewNameRef>
```

```
58 </spirit:wireTypeDef>
59 </spirit:wireTypeDefs>
60 </spirit:wire>
61 </spirit:port>
62 </spirit:ports>
63 </spirit:model>
64 </spirit:component>
```

Listing A.4 – Instantiation des composants dans un design IP-Xact

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <spirit:design xmlns:spirit="http://www.spiritconsortium.org/XMLSchema/
       SPIRIT/1.4" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.spiritconsortium.org/XMLSchema/SPIRIT
       /1.4 http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.4/index.xsd ">
3  <spirit:vendor>vendor</spirit:vendor>
4  <spirit:library>lib</spirit:library>
5  <spirit:name>noname</spirit:name>
6  <spirit:version>version</spirit:version>
7  <spirit:componentInstances>
8  <spirit:componentInstance>
9  <spirit:instanceName>sourceInst</spirit:instanceName>
10 <spirit:componentRef spirit:library="lib" spirit:name="Source_b1_-1
       __dynOUT_0__statOUT_-1__arrayOUT_1__" spirit:vendor="vendor"
       spirit:version="version"/>
11 </spirit:componentInstance>
12 <spirit:componentInstance>
13 <spirit:instanceName>targetInst</spirit:instanceName>
14 <spirit:componentRef spirit:library="lib" spirit:name="Target_b1_-1
       __arrayIN_1__dynIN_0__statIN_-1__" spirit:vendor="vendor"
       spirit:version="version"/>
15 </spirit:componentInstance>
16 </spirit:componentInstances>
17 <spirit:adHocConnections>
18 <spirit:adHocConnection>
19 <spirit:name>channel0</spirit:name>
20 <spirit:internalPortReference spirit:componentRef="targetInst"
       spirit:portRef="statIN"/>
21 <spirit:internalPortReference spirit:componentRef="sourceInst"
       spirit:portRef="statOUT"/>
22 </spirit:adHocConnection>
23 <spirit:adHocConnection>
24 <spirit:name>channel1</spirit:name>
```

```
25 <spirit:internalPortReference spirit:componentRef="targetInst" spirit:left=
       "0" spirit:portRef="dynIN" spirit:right="0"/>
26 <spirit:internalPortReference spirit:componentRef="sourceInst" spirit:left=
       "0" spirit:portRef="dynOUT" spirit:right="0"/>
27 </spirit:adHocConnection>
28 <spirit:adHocConnection>
29 <spirit:name>channel2</spirit:name>
30 <spirit:internalPortReference spirit:componentRef="targetInst" spirit:left=
       "1" spirit:portRef="arrayIN" spirit:right="1"/>
31 <spirit:internalPortReference spirit:componentRef="sourceInst" spirit:left=
       "0" spirit:portRef="arrayOUT" spirit:right="0"/>
32 </spirit:adHocConnection>
33 <spirit:adHocConnection>
34 <spirit:name>channel3</spirit:name>
35 <spirit:internalPortReference spirit:componentRef="targetInst" spirit:left=
       "0" spirit:portRef="arrayIN" spirit:right="0"/>
36 <spirit:internalPortReference spirit:componentRef="sourceInst" spirit:left=
       "1" spirit:portRef="arrayOUT" spirit:right="1"/>
37 </spirit:adHocConnection>
38 <spirit:adHocConnection>
39 <spirit:name>channel4</spirit:name>
40 <spirit:internalPortReference spirit:componentRef="targetInst" spirit:left=
       "1" spirit:portRef="arrayIN" spirit:right="1"/>
41 </spirit:adHocConnection>
42 </spirit:adHocConnections>
43 </spirit:design>
```

Annexe B

Extension de la librairie SystemC

Objet :

La traçabilité d'objet durant une simulation SystemC est limitée aux concepts capturables par cette même librairie. Cependant, cette librairie peut être modifiée afin de combler ces manques. Nous proposons dans cette annexe, une extension de la librarie SystemC pour couvrir le concept particulier de Register tel que défini dans IP-Xact.

Dans la section 3, la bibliothèque SystemC a montré plusieurs limitations dans son implémentation en terme de possibilités d'analyse. Notre approche tentait de réconcilier des informations provenant de deux sources différentes (statique et dynamique). La fin de la phase d'élaboration marquait l'instant où le design devenait figé. Cependant, à ce point, n'étaient conservées/accessibles que les informations nécessaires à la simulation proprement dite. Il aurait été possible de modifier la bibliothèque SystemC afin qu'elle conserve plus d'informations. Cependant, modifier cette bibliothèque l'aurait rendue incompatible avec les librairies de composants existantes. Nous considérons dans cette annexe non pas les concepts déjà présents en SystemC mais une manière faire perdurer au moment de la phase dynamique des concepts externes à la librairie SystemC mais cependant présents dans le monde de la conception de systèmes.

