RAPPORT DE PROJET DE FIN D'ÉTUDES

ESISAR 2003/2004

Modélisation et traduction de systèmes sur puce à base de composants

INRIA Rhône-Alpes
Projet POP ART
655 av Europe
38330 Montbonnot Saint Martin

Nicolas Palix

Dates du stage	1 Mars 2004 au 31 Août 2004
Module d'approfondissement	Informatique et réseaux
Tuteur INRIA	Gregor Gößler
Tuteur ESISAR	Philippe Marcel

Remerciements

Je remercie Éric Rutten pour m'avoir accueilli dans le projet Pop Art.

Je tiens à remercier Gregor Gößler, mon tuteur de stage, pour sa confiance, ses commentaires judicieux lors des relectures de ce rapport et sa disponibilité, avant et pendant ce stage.

Je remercie les membres du jury, Philippe Marcel et Yves Guido.

Je remercie les autres stagiaires, les doctorants et post-doctorant pour la bonne ambiance et plus généralement les membres des projets Pop Art, Mistis, Opale et Bip Pop.

Je remercie également Élodie Toïhein, secrétaire du projet, pour l'aspect administratif de mon stage.

Merci à mes parents et Aurore pour leur soutien.

Table des matières

Chapitre 1

Introduction

La complexité physique et logique des systèmes à base de composants électroniques est croissante. Afin de résoudre les contraintes de conception comme la consommation, le coût de fabrication et l'encombrement, les fonctions des différents composants ont été intégrées au sein d'un unique composant physique, de tels systèmes se nomment des systèmes intégrés, SoC (System on Chip). Les SoC résolvent le problème de la complexité physique puisqu'ils permettent de diminuer les pertes par chaleur, augmenter le nombre de puces produites simultanément et occupent moins de place. La maîtrise de la complexité logique, celle liée à l'interconnexion de composants, reste un challenge.

Les concepteurs de SoC cherchent à masquer la complexité pour pouvoir humainement appréhender les problèmes lors de nouveaux développements. Par exemple, les fonctionnalités des composants sont décrites par des programmes dont le langage est orienté matériel. Malgré cela, le niveau de description est encore trop élevé. Des langages synchrones, comme Signal, Esterel ou Lustre, encore plus abstraits sont donc utilisés. Ils permettent de décrire les SoC, qui sont généralement des systèmes réactifs synchrones : un système réactif est un système ouvert répondant constamment aux sollicitations de son environnement en produisant des actions sur celui-ci. L'hypothèse de synchronisme consiste à considérer que chaque réaction d'un système réactif est instantanée. Le système produit ses sorties de manière synchrone aux entrées. C'est comme si le concepteur disposait d'une machine si rapide qu'elle pouvait exécuter les opérations liées au séquencement des instructions, à la gestion des processus, aux communications inter-processus et aux traitements de données élémentaires dans un temps nul (non observable).

Dans ce projet nous nous intéressons à la construction de circuits électroniques modélisés en langage SystemC [12, 15]. SystemC est une librairie de classes implémentée en C++ qui fournit un environnement de développement orienté matériel dans le contexte de C++.

SystemC utilise un modèle de conception et de vérification qui veut couvrir toutes les étapes du concept à l'implémentation, aussi bien en matériel qu'en logiciel. La plateforme propose un modèle interopérable qui veut rendre possible le développement rapide et l'échange de modèles C++ de niveau système.

Dans le cadre de ce projet, nous utiliserons l'implémentation de l'Open SystemC Initiative (OSCI) qui met à disposition les sources de son implémentation. Cette organisation est composée d'entreprises et d'universités et tente d'établir SystemC comme un standard pour la conception de système.

Des entreprises de semiconducteurs, logiciels embarqués, conception électronique automatisée (EDA) et propriété intellectuelle (IP) se sont ainsi regroupées autour de SystemC. Chacun de ces groupes profite de SystemC de différentes manières. Les entreprises de semiconducteurs qui l'utilisent, s'en servent pour la conception et la vérification de leurs produits. Celles de conception électronique automatisée s'en servent comme une plateforme pour développer des outils interopérables. Certaines entreprises de commerce de propriété intellectuelle utilise SystemC comme un moyen de délivrer des modèles interopérables et performant de composants matériels à des niveaux d'abstraction différents.

Les composants permettent de "casser" la complexité des SoC. Pour la maîtriser, la modélisation à base de composants est donc cruciale. Elle permet la séparation des préoccupations et donc des problèmes. Chaque composant est spécialisé et n'effectue qu'une fonctionnalité, mais pas nécessairement qu'une fonction. Les dépendances entre les composants doivent être limitées au stricte nécessaire. En cas d'erreur, les causes possibles peuvent alors être plus facilement identifiées et la faute plus rapidement corrigée.

Le principe de la modélisation à base de composants est également de pouvoir réutiliser les composants rapidement. Il y a donc un enjeu économique important pour les sociétés qui travaillent sur et autour des SoC. Elles auraient alors un outil et une méthodologie permettant un développement rapide de systèmes de plus en plus complexes. La notion de composant étant hiéarchique, un système, vu alors comme un composant, peut être intégré dans un nouveau système.

Mais la complexité se retrouve lors de la composition. Il est donc important d'avoir des méthodes formelles permettant de garantir le fonctionnement du système après composition. Les méthodes formelles offrent l'avantage d'être généralement associées à des outils qui les implémentent. Ceux-ci sont plus rapides que les méthodes manuelles pour effectuer une vérification. De plus, ils reposent sur un base mathématique qui garantit leur validité contrairement aux méthodes manuelles souvent empiriques.

Les systèmes à transitions et les automates d'états finis sont les outils privilégiés pour la modélisation formelle. Ils sont souvent utilisés pour la validation des systèmes. Il existe plusieurs types de validation qui sont complémentaires, la simulation, le test, la vérification de modèle (model-checking).

Actuellement, l'utilisation de la modélisation à base de composants est limitée par le manque de résultats pour garantir la correction du système composé.

Le projet s'inscrit dans un travail de recherche mené conjointement à l'INRIA, au projet PopArt, et au laboratoire Vérimag, dans le cadre duquel une méthodologie de modélisation et vérification formelle basée sur le formalisme des systèmes à transitions avec contraintes a été établie. La plupart des résultats sont utilisables grâce à l'outil Prometheus [8] qui permet de vérifier certaines propriétés nécessaires au bon

fonctionnement d'un programme comme le non blocage.

Afin de garantir la correction du système, il n'est en général pas envisageable de vérifier la correction du système composé, à cause de la complexité de ce dernier. La technique de modélisation compositionnelle permet de garantir la correction du système composé à partir des propriétés de ses composants. Une propriété à montrer est, par exemple, que l'assemblage de composants sans blocages reste sans blocage et ce uniquement à partir de la description et des propriétés des composants d'origine.