Par exemple, nous avons considéré la description IP-Xact comme valide et complète si nous arrivions à extraire toutes les informations topologiques d'un design SystemC. Les critères étaient que les fichiers résultant puissent être analysés correctement par n'importe quel outil capable d'importer un design IP-Xact standard. Mais un autre aspect important d'un composant IP-Xact est détaillé par la *memory map* de celui-ci ainsi que la description de son espace d'adressage. Caché dans le paquet de composants mémoire, on peut trouver la notion de registres comme décrit dans la figure B.1.

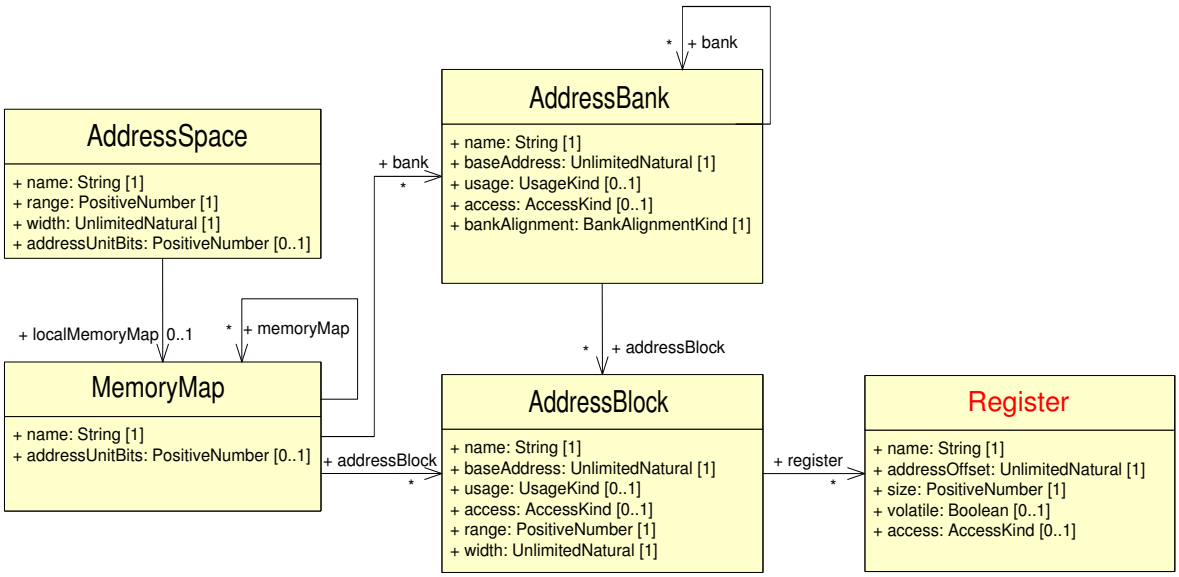


Figure B.1 – IP-Xact component memory metamodel

Ces registres, blocs mémoire et *memory map* ne sont pas explicitement désignées comme des objets spécifiques dans la librairie SystemC, même si la plupart des concepteurs les utilisent

implicitement comme des caractéristiques importantes de leur architecture. D'autre part, IP-Xact intègre des mécanismes pour exprimer ces entités puisque d'autres outils doivent utiliser ces informations au moment de l'assemblage. Un moyen facile de récupérer ces informations devrait être d'utiliser les mêmes techniques que celles mises en œuvre dans SCiPX. Pour ce faire, le concepteur devra utiliser des types/classes spécifiques qui puissent être suivis à l'exécution et puissent être accessibles publiquement. Évidemment, les concepteurs ont à utiliser les classes SystemC afin d'écrire un programme SystemC et nous ne pouvons ici que supposer que c'est bien le cas. Néanmoins, ces types/classes spécifiques peuvent être très facilement implémentés, voire même générés automatiquement.

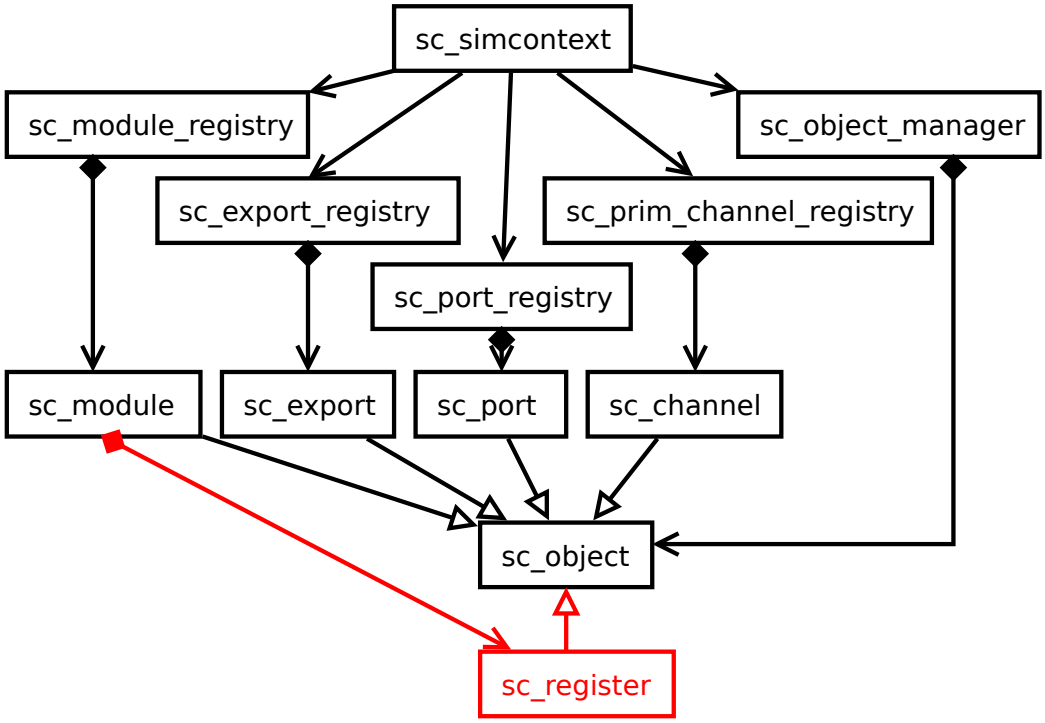


Figure B.2 – Intégration du concept registre

Pour illustrer le style de codage proposé, nous prenons comme exemple le registre. Un registre est un bloc spécifique de bits d'un composant matériel. Sa longueur dépend de l'architecture (*i.e.*,32 bits, 64 bits). Il est utilisé pour interroger, configurer ou contrôler un composant matériel. Un registre est donc étroitement lié à un composant et ce lien doit être préservé comme le décrit la figure B.2. En SystemC, les développeurs ont pour habitude de déclarer un registre à l'aide d'un type générique (*i.e.*,`unsigned int`). De ce fait, l'essence même d'un registre est perdue et il est interprété comme un `unsigned`. Nous proposons d'utiliser une classe spécifique pour chacun des éléments que l'on souhaite tracer comme les registres. Une déclaration de la classe registre

est donnée comme exemple dans la suite et peut être modifiée pour répondre aux besoins de traçabilité spécifique à chacun. Par ailleurs, cette déclaration utilise les mécanismes de SystemC sans avoir besoin de modifier cette dernière. Les développeurs auront seulement besoin de déclarer un registre avec la classe spécifique au lieu d'utiliser un type unsigned int.

Listing B.1 – Instanciation des deux composants en SystemC

```
1  class Register: public sc_object{
2  public:
3    Register();
4    virtual ~Register();
5    virtual const char * kind() const
6      {return "sc_register";}
7      unsigned m_value;
8      sc_object * m_parent_module;
9  };
10 Register::Register():
11   sc_object(sc_gen_unique_name("sc_register")){
12     sc_object * parent=get_parent();
13     while(parent!=NULL){
14       if(parent->kind()=="sc_module"){
15         m_parent_module=parent;
16         break;
17       }
18       parent=parent->get_parent();}}
```

Sur la première ligne, l'héritage à la classe sc_object assure que nous pourrons récupérer une référence à tous les objets Register instanciés durant la phase d'élaboration et stocké dans l'objet sc_simcontex. Des lignes cinq et six, nous surchargeons la méthode sc_object::kind() pour être capable d'identifier un registre dans un ensemble de sc_object. La ligne sept représente l'endroit où nous allons stocker la valeur de notre registre. Comme le lien entre le registre et le composant est important, nous avons besoin de garder une trace de ce confinement. C'est le but du contenu du constructeur des lignes dix à la fin. Il remplira automatiquement l'attribut m_parent_module avec une référence au plus proche parent sc_module au moment de l'instanciation. Lors du traitement de la phase d'élaboration, une référence à tous les registres sera stockée dans l'objet sc_simcontex. Parcourir la liste des sc_object permettra de les identifier. La lecture de l'attribut m_parent_module permettra de récupérer la référence du composant qui le contient. De là, les mêmes techniques expliquées dans les sections précédentes pourront être appliquées et cette information pourra être extraite pour ensuite être exploitée par le modèle IP-Xact.

Annexe C

Généralisation des équations CCSL

Objet :

Les équations CCSL nécessaires à la spécification de protocole sont souvent lourdes à écrire lorsque l'on considère des systèmes complexes. Cependant, pour certains paramètres fixés, il est possible de les générer automatiquement. Cette annexe contient les règles de générations automatiques pour différents paramètres.