La vérification compositionnelle est donc l'application de cette méthode sur des systèmes à composants. Il s'agit de vérifier à partir des composants et de leurs propriétés que les règles de composition employées ne vont pas conduire à une malformation du système.

Chapitre 2

Vocabulaire

Système sur puce (Un) est un systèmes issus de la micro-électronique. Il s'agit généralement d'intégration de systèmes existant. La dénomination anglaise est System on Chip (SoC).

Système en couche à base de composants (Un) est un ensemble de systèmes à transitions définissant des composants et leur comportement propre (couche 1). Un ensemble d'interactions lié ces composants entre eux et forme le comportement du système global (couche 2). Enfin des contraintes sur l'état du système et sur les interactions raffinent les comportements possibles du système (couche 3). Pour faciliter la lecture, il arrivera de nommer un tel système, modèle à composants ou modèle en couches.

Système à transitions (Un) est modélisé par un graphe. Les sommets représentent les états possibles du système tandis que les arcs représentent des transitions entre certains états.

Les transitions sont appelées dans le modèle considéré des actions et une garde contrôle la réalisation de la transition. Si celle-ci est franchie, une fonction de transition modifie l'état du système.

Interaction (Une) est un ensemble non-vide d'actions liées devant être réalisées ensembles lors de l'évolution du système.

Chapitre 3

Présentation de l'établissement

3.1 INRIA

Créé en 1967 à Rocquencourt près de Paris, l'INRIA (Institut national de recherche en informatique et en automatique) est un établissement public à caractère scientifique et technologique (EPST) placé sous la double tutelle du ministère de la recherche et du ministère de l'économie, des finances et de l'industrie.

Le budget total est de 119,4 M Euros. Bien qu'il s'agisse d'un établissement public, 23% de ce budget est issu de ressources propres.

Cet institut dispose de ressources humaines (3 200 personnes dont 2 400 scientifiques) aussi bien internes que externes et à court, moyen et long terme dont voici la répartition à titre indicatif :

- titulaires INRIA : 900 (400 chercheurs, 500 ingénieurs et techniciens)
- 130 post-doctorants, 300 stagiaires, 320 invités
- doctorants : 800
- chercheurs et enseignants d'autres organismes : 450
- "ingénieurs experts" (sur contrat de recherche) : 200

Il existe quelques indicateurs témoins de la vivacité de l'institut. Il y a par exemple environ 2 300 publications scientifiques, plus de 800 contrats de recettes actifs et plus de 300 contrats de recettes signés dans l'année. Une soixantaine de sociétés sont issues de l'INRIA, depuis Ilog, aujourd'hui cotée au Nasdaq, jusqu'aux toutes dernières, 12 en 2000, 4 en 2001, 3 en 2002 et 7 en 2003. L'INRIA disposait en 2003 de 150 brevets actifs, 50 licences payantes de logiciels et 95 licences de logiciels libres étaient actives en décembre 2003. Plus de 150 logiciels sont disponibles en accès gratuit sur le site de l'INRIA ou à travers la diffusion d'un CD ROM.

Depuis sa création, d'autres établissements dépendants ont été ouverts à travers le territoire français. Il y en a actuellement 6 en France :

- Siège et Unité de recherche Rocquencourt
- Unité de recherche Futurs (Lille, Saclay, Bordeaux)
- Unité de recherche Lorraine - Nancy
- Unité de recherche Rennes - INRIA Rennes / Irisa

– Unité de recherche Sophia Antipolis
– Unité de recherche Rhône-Alpes - Montbonnot Saint-Martin

3.2 INRIA Rhône-Alpes

Créée en décembre 1992, l'unité de recherche INRIA Rhône-Alpes regroupe environ 420 personnes réparties sur trois sites : la Zirst de Meylan-Montbonnot, le campus universitaire de Grenoble et le site technopolitain de Lyon (dont Lyon-Gerland et le domaine scientifique de la Doua).

265 scientifiques participent à 24 équipes (21 projets de recherche), dont 16 sont en partenariat avec :
– le CNRS,
– l'université Joseph Fourier et l'Institut national polytechnique de Grenoble (laboratoire Gravir et ID),
– le CNRS et l'École nationale supérieure de Lyon (laboratoire LIP),
– l'université Pierre Mendès-France,
– l'universitté Claude Bernard et l'Institut national des sciences appliquées (INSA) de Lyon

Six équipes de recherche sont totalement ou partiellement localisées à Lyon.

L'INRIA Rhône-Alpes mène ses activités en étroite collaboration avec les laboratoires de recherche publics et privés, nationaux et internationaux, et elle entretient des liens privilégiés avec l'institut d'Informatique et Mathématiques Appliquées de Grenoble (IMAG).

3.3 Le projet Pop Art

Il s'agit du projet dans lequel j'ai effectué mon projet de fin d'étude. Il se trouve dans les locaux de l'INRIA Rhône-Alpes sur le site de Montbonnot.

Il est issu de l'évolution du projet BIP qui, en prenant de l'ampleur, a été scindé en deux projets distincts :
– branche contrôle-commande temps réel : POP ART
– branche automatique et bipède : BIPOP

Pop Art est l'acronyme anglais de "Programming and Operating systems for Applications in Real Time" mais le thème de recherche du projet n'est pas une traduction littérale puisqu'il s'agit de "Contrôle-commande temps réel sûr".

L'équipe du projet est composé de :
– 5 permanents
– 1 post-doctorant
– 2 membres extérieurs
– 2 doctorants

– 4 stagiaires

La conception sûre de systèmes embarqués pour le contrôle-commande temps réel concerne des domaines aussi variés que les transports (avionique, ferroviaire, automobile), production, médical, énergie, télécommunication, ...

L'objectif est d'offrir une aide à la conception des systèmes donc la complexité augmente. De plus, cette aide veut prendre en compte plusieurs aspects du système étudié et lui assurer des propriétés de sûreté critique. Il devient vital de prendre en considération l'aspect sûreté de fonctionnement dans le logiciel car il tend à se trouver dans tous les systèmes modernes y compris ceux liés à des vies humaines ou des enjeux financiers importants.

L'utilisation des modèles formels permet d'avoir des méthodes automatisées utilisable par des utilisateurs experts des applications et leurs masquer ainsi la complexité qu'ils gérent.