Généralisation à un ensemble fini d'esclaves Si l'on considère un ensemble fini S d'esclaves connectés, il est alors possible de généraliser les contraintes de la manière décrites dans les équations C.1 :

$$
\begin{aligned}
begin &\boxed{\simeq} hready \\
hready &\boxed{\simeq} end \\
\forall n \in S, Slave_n\_begin &\boxed{\simeq} Slave_n\_hready \\
\forall n \in S, Slave_n\_hready &\boxed{\simeq} Slave_n\_end \\
begin &\boxed{=} \bigcup_{\forall n \in S} Slave_n\_begin \\
hready &\boxed{=} \bigcup_{\forall n \in S} Slave_n\_hready \\
end &\boxed{=} \bigcup_{\forall n \in S} Slave_n\_end \\
\forall i \in S, Slave_i\_begin &\boxed{\#} \bigcup_{\forall j \in S \setminus i} Slave_j\_begin \\
\forall i \in S, Slave_i\_hready &\boxed{\#} \bigcup_{\forall j \in S \setminus i} Slave_j\_hready
\end{aligned}
\tag{C.1}
$$

Généralisation à un ensemble fini d'esclaves et de maîtres Si l'on considère un ensemble fini S d'esclaves et un ensemble fini M de maîtres connectés, il est alors possible de généraliser les contraintes de la manière décrites dans les équations C.2 :

$$
\begin{array}{rcl}
\forall n \in M, Master_n\_begin & \boxed{\simeq} & Master_n\_hready \\
\forall n \in M, Master_n\_hready & \boxed{\simeq} & Master_n\_end \\
begin & \boxed{=} & \bigcup_{\forall n \in M} Master_n\_Start \\
hready & \boxed{=} & \bigcup_{\forall n \in M} Master_n\_hready \\
end & \boxed{=} & \bigcup_{\forall n \in M} Master_n\_end \\
\forall i \in M, Master_i\_begin & \boxed{\#} & \bigcup_{\forall j \in M \setminus i} Master_j\_begin \\
\forall i \in M, Master_i\_hready & \boxed{\#} & \bigcup_{\forall j \in M \setminus i} Master_j\_hready \\
Start & \boxed{\simeq} & hready \\
hready & \boxed{\simeq} & end \\
\forall n \in S, Slave_n\_begin & \boxed{\simeq} & Slave_n\_hready \\
\forall n \in S, Slave_n\_hready & \boxed{\simeq} & Slave_n\_end \\
begin & \boxed{=} & \bigcup_{\forall n \in S} Slave_n\_begin \\
hready & \boxed{=} & \bigcup_{\forall n \in S} Slave_n\_hready \\
end & \boxed{=} & \bigcup_{\forall n \in S} Slave_n\_end \\
\forall i \in S, Slave_i\_begin & \boxed{\#} & \bigcup_{\forall j \in S \setminus i} Slave_j\_begin \\
\forall i \in S, Slave_i\_hready & \boxed{\#} & \bigcup_{\forall j \in S \setminus i} Slave_j\_hready
\end{array}
\tag{C.2}
$$

Généralisation d'un système à priorités pour un ensemble fini de maîtres Pour chaque maître, il est possible de construire l'horloge logique représentant les requêtes effectives de plus hautes priorités de manière itérative. Pour le maître de plus haute priorité, cette horloge se résume à une horloge ne comportant aucun événement. Pour les suivants, cette horloge est l'union des horloges de requêtes effectives de tous les maîtres plus prioritaires. Dans le cas du deuxième plus prioritaire, cette horloge se réduit à l'horloge des requêtes émises par le plus prioritaire (*i.e.*,les

requêtes effectives sont les requêtes émises car on ne peut les interdire). Les requêtes émises sont laissées libres et se décomposent en deux sous catégories d'événements exclusifs, ceux en conflits avec l'horloge des requêtes effectives de plus hautes priorités et celles sans conflits. Les événements en conflits doivent alors générer autant d'autres événements se positionnant en fonction de l'horloge des requêtes effectives de plus hautes priorités (à savoir en exclusion avec celle-ci). Une telle spécification peut d'ores et déjà être exprimée à l'aide des équations C.3 où l'on indice par E les requêtes effectives d'un maître, par EG les requêtes effectives plus prioritaires, par C les requêtes en conflit, par CF (Conflict Free) les requêtes sans conflit et pour finir par D (Delayed) les requêtes générées par les requêtes en conflit.

$$
\begin{aligned}
\forall i \leq N, \quad & Master_i\_begin_{EG} && = && \bigcup_{\forall j, P(j) > P(i)} Master_j\_begin_E \\
\forall i \leq N, \quad & Master_i\_begin_E && = && Master_i\_begin_{CF} + Master_i\_begin_D \\
\forall i \leq N, \quad & Master_i\_begin_{EG} && \# && Master_i\_begin_E \\
\forall i \leq N, \quad & Master_i\_begin && = && Master_i\_begin_C + Master_i\_begin_{CF} \\
\forall i \leq N, \quad & Master_i\_begin_C && \# && Master_i\_begin * Master_i\_begin_{EG} \\
\forall i \leq N, \quad & Master_i\_begin_C && \# && Master_i\_begin_{CF} \\
\forall i \leq N, \quad & Master_i\_begin_C && \simeq && Master_i\_begin_D
\end{aligned}
\tag{C.3}
$$

Annexe D

Complément sur les Fonctions Booléennes

Objet :

Cette annexe contient des compléments sur les fonctions booléennes et les démonstrations des propriétés données au chapitre 4.