Actuellement, des techniques diverses de génération automatique d'exécutifs sont utilisés :
- Middleware, schémas de code, bibliothèques
- Langages, compilation, propriétés statiques
- Analyse, synthèse sur les propriétés dynamique
- Modélisation et analyse compositionnelle
- Programmation orientée aspects

Elles sont appliquées aux problématiques rencontrées :
- Mises en oeuvre distribuées et tolérantes aux fautes de programmes synchrones
- Conception conjointe commande/ordonnancement, ordonnancement régulé
- Génération automatique de contrôleurs corrects, appliquant la synthèse de contrôleurs
- Vérification de système à composants

Ce stage met en oeuvre les techniques de compilation et de modélisation et d'analyse compositionnelle en vue de vérifier un système à composants.

Chapitre 4

Cahier des charges

Lors de ce stage, une étude bibliographique nous permettra de se familiariser avec SystemC, les systèmes à transitions et la vérification formelle compositionnelle. Il s'agit ensuite de proposer et d'implanter un prototype d'outil pour la traduction du langage SystemC vers les systèmes de transitions avec contraintes. Enfin, nous vérifierons la validité des résultats obtenus avec l'outil.

Cet outil est donc un compilateur permettant, après analyse d'un programme écrit en SystemC, de générer un programme Prometheus. Le programme SystemC doit pouvoir faire appel à la librairie TLM écrite par la société STMicroelectronics.

Le niveau d'abstraction des systèmes compositionnels en couche à base de systèmes à transitions avec contraintes étant plus élevé que SystemC, il sera nécessaire de faire certaines abstractions dans l'interprétation du programme SystemC. Les problèmes présents dans le programme doivent cependant être signalés aux utilisateurs.

Les abstractions introduiront éventuellement une erreur lors de la vérification des propriétés telles que le non-blocage, la vivacité ou le déterminisme. Mais ces abstractions sont pessimistes, elles ne doivent pas masquer une propriété qui n'est pas vérifiée dans le programme d'origine. Il s'agirait alors de faux négatif et l'utilisateur devrait alors vérifier que, par exemple, le blocage détecté est lié à une abstraction du modèle et non pas à une réelle erreur de conception.

Pour que la tâche soit réalisable dans le temps imparti, nous nous limiterons au langage SystemC en utilisant si possible un analyseur syntaxique pour SystemC déjà développé[18]. La librairie de STMicroelectronics sera ensuite prise en compte.

Chapitre 5

État de l'art

5.1 La modélisation des systèmes électroniques sur puce

5.1.1 Flôt de conception

Dans le schéma 5.1, proposé par [16], les phases généralement employées lors d'important développement de systèmes électroniques embarqués sont données. Les spécifications sont réalisées dans un langage humain. Pour éviter les erreurs de compréhension, il peut être "standardisé", utilisation d'un vocabulaire spécifique associé à une sémantique particulière. Pour garantir une meilleur compréhension, des langages formels de description comme SDL, Specification and Description Language ou UML, Unified Modeling Language, peuvent également être employés .

Lors de la conception du système, l'utilisation d'un programme en langage de haut niveau tel que C ou C++, permet de valider les concepts exprimés lors des spécifications.

Le niveau transfert de registre, RTL, correspond à une description des circuits à l'aide de langage de description du matériel, HDL (Hardware Description Language) tel que ISP, UDLI, Verilog ou VHDL.

Enfin, des outils automatiques permettent de générer le masque de circuit comportant les pistes et les transistors. Ils utilisent des techniques de compilation et effectuent des opérations d'optimisation et de simplification.
Le principe des HDL est de décrire de façon naturelle les circuits électroniques. Ceux-ci sont donc représentés sous forme de modules et une séparation est faite entre le comportement et l'architecture. Mais tout l'intérêt des HDL est de pouvoir les exprimer à l'aide d'un même langage. L'aspect comportemental est cependant moins aisément décrit qu'à l'aide de langages tel que le C ou le C++.

Le principe est de décrire un **modèle**, aussi appelé une **architecture**, qui représente le système à étudier. Celui-ci est constitué de **composants**, de **signaux** et de **ports**. Ces éléments peuvent eux-mêmes être vus comme des modèles plus simples.

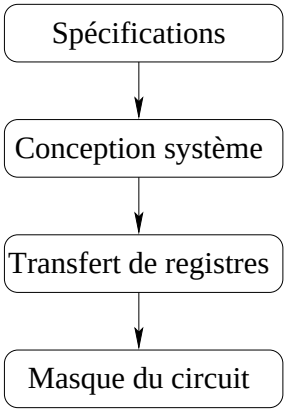


FIG. 5.1 – Les différents niveaux de conception d'un système électronique

Des modèles de base forment les éléments triviaux d'une construction récursive.

Un modèle hiérarchique est ainsi obtenu. Les composants internes sont des fils et le modèle est leur père : le père **instantie** les modules fils. Les systèmes sont souvent représentés sous forme graphique à l'aide d'imbrication ou d'arbre afin de faire ressortir l'aspect hiérarchique du système.

Afin d'accroître la vitesse de développement des systèmes électroniques embarqués, et de valider le plus tôt possible les concepts de haut niveau, il est nécessaire d'avoir des outils qui permettent de les exprimer de façon rapide. Il est également intéressant de pouvoir utiliser ces mêmes outils pour valider les systèmes et les générer avec un niveau de détails plus élevé. Il est alors possible de mettre en place des outils qui vérifient la validité des systèmes décrits. Le but à long terme est d'avoir une chaîne automatisée de la description des spécifications jusqu'au masque de circuit. Elle devra disposer d'outil de vérification et de phase d'optimisation.

5.1.2 Modélisation à l'aide de SystemC

SystemC étend le concept d'architecture modulaire au langage C++ en ajoutant de la syntaxe supplémentaire. Il s'agit d'un ensemble de macros et de classes qui permettent une description aisée des systèmes électroniques synchrones. Ils peuvent être décrits à plusieurs niveaux :

Behavioral Level Modeling - BLM Seul le comportement des composants est implémenté. Les communications sont effectuées par des transactions qui permettent l'échange structuré de données. Les variables peuvent être définies sur des domaines non optimaux, plus grands que nécessaire, et les échanges sont synchrones.

Hardware Oriented Data Types Ce niveau de conception est intermédiaire. Il ne modélise pas fidèlement les échanges mais prend déjà en compte les types de données et leur structure telle qu'elle doit être implantée matériellement. On trouve à ce niveau de description la taille des registres par exemple. L'usage de

bit et de vecteur de bit est également introduit ainsi que les valeurs logiques à trois états.

Register Transfert Level Modeling - RTL A ce niveau de conception, les transactions disparaissent au profit des bus d'adresse, de contrôle et de données. Les variables internes au composant sont revues et mieux adaptées. On détermine à ce niveau la taille des registres et leur organisation au sein du composant.

SystemC veut uniformiser la méthode de description utilisée dans les entreprises afin de supprimer les problèmes d'hétérogénéité liés à l'usage des différents HDL existants et rentre la description des systèmes uniformes entre les niveaux comportemental et architectural.

La librairie SystemC permet de générer des simulateurs de composants électroniques. Elle fournit un environnement à composants, les **modules**. Ceux-ci peuvent être assemblés et communiquent par signaux via des ports. Les composants contiennent des **processus** et peuvent être sensibles à des signaux extérieurs. A la compilation, un simulateur est généré et exécute le système de manière synchrone en apparence, le programme généré étant séquenciel.

La figure 5.2 permet d'illustrer comment sont représentés une architecture et les éléments qui la composent, les signaux et les canaux pour la communication, et les modules avec leurs ports et leurs interfaces. Lors de la conception d'une architecture, il y a une partie d'initialisation qui permet :
- d'instancier les composants et les signaux,
- de les connecter,
- de mettre les signaux dans leur état initial.

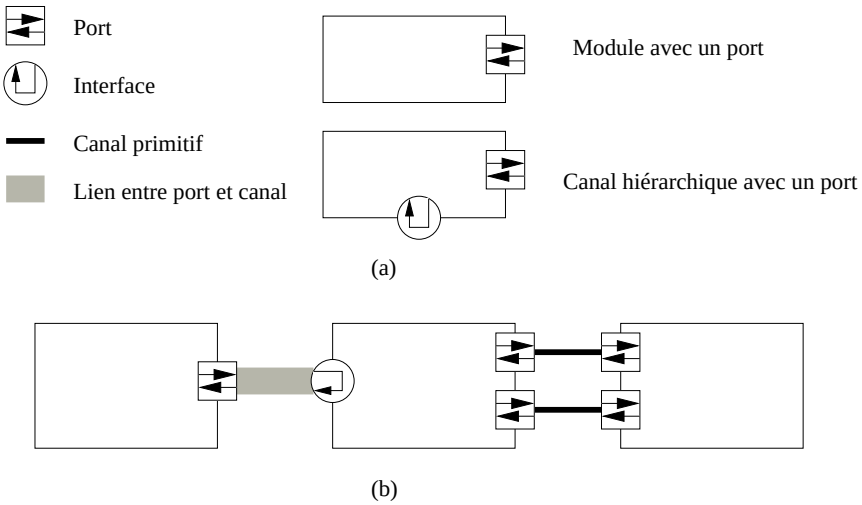


FIG. 5.2 – Représentation graphique en SystemC – (a) Notation graphique des éléments de bases – (b) Une architecture simple

Cependant SystemC peut paraître trop généraliste puisqu'il permet de décrire un même système aussi bien au niveau du comportement qu'au niveau physique. La

société STMicroelectronics utilise donc une librairie pour standardiser la modélisation de niveau système, description du comportement, lors de ses développements.

5.2 La vérification formelle

La vérification formelle consiste à rechercher des propriétés sur un modèle mathématique du système physique. Ce modèle est plus ou moins abstrait selon le raffinement nécessaire pour montrer la propriété désirée.

La vérification formelle est basée sur une description formelle qui est supposée fidèle au système. Les moyens de description formel sont, par exemple, le langage SDL (Specification and Description Language), les méthodes B et Z, les systèmes à transitions et les automates.

Parmi les méthodes de validation des systèmes, nous citons :
- la simulation,
- le test,
- la vérification de modèle (model-checking).

La simulation permet de valider tôt dans le processus de conception un système puisqu'elle peut souvent être effectué par des outils généralistes avant même le début de la réalisation. Elle est utilisée pour valider la description d'un système ou sa conception. Il s'agit d'un modèle exécutable qui peut donc fournir une trace d'exécution.

Le test veut montrer qu'un système est bien formé, c'est à dire le valider par rapport aux besoins et le vérifier par rapport au cahier des charges initial. Comme il n'est pas possible de tester l'ensemble des combinaisons des séquences d'entrées, l'étude se restreint à un aspect de la description. Cela peut par contre couvrir un ensemble de cas de test. Les tests sont effectués sur l'ensemble des aspects critiques des systèmes étudiés.

Enfin la vérification de modèle consiste à construire un modèle du système, une représentation abstraite, et vérifier des propriétés sur ce modèle comme par exemple la sûreté, l'atteignabilité et la vivacité.

Les outils de vérification formelle construisent, par exemple, un automate d'états finis et cherchent un chemin d'un état vers un autre état pour vérifier une propriété. Pour faciliter cette recherche, une étape de simplification et de réduction est appliquée.

5.2.1 Vérification compositionnelle

Les systèmes sont de plus en plus complexes et il devient illusoire de vouloir les vérifier dans leur globalité. La vérification compositionnelle part du fait qu'il est souvent plus judicieux de vérifier le système, en vérifiant uniquement ses composants d'une part, et ses interactions et contraintes d'autre part et d'inférer la correction

du système composé, des propriétés des composants.

Des résultats sont actuellement disponibles sur la sûreté de fonctionnement et le non-blocage. Ces propriétés sont inférées de l'opération de composition et du modèle utilisé comme expliqué dans [11].

5.3 Les outils de la vérification de systèmes à composants

Nous allons à présent voir deux outils de vérification de systèmes à composants :
- celui développé dans le cadre de la collaboration entre STMicroelectronics et Vérimag,
- l'outil de vérification compositionnelle Prometheus.

5.3.1 L'outil STMicroelectronics - Vérimag

Une chaîne de vérification [18] est actuellement développée par STMicroelectronics et Verimag. Elle est basée sur l'analyse de code SystemC et TLM et utilise l'analyseur syntaxique de GCC qui a modifié ainsi qu'une version modifiée de SystemC. Après l'analyse, le programme construit des automates qui décrivent le comportement du programme analysé. Ces automates sont produits à partir de l'arbre abstrait issu de GCC. Les automates abstraits sont ensuite mis en forme en Lustre par un générateur de code. Le programme Lustre est alors analysé par l'outil Lesar ou converti avec Lus2nbac pour l'outils Nbac qui permettent de vérifier les propriétés désirées.

Lustre [14, 13] est un langage synchrone pour décrire les systèmes réactifs synchrones. Lesar et Nbac sont deux outils de vérification formelle de modèles abstraits. Lesar travaille sur les programmes Lustre tandis que Nbac travaille sur un format différent mais inspiré de Lustre. Nbac permet l'analyse de système réactifs synchrones et déterministes. Ces systèmes contiennent des combinaisons de booléens et de variables numériques. Les systèmes asynchrones et/ou non-déterministes peuvent être compilé dans le modèle Nbac.

Chaque processus représente un automate. Les types d'objets SystemC permettent de définir des modèles d'automate. Ces modèles sont paramétrés par les informations recueillies lors de l'analyse du programme SystemC.