Pour les définitions de base et certains théorèmes classiques sur les fonctions booléennes nous renvoyons à l'ouvrage récent[?] d'Yves Crama et Peter L. Hammer intitulé *"Boolean Functions : Theory, Algorithms, and Applications"*. Pour les références concernant les BDD à l'article de Fabio Somenzi[?] et à l'ouvrage de Christoph Meinel et Thorsten Theobald[?].

D.1 Cubes et co-facteurs

Nous aurons besoin de la notion de *conjonction élémentaire* ou *cube* :

Définition D.1 *(Cube) Un cube sur $\mathcal{V}$ est une expression de la forme*

$$\left(\bigwedge_{x\in A} x\right) \wedge \left(\bigwedge_{x\in B} \neg x\right) \text{ où } A \subseteq \mathcal{V}, B \subseteq \mathcal{V}, A \cap B = \emptyset$$

Nous notons $\mathcal{K}(A, B)$ le cube sur $\mathcal{V}$ défini par les sous-ensembles disjoints A et B de $\mathcal{V}$

ainsi que des propriétés de co-facteurs[?] :

Théorème D.1 *Soit f et g deux fonctions booléennes de l'ensemble des variables $\mathcal{V}$. Pour tout $x, y \in \mathcal{V}$: les co-facteurs commutent*

$$(f_x)_y = (f_y)_x = f_{x\wedge y} = f_{y\wedge x} \tag{D.1}$$

et ils sont distributifs sur la négation, la conjonction et la disjonction.

$$(\neg f)_x = \neg(f_x) \tag{D.2}$$

$$(f \wedge g)_x = f_x \wedge g_x \tag{D.3}$$

$$(f \vee g)_x = f_x \vee g_x \tag{D.4}$$

Lemme D.1 *(Co-facteur d'un cube) Soit f une fonction booléenne de l'ensemble des variables $\mathcal{V}$. Soit κ un cube sur $\mathcal{V}$. Pour tout cube κ' obtenu par permutation des facteurs de κ on a*

$$f_\kappa = f_{\kappa'} \text{ pour tout } \kappa' \text{ permutation de } \kappa \tag{D.5}$$

Ceci résulte du fait que le co-facteurs commutent (équation D.1). On itère ensuite sur les facteurs de κ.

Lemme D.2 *Soit f une fonction booléenne de l'ensemble des variables $\mathcal{V}$. Soit κ un cube sur $\mathcal{V}$*

$$\kappa \wedge f_\kappa = \kappa \wedge f \tag{D.6}$$

Par la décomposition de Shannon 4.13, $v \wedge f = v \wedge \big((v \wedge f_v) \vee (\neg v \wedge f_{\neg v})\big) = v \wedge f_v$. On itère ensuite sur les littéraux de κ.

Lemme D.3 *Soit ϕ une formule sur $\mathcal{V}$, si $\phi \neq \bot$ alors l'ensemble $\mathcal{V}_{\phi_U}$ des variables de ϕ non fixées est*

$$\mathcal{V}_{\phi_U} = \{v \in \mathcal{V} \mid (\phi_v \neq \bot) \wedge (\phi_{\neg v} \neq \bot)\} \tag{D.7}$$

Preuve $\mathcal{V}_{\phi_U} = \mathcal{V} \setminus (\mathcal{V}_{\phi_{F^+}} \cup \mathcal{V}_{\phi_{F^-}})$ résulte de la partition de $\mathcal{V}$ (Définition 4.2). Puisque $\phi \neq \bot$ des propositions 4.1 et 4.2 on déduit $\mathcal{V}_{\phi_{F^+}} \cup \mathcal{V}_{\phi_{F^-}} = \{v \in \mathcal{V} \mid (\phi_{\neg v} = \bot) \vee (\phi_v = \bot)\}$, d'où en complémentant cet ensemble par rapport à $\mathcal{V}$, $\mathcal{V}_{\phi_U} = \{v \in \mathcal{V} \mid (\phi_v \neq \bot) \wedge (\phi_{\neg v} \neq \bot)\}$ ∎

Lemme D.4 *Soit ϕ une formule sur $\mathcal{V}$,*

$$(\phi \neq \bot) \Longrightarrow (\forall v \in \mathcal{V}_{\phi_U})\ (\phi_v \neq \bot) \tag{D.8}$$

Conséquence du lemme D.3.

D.2 Graphe de décision

Lemme D.5 *Toute solution de ϕ est reconnue par un sommet acceptant dans le graphe de ϕ.*

Preuve Soit μ une solution de ϕ, μ peut être représenté par son cube $\mathcal{K}(A, \mathcal{V} \setminus A)$. L'idée est de rechercher par traversée du graphe $\mathcal{G}_\phi$ un sommet acceptant la solution μ. Pour cela, nous définissons une procédure récursive de parcours $\mathsf{match}(s, \mu)$ définie ainsi :

Soit $< \psi, D, L >$ l'étiquette de s. 3 cas sont à considérer :

- $A = D \cup \mathcal{V}_{\psi_{F^+}}$ alors μ est reconnue directement par la définition 4.5 d'un sommet acceptant. retourner s.
- $A \subseteq D \cup \mathcal{V}_{\psi_{F^+}} \cup L$ alors on choisit une variable $v \in A \cap L$. Puisque $L = \mathcal{V}_{\psi_U}$, il existe dans $\mathcal{G}_\phi$ un sommet s' et un arc (s, s') étiqueté v. On procède donc à un appel récursif : retourner $\mathsf{match}(s', \mu)$.
- $A \cap \mathcal{V}_{\psi_{F^-}} \neq \emptyset$. Soit $v \in A \cap \mathcal{V}_{\psi_{F^-}}$. v ne peut donc pas être dans $\mathcal{V}_{\psi_U}$. Il n'est pas possible de choisir v à partir du sommet s. La traversée du graphe est donc stoppée. Nous allons montrer que cette situation ne peut pas se présenter. En effet, μ étant solution de ϕ, on peut toujours écrire ϕ sous la forme d'une disjonction $\phi = \mu \vee \phi'$. Soit $\delta = \bigwedge_{x \in D} x$ le cube représentant les décisions prises pour arriver au sommet s. On a $\psi = \phi_\delta$. Par application itérée de l'équation D.4 on obtient $\psi = \phi_\delta = \mu_\delta \vee \phi'_\delta$. Puisque $v \in A$, $\mu_{\delta \wedge v} \neq \bot$, donc $\psi_v = \mu_{\delta \wedge v} \vee \phi'_{\delta \wedge v} \neq \bot$. Par le lemme D.3, v est dans $\mathcal{V}_{\psi_U}$, ce qui est en contradiction avec l'hypothèse que $v \in \mathcal{V}_{\psi_{F^-}}$.

∎

Bibliographie

[1] C. André, F. Mallet, and J. DeAntoni. VHDL observers for clock constraint checking. In *Industrial Embedded Systems (SIES), 2010 International Symposium on*, pages 98 –107, July 2010.

[2] Charles André. Syntax and Semantics of the Clock Constraint Specification Language (CCSL). Research Report RR-6925, INRIA, 2009.

[3] Charles André. Modèles de temps et de contraintes temporelles de MARTE et leurs applications. Rapport de recherche RR-7788, INRIA, November 2011. Cours donné à l'École d'été temps réel - Brest - Août 2011.

[4] Charles André and Frédéric Mallet. Specification and verification of time requirements with CCSL and Esterel. In Christoph Kirsch and Mahmut Kandemir, editors, *Languages, Compilers, and Tools for Embedded Systems ACM SIGPLAN Notices*, volume 44, pages 167–176, Dublin, Ireland, 2009. ACM SIGPLAN/SIGBED.

[5] Charles André, Julien DeAntoni, Frédéric Mallet, and Robert de Simone. The time model of logical clocks available in the OMG MARTE profile. In Sandeep K. Shukla and Jean-Pierre Talpin, editors, *Synthesis of Embedded Software : Frameworks and Methodologies for Correctness by Construction*, chapter 7, pages 201–227. Springer Science+Business Media, LLC 2010, July 2010.

[6] Aynsley, John. OSCI TLM 2.0 Language Reference Manual, July 2009. `http://www.systemc.org/members/download_files/check_file?agreement=tlm_2-0_lrm`.

[7] David Berner, Hiren D. Patel, Deepak A. Mathaikutty, and Sandeep K. Shukla. Automated Extraction of Structural Information from SystemC-based IP for Validation. *Microprocessor Test and Verification, International Workshop on*, 0 :99–104, 2005.

[8] David Berner, Jean-Pierre Talpin, Hiren Patel, Deepak Abraham Mathaikutty, and Eep Shukla. SystemCXML : An extensible SystemC front end using XML. In *In Proceedings of the Forum on specification and design languages (FDL)*, 2005.

[9] Nicolas Blanc, Daniel Kroening, and Natasha Sharygina. Scoot : A Tool for the Analysis of SystemC Models. In *TACAS*, 2008.

[10] Lossan Bonde, Cédric Dumoulin, and Jean-Luc Dekeyser. Metamodels and MDA Transformations for Embedded Systems. In FDL'04,? pages 240–252.