Quelques exemples sont fournis avec le programme qui est encore en développement. La première mise à disposition est prévue pour début septembre. Le code source de la librairie TLM est disponible depuis début avril 2004.

5.3.2 Prometheus

Prometheus est une implémentation de la vérification compositionnelle. Il applique cette méthode sur des systèmes en couche à base composants.

Les algorithmes utilisés sont ceux de vérification formelle par composition donnés dans [11, 10].

Il s'agit d'un outil expérimental qui est encore en développement mais il fournit déjà des résultats sur le comportement du système composé. Prometheus permet d'analyser des propriétés sur les programmes tel que le non blocage, la vivacité et le déterminisme. Plusieurs types de non blocage sont vérifiés :

- Non blocage du système composé : Il s'agit de vérifier qu'il existe toujours une interaction réalisable dans le système.
- Non blocage d'un composant dans la composition : Il s'agit de vérifier que aucun des composants n'est bloqué par composition et que malgré les interactions, le composant possède toujours une interaction réalisable.

Chapitre 6

Cadre de modélisation

Nous allons dans un premier temps voir le cadre de modélisation offert par SystemC puis le cadre formel des systèmes à transitions [11]. Dans le chapitre 8 nous verrons comment passer d'un modèle à l'autre.

6.1 La modélisation en SystemC

La structure de la librairie SystemC est donnée par la figure 6.1. Elle s'insère au-dessus de C++ par rapport à l'utilisateur et c'est une structure extensible, l'utilisateur peut ajouter ses propres modèles de composants.

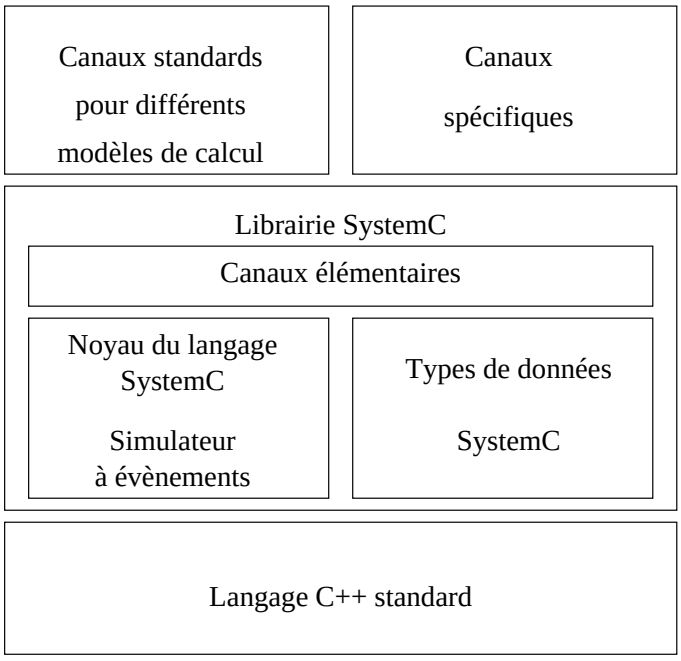


FIG. 6.1 – Architecture de SystemC

La modélisation en SystemC présenté ici est basée sur celle présentée dans [12, 16, 18]. Afin de clarifier le discours, un exemple très simple est donné. Il servira tout au long de cette explication.

```
1  SC_MODULE(emitter) {
2    sc_out<bool> my_output;
3
4    void emitter_process() {
5      int y = 0;
6      while (y < 5) {
7        my_output.write(true);
8        wait(10, SC_US);
9
10       scp_begin_critical_section(name());
11       y++;
12       // Working 10 ms...
13       wait(10, SC_MS);
14       scp_end_critical_section(name());
15
16       my_output.write(false);
17       wait(10, SC_US);
18     }
19   }
20
21   SC_CTOR(emitter) {
22     SC_THREAD(emitter_process);
23   }
24 };
25
26 SC_MODULE(receiver) {
27   sc_in<bool> my_input;
28
29   void receiver_process() {
30     while(true) {
31       scp_begin_critical_section(name());
32       wait();
33       scp_end_critical_section(name());
34       wait();
35     }
36   }
37
38   SC_CTOR(receiver) {
39     SC_THREAD(receiver_process);
40     sensitive << my_input;
41   }
42 };
43
44
45 emitter  *my_emitter;
46 receiver *my_receiver;
```

```
47  sc_signal<bool> wire;
48
49  int sc_main(int argc, char *argv[]) {
50    my_emitter = new emitter("my_emitter");
51    my_receiver = new receiver("my_receiver");
52
53    my_receiver->my_input(wire);
54    my_emitter->my_output(wire);
55
56    wire.write(false);
57
58    sc_start(-1);
59    return 0;
60  }
```

Après l'initialisation de la librairie, la fonction `sc_main` (ligne 49) du système décrit par l'utilisateur est appellée. Il s'agit de l'instantiation de l'architecture (lignes 45 à 47). La première phase, **l'élaboration**, instancie (lignes 47, 50 et 51) tous les composants du système, les connecte (lignes 53 et 54) et les initialise (ligne 56). Le système de l'utilisateur rend ensuite la main à la librairie en appellant la fonction `sc_start` (ligne 58). La deuxième phase, **la simulation**, débute avec l'exécution de l'ordonnanceur. Elle se termine soit à la fin des processus, soit par l'interruption de l'utilisateur, soit parce que la durée de la simulation a été atteinte.

Les types de C++ restent disponibles ainsi que les types construits par l'utilisateur. SystemC fournit cependant des types construits comme les valeurs logiques à quatre états, les vecteurs de valeurs logiques, les bits et vecteurs de bits, les nombres à virgule fixe, les entiers à précision arbitraire. Tous les objets SystemC dérivent de la classe primitive `sc_object`.

Parmis les nouveaux mots clés, ce trouve `SC_MODULE` (lignes 1 et 26) qui permet d'identifier un module SystemC. C'est une macro qui sert à déclarer une classe C++. Chaque module est un objet du système. Il existe d'autres types d'objet SystemC comme `sc_signal, sc_in, sc_out` (lignes 2, 27 et 47) par exemple pour les signaux et les ports. Des composants plus évolués mais courants sont également disponibles tels que des files, des piles, des sémaphores. Le nouvel objet SystemC créé avec `SC_MODULE` peut être un canal hiéarchique, voir la figure 5.2 page 15, par opposition aux canaux primitifs, ou un module classique qui modélise un composant électronique comme les composants `emitter` et `receiver` de notre exemple.

Un module est constitué d'un identifiant et d'une initialisation et potentiellement de ports, de processus, de signaux, de modules.