[11] H. Broeders. Extracting behavior and dynamically generated hierarchy from SystemC models. M.sc. thesis, Delft, The Netherlands, December 2010.

[12] H. Broeders and R. van Leuken. Extracting behavior and dynamically generated hierarchy from SystemC models. In *Design Automation Conference (DAC 2011)*, San Diego, CA, June 2011.

[13] Yan Chen, Xuan Du, Xuegong Zhou, and Chenglian Peng. An Automatic Coverage Analysis for SystemC Using UML and Aspect-Oriented Technology. In Weiming Shen, Zongkai Lin, Jean-Paul A. Barthès, and Tangqiu Li, editors, *CSCWD (Selected papers)*, volume 3168 of *Lecture Notes in Computer Science*, pages 398–405. Springer, 2004.

[14] Yves Crama and Peter L. Hammer. *Boolean Functions : Theory, Algorithms, and Applications*. Number 142 in Encyclopedia of Mathematics and its Applications. Cambridge University Press, July 2011.

[15] Rolf Drechsler, Görschwin Fey, Christian Genz, and Daniel Große. SyCE : An integrated environment for system design in SystemC. In *In IEEE International Workshop on Rapid System Prototyping*, pages 258–260, 2005.

[16] *Forum on specification and Design Languages, FDL 2004, September 14-17, 2004, Lille, France, Proceedings*. ECSI, 2004.

[17] Görschwin Fey, Daniel Große, Tim Cassens, Christian Genz, Tim Warode, and Rolf Drechsler. ParSyC : An Efficient SystemC Parser. In *In Workshop on Synthesis And System Integration of Mixed Information technologies (SASIMI)*, pages 148–154, 2004.

[18] FZI Forschungszentrum Informatik, Department of Microelectronic System Design. KaSCPar - Karlsruhe SystemC Parser Suite. `http://www.fzi.de/sim/kascpar.html`.

[19] D. D. Gajski, J. Zhu, R. Doemer, A. Gerstlauer, and S. Zhao. *SpecC : Specification Language and Methodology*. Kluwer Academic Publishers, 2000.

[20] Christian Genz, Rolf Drechsler, Gerhard Angst, and Lothar Linhard. Visualization of SystemC Designs. In *ISCAS*, pages 413–416, 2007.

[21] Daniel Große, Rolf Drechsler, Lothar Linhard, and Gerhard Angst. Efficient Automatic Visualization of SystemC Designs. In *FDL*, pages 646–658, 2003.

[22] Th. Grötker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[23] Daniel Große and Rolf Drechsler. CheckSyC : An efficient property checker for RTL SystemC designs. In *In IEEE International Symposium on Circuits and Systems*, pages 4167–4170, 2005.

[24] Paula Herber, Joachim Fellmuth, and Sabine Glesner. Model checking SystemC designs using timed automata. In *Proceedings of the 6th IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis*, CODES+ISSS '08, pages 131–136, New York, NY, USA, 2008. ACM.

[25] Fernando Herrera, Pablo Sánchez, and Eugenio Villar. Modeling of csp, kpn and sr systems with systemc. In *FDL*, pages 572–583, 2003.

[26] Fernando Herrera, Pablo Sánchez, and Eugenio Villar. *Modeling of CSP, KPN and SR systems with systemC*, pages 133–148. Kluwer Academic Publishers, Norwell, MA, USA, 2004.

[27] Fernando Herrera and Eugenio Villar. A framework for heterogeneous specification and design of electronic embedded systems in SystemC. *ACM Trans. Des. Autom. Electron. Syst.*, 12 :1–31, May 2008.

[28] IEEE Standards Association. *Open SystemC Language Reference Manual*. Open SystemC Initiative, 2005. IEEE Std. 1666–2005.

[29] IP-Xact standard IEEE 1685. http ://www.accellera.org/downloads/ieee.

[30] Chris Lattner and Vikram Adve. LLVM : A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[31] Jean-François Le Tallec and Julien DeAntoni. Toward a TLM to RTL refinement : a formal approach. In *Proc. of the 3rd Junior Researcher W. on Real-Time Computing, JRWRTC'09, in conjunction with RTNS'09*, Paris, France, October 2009.

[32] Jean-François Le Tallec and Robert De Simone. SCIPX : a SystemC to IP-XACT extraction tool. In *ESLsyn : Electronic System Level Synthesis Conference*, San Diego, États-Unis, June 2011.

[33] Jean-François Le Tallec, Julien DeAntoni, Robert de Simone, Benoît Ferrero, Frédéric Mallet, and Laurent Maillet-Contoz. Combining SystemC, IP-XACT and UML-MARTE in model-based SoC design. In *Proc. of the 2011 Workshop on Model Based Engineering for Embedded Systems Design*, M-BED'2011, Grenoble, France, March 2011.

[34] LIP6, Laboratoire d'informatique de Paris 6. SoCLib - system on chip library. `http://www.soclib.fr`.