Les modules utilisent généralement les mots clés suivants :

- `SC_HAS_PROCESS`
- `SC_CTOR` (lignes 21 et 28) ou le constructeur C++ équivalent.
- `sensitive` (ligne 40)
- `SC_THREAD` (lignes 22 et 29)
- `SC_CTHREAD`
- `SC_METHOD`

`SC_CTOR` est le constructeur d'une classe `SC_MODULE`. Il marque le code qui instancie les composants internes et les lie entre eux. Lorsque le développeur n'utilise pas `SC_CTOR`, il doit utiliser le mot clé `SC_HAS_PROCESS`, voir page 24, qui permet à la librairie SystemC d'initialiser l'ordonnanceur.

La liste des processus se trouve dans le corps du constructeur. Ils peuvent être de 3 types : `SC_THREAD, SC_METHOD, SC_CTHREAD`. Le mot clé `sensitive` qui suit indique sur quel(s) signal(s) est sensible le dernier processus déclaré. Cette liste de sensibilité ne s'applique qu'à lui. Le dernier type est sensible sur une horloge.

Les processus de type `SC_METHOD` sont appelés à chaque modification de l'état du signal auquel ils sont sensibles tandis que les processus `SC_THREAD` sont exécutés une seule fois en début de simulation et contiennent généralement des appels à la méthode `wait()` ainsi qu'une boucle infinie telle que `while(true)` (lignes 29 à 36). Dans notre exemple, le modèle du composant `receiver` est déclaré `sensitive` sur le `sc_signal wire` connecté au port `sc_in my_input` du composant en question. A chaque modification du signal, dans un δ-pas, une notification interne à SystemC permet de débloquer les appels `wait` (lignes 32 et 34) du processus sensible `receiver_process`.

L´état interne des composants est stocké à l'aide de variables. Et les opérations d'entrées et sorties entre composants doivent transitées par des ports reliés par des signaux. Si cette condition n'est pas respectée, l'architecture est dite mal formée. En d'autre terme, le fait d'utiliser des mécanismes C++ comme la mémoire partagée sont à proscrire bien qu'ils soient encore possible puisque l'environnement C++ reste disponible.

6.1.1 L'ordonnanceur

L'ordonnanceur du simulateur, illustré en 6.2, est à événements. Il est le cœur de la librairie. L'évolution des composants est constituée de δ-pas, qui forment un δ-cycle. Ces deux concepts modélisent un temps simulé nul.

À la fin d'un δ-pas, l'ordonnanceur regarde quels sont les événements en attente puis choisit un processus éligible. Son exécution est réalisée, c'est à dire qu'il effectue un δ-pas. Des notifications immédiates peuvent avoir eu lieu pendant ce δ-pas. Des processus peuvent alors être immédiatement (ré)évalué pour le prochain δ-pas.

Une boucle infinie existe s'il existe un cycle de notification immédiate entre les processus. Il s'agit là d'un comportement non désiré. Une infinité de δ-pas est alors réalisée sans jamais finir le δ-cycle.

Lorsque aucun processus ne peut évoluer plus, les notifications δ-délai pendantes sont émisses aux modules qui en dépendent et une mise à jour des événements et des signaux est effectuée. L'ordonnanceur choisit alors un nouveau processus éligible en fonction des notifications δ-délais. Le temps simulé n'a toujours pas évolué à ce moment là alors que les processus ont évolué.

Lorsque aucun processus ne peut évoluer plus, il n'y a alors plus de notifications immédiates ou δ-délai pendantes, le temps simulé évolue jusqu'au prochain événement. C'est le début d'un nouveau δ-pas qui commence par une évaluation des

processus éligibles.

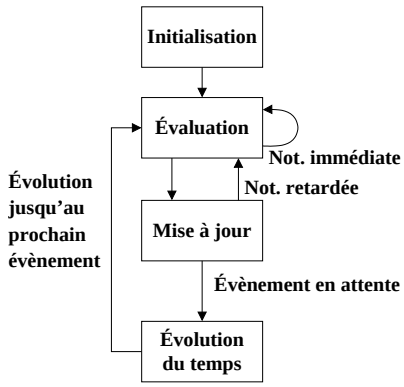


FIG. 6.2 – Représentation des δ-cycles

6.1.2 Les événements

Les composants SystemC peuvent communiquer par événements. Certains notifient des événements pendant que d'autres les attendent. Les notifications utilisées sont celles que gère l'ordonnanceur, notification immédiate, δ-délai (ou temps nul) et temporisée. Pour un événement `e`, SystemC définit respectivement les opérations `e.notify(), e.notify(SC_ZERO_TIME), e.notify(t)`. Les processus se mettent en attente grâce à l'instruction `wait`. La liste de sensibilité de l'opération peut être statique (ligne 40) ou dynamique avec des constructions tels que `wait(`e_1 `&` e_2`)` ou `wait(`e_1 `|` e_2`)`. Les événements sont modélisés par l'automate de la figure 6.3, conformément à la sémantique SystemC.

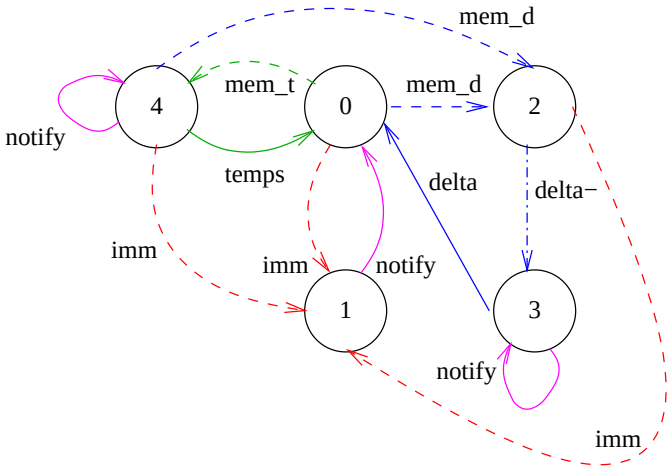


FIG. 6.3 – Comportement d'un événement

6.1.3 Les signaux

Pour les signaux, la modélisation présentée dans [18] est utilisée comme base. Le schéma 6.4 illustre le comportement d'un signal. Le couple (C, N) caractérise le signal, C est la valeur courante du signal et N la valeur future. Les notifications (action **NOTIFY_d**) ne sont émisses aux composants attachés en sortie des signaux que lorsque la valeur du signal est modifiée (action **WR**). Le signal doit avoir pour notifier des valeurs courante et future différentes. L'atomicité de l'exécution d'un pas et le temps nul représenté par un δ-cycle impliquent que seul la dernière écriture d'un même δ-cycle est prise en compte. Le franchissement de l'action **delta** met à jour la valeur courante du signal et marque le début d'un nouveau cycle. La valeur du signal est lue par l'action **RD** qui est toujours réalisable. La valeur lue est toujours la valeur courante, voir la partie inférieure du schéma 6.4. Ainsi lorsqu'une écriture et une lecture ont lieu sur un même signal dans un même cycle, elles n'interfèrent pas. L'ordonnanceur peut alors indifféremment exécuter l'un ou l'autre des processus. La mise à jour ayant eu lieu à la fin de la notification, les composants notifiés peuvent ensuite effectuer une opération de lecture qui lira la valeur mise à jour.

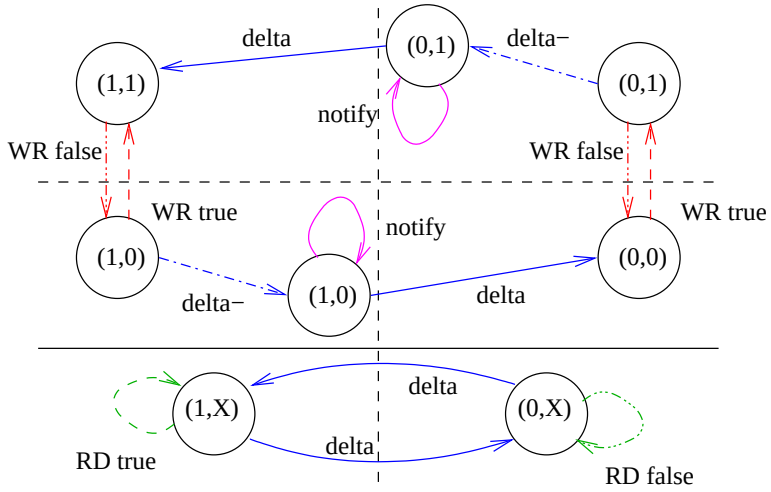


FIG. 6.4 – Comportement d'un signal

6.1.4 Cas spécifique de TLM

Il existe un autre cas de figure vu lors de l'utilisation de la librairie TLM utilisée à STMicroelectronique. Dans l'exemple de base fournis, le composant `signal_slave` contient des processus (utilisation de `SC_HAS_PROCESS`) mais n'en déclare aucun ! En fait, l'utilisation du mécanisme de transaction, propre à TLM, fait que l'appel à une méthode (`read` ou `write`) sur le port d'un composant génère un appel à la même méthode sur le composant cible, identifié par un canal et une adresse.

TLM est l'acronyme de Transaction Level Model. L'analyse se place ici au niveau description du comportement, voir BLM à la section 5.1.2, page 14.

C'est donc des structures de données qui transitent entre les composants par des canaux plutôt que des échanges par signaux booléens et entiers. La librairie fournit

donc un ensemble de ports et d'interfaces SystemC adaptés.

6.2 Les systèmes à transitions

6.2.1 Définitions

Voyons à présent le modèle formel mis en place dans [11].

Définition 1 (Système à transitions). *C'est le tuple* $(X, IC, \{G^a\}_{a\in IC}, \{F^a\}_{a\in IC})$, *où*

- X *est un ensemble fini de* **variables booléennes**,
- IC *est un vocabulaire fini d'***interactions**,
- G^a *est un prédicat sur* X *appelé* **garde**,
- $F^a : X \to X$ *est une* **fonction de transition**.

Définition 2 (Sémantique d'un système à transitions). *Un système à transitions* $(X, IC, \{G^a\}_{a\in IC}, \{F^a\}_{a\in IC})$ *définit une relation de transition* $\to$: $X \times IC \times X$ *si* $\forall x, x' \in X, \forall a \in IC$, *on a* $x \xrightarrow{a} x' \Leftrightarrow (G^a(x) \wedge (x' = F^a(x)))$

G^a est la garde sur l'interaction a. Elle dépend de la valeur de x qui définit l'état du processus. x' est l'état après l'exécution de a et est déterminé par la fonction de transition de a appliquée à l'état x.

Passer de l'état x à l'état x' en exécutant l'interaction a est possible si $G^a(x)$ est vraie, c'est à dire que a est réalisable dans l'état x, et que $x' = F^a(x)$, c'est à dire que x' est bien l'état du système après exécution de a alors que le système était dans l'état x.

Une version simplifiée du modèle d'interactions est utilisée, le détail est en [11], en particulier la définition d'interaction.

Définition 3 (Modèle d'interactions). *Le modèle d'interactions d'un système composé d'un ensemble* K *de composants est un couple* $IM = (IC, IC^-)$ *où* $IC^- \subseteq IC$, *est l'ensemble des interactions incomplètes tel qu'il ne contient pas d'interaction maximale et* $\forall b, b' \in IC, b \in IC^-$ *et* $b' \subseteq b$, *on a* $b' \in IC^-$.

IC^+ *et* IC^- *forment une partition de* IC. IC^+ *sera utilisé pour désigner des interactions complètes.*

Il faut ajouter à cela les contraintes liées à la composition des composants et aux interactions. Les interactions IC^- sont illégales. Les éléments IC qui sont des compositions d'interactions et ceux de IC^-, qui sont des interactions simples, cf figure 6.6, doivent être explicitement déclarés. Pour le cas de la figure 6.7 (c), les interactions `out|in1|in2, out|in1, out|in2` sont déclarées complètes, `out` est implicitement complète, les interactions `in1, in2` sont déclarées incomplètes et l'interaction `in1|in2` est implicitement incomplète.

Ce modèle relativement riche peut être utilisé avec SystemC en conservant plus de détails qu'avec un modèle de système à transitions basiques tel que celui utilisé dans [16] par exemple.

Le système donné en exemple en 6.1 peut être représenté par les automates de la figure 6.5. Une interaction est associée à chaque arc et un état est associé à chaque sommet. Les composants sont ici isolés, la seule contrainte sur les transitions est d'être dans l'état à l'origine de l'arc. Les fonctions de transitions sont données entre {} .

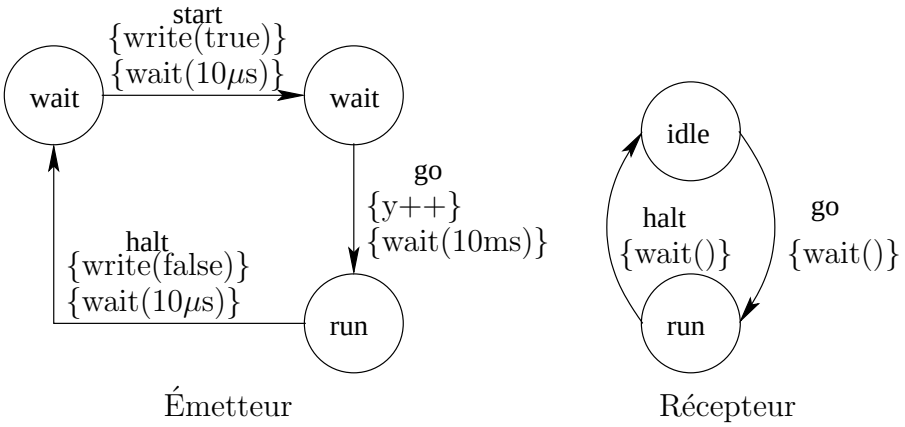


FIG. 6.5 – Exemples d'automate

À partir des éléments de base de la légende 6.6, les schémas 6.7, page 27, sont donnés pour illustrer le modèle d'interactions et comment sont formés les ensembles IC et IC^-. Les principales possibilités y sont représentées : rendez-vous binaire bloquant (a), rendez-vous binaire non-bloquant (b) et diffusion non-bloquante (c).