[35] Kevin Marquet and Matthieu Moy. PinaVM : a SystemC front-end based on an executable intermediate representation. In *International Conference on Embedded Software*, page 79, Scottsdale, USA, 10 2010. SD B.4.4, I.6.4, D.2.4 OpenTLM (projet Minalogic).

[36] Kevin Marquet, Matthieu Moy, and Bageshri Karkare. A theoretical and experimental review of SystemC front-ends. In *Forum for Design Languages (FDL)*, 2010. B.1.4, C.3 OpenTLM (Projet Minalogic).

[37] Deepak A. Mathaikutty and Sandeep K. Shukla. Mcf : A metamodeling-based component composition framework - composing systemc ips for executable system models. *IEEE Transactions on Very Large Scale Integration Systems*, 16 :792–805, 2008.

[38] Aamir Mehmood Khan. *Model-Based Design for On-Chip Systems : using and extending Marte and IP-Xact*. PhD thesis, Université de Nice/Sophia-Antipolis, March 2010.

[39] Christoph Meinel and Thorsten Theobald. *Algorithms and data structures in VLSI design : OBDD-foundations and applications*. Technology & Engineering. Springer, Nov. 1998.

[40] Matthieu Moy, Florence Maraninchi, and Laurent Maillet-Contoz. Pinapa : an extraction tool for SystemC descriptions of systems-on-a-chip. In *Proceedings of the 5th ACM international conference on Embedded software*, EMSOFT '05, pages 317–324, New York, NY, USA, 2005. ACM.

[41] Matthieu Moy, Florence Maraninchi, and Laurent Maillet-Contoz. LusSy : an open Tool for the Analysis of Systems-on-a-Chip at the Transaction Level. Design Automation for Embedded Systems, 2006.

[42] OMG. *UML Profile for Schedulability, Performance, and Time Specification*. Object Management Group, Object Management Group, Inc., 492 Old Connecticut Path, Framing-ham, MA 01701., January 2005. OMG document number : formal/05-01-02 (v1.1).

[43] OMG. *UML Profile for MARTE, v1.0*. Object Management Group, Nov. 2009. OMG document number : formal/09-11-02.

[44] Terence John Parr. *Language Translation Using PCCTS and C++ : A Reference Guide*. Automata Publishing Company, San Jose, CA 95129, 1996.

[45] Hiren D. Patel and Sandeep K. Shukla. Towards a heterogeneous simulation kernel for system level models : A SystemC kernel for synchronous data flow models. *VLSI, IEEE Computer Society Annual Symposium on*, 2004.

[46] David J. Pearce, Paul H. J. Kelly, and Chris Hankin. Efficient Field-Sensitive Pointer Analysis for C. In *In ACM workshop on Program Analysis for Software Tools and Engineering (PASTE*, pages 37–42. ACM Press, 2004.

[47] Sébastien Revol. *Profil UML pour TLM : contribution à la formalisation et à l'automatisation du flot de conception et vérification des systèmes sur puce*. PhD thesis, INPG, Grenoble, France, June 2008.

[48] Elvinia Riccobene, Alberto Rosti, and Patricia Scandurra. Improving SoC Design Flow by means of MDA and UML Profiles, 2004.

[49] Elvinia Riccobene and Patrizia Scandurra. Model transformations in the UPES/UPSoC development process for embedded systems. *ISSE*, 5(1) :35–47, 2009.

[50] M. Samyn, Samy Meftali, and Jean luc Dekeyser. MDA Based, SystemC Code Generation, Applied to Intensive Signal Processing Applications. In *Forum on specification and Design Languages*, pages 452–463, 2004.

[51] Thorsten Schubert and Wolfgang Nebel. The Quiny SystemC Front End : Self-Synthesising Designs. In *FDL*, pages 135–143. ECSI, 2006.

[52] Fabio Somenzi. Binary decision diagrams. In *Calculational System Design, volume 173 of NATO Science Series F : Computer and Systems Sciences*, pages 303–366. IOS Press, 1999.

[53] Jean-François LE TALLEC and Julien Deantoni. Toward a tlm to rtl refinement : a formal approach. available soon at http ://rtns09.ece.fr, October 2009.

[54] Eugenio Villar, Axel Jantsch, Christoph Grimm, and Tim Kogel. Heterogeneous System-level Specification Using SystemC. In *DATE'08*. IEEE, 2008.

[55] Ying Wang, Xuegong Zhou, Bo Zhou, Liang Liang, and Chenglian Peng. A MDA based SoC Modeling Approach using UML and SystemC. In *Proceedings of the Sixth IEEE International Conference on Computer and Information Technology*, Washington, DC, USA, 2006. IEEE Computer Society.

[56] J. Zhu, I. Sander, and A. Jantsch. HetMoC : Heterogeneous modeling in SystemC. In *Proceedings of the Forum on Design Languages (FDL'2010)*, 2010.

plain references/these