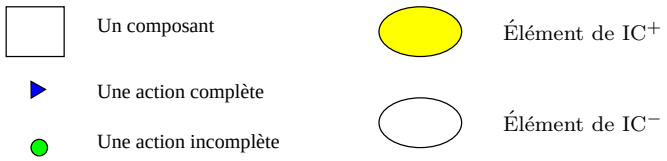


FIG. 6.6 – Éléments de base du modèle d'interaction

Une interaction est réalisable dans Prometheus si sa garde est satisfaite, ainsi que les contraintes du système avant et après l'interaction, notées respectivement $pre(U)$ et U.

Définition 4 (Contrainte). *Une contrainte sur un système à transitions* $B = (X, IC, \{G^a\}_{a\in IC}, \{F^a\}_{a\in IC})$ *est un terme de la forme* $U^X \wedge \bigwedge_{a\in IC} t^a(U^a)$*, où :*

- U^X *est un prédicat d'état sur* X*,*
- $t^a(U^a)$ *est un prédicat d'interactions tel que* $\forall x \in X$*, on est* $t^a(U^a)(x)$ *si* $G^a(x) \Rightarrow U^a(x)$*.*

Afin de garantir le déterminisme et la confluence locale, il est utile de disposer une relation d'ordre entre les interactions qui établit une relation de priorité. Il s'agit d'une nouvelle contrainte car une interaction est irréalisable si une interaction réalisable la domine.

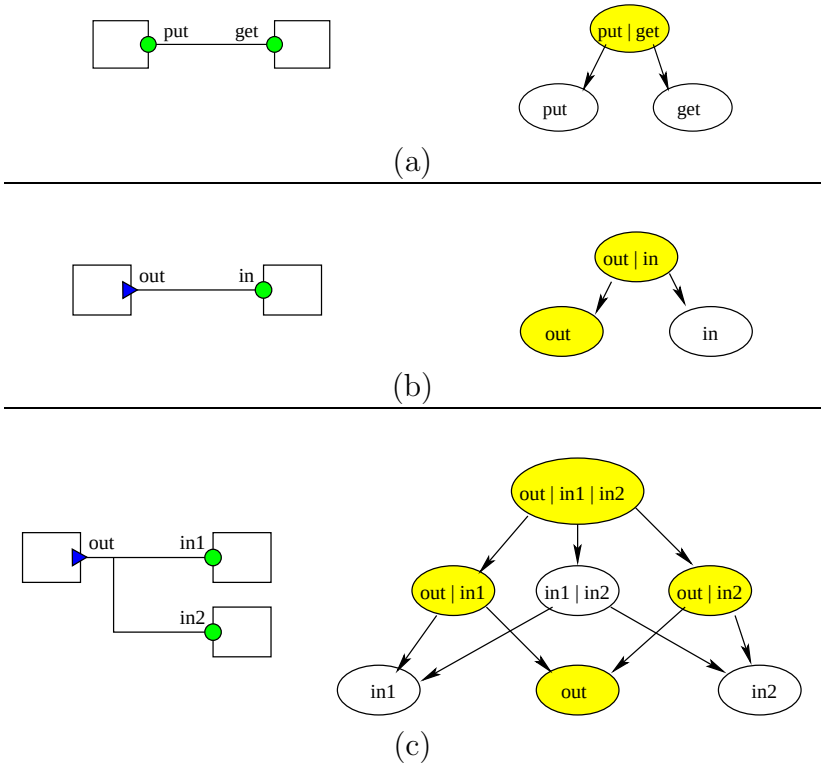


FIG. 6.7 – Les principaux modèles d'interaction

Définition 5 (Priorité). *Une priorité est une relation d'ordre, notée* $\prec$*, entre deux interactions.*

$\forall a, b \in IC,\ a \prec b \Leftrightarrow a$ *est dominé par* b*.*

Cette définition fournit une contrainte qui maintient le système dans les états qui satisfont U^X et depuis lesquels seuls les interactions qui satisfont U^a sont exécutables.

Le système compositionnel étudié S qui nous intéresse est le système à transitions B auquel est appliqué une contrainte U et les interactions IM. La restriction d'un système est définie par une contrainte.

Définition 6 (Restriction). *La restriction d'un système à transitions*

$$B = (X, IC, \{G^a\}_{a \in IC}, \{F^a\}_{a \in IC})$$

avec une contrainte $U^X \wedge \bigwedge_{a \in IC} t^a(U^a)$ *est le système*

$$B/U = (X, IC, \{(G^a)'\}_{a \in IC}, \{F^a\}_{a \in IC})$$

où $(G^a)' = G^a \wedge U^X \wedge U^a \wedge U^X([F^a(x)/x])$ *est la garde restreinte.*

Le terme $U^X([F^a(x)/x])$ de la restriction impose que l'état du système soit valide après le franchissement de l'interaction a. Pour cela, les variables X, dans U, sont remplacées par la fonction associée à la transition, $F(X)$. La pré-condition sur X ($pre(U)(X)$), nécessaire pour que l'application de la fonction $F(X)$ ne sorte pas de la condition U, est ainsi obtenue.

Par exemple, pour garantir $x < 2$ dans le composant `comp`, une contrainte `comp.x<2` est définie sur le système . Si maintenant il existe une interaction qui incrémente x, $x = x+1$. Prometheus doit calculer la contrainte avant l'incrémentation, x est donc remplacé dans la condition $x < 2$ par $x+1$ qui est la fonction de transition. La contrainte devient $x + 1 < 2$ soit $x < 1$. C'est la condition $pre(U)(x)$ pour que la contrainte du système soit respecté après franchissement de l'interaction considérée. Les interactions susceptibles de mettre le système dans un état incorrect, sont ainsi interdites.

La relation de priorité définit précédemment induit la proposition suivante.

Proposition 1 (Priorité). *Une priorité entre deux interactions donne l'implication suivante :*

$$\forall a, b \in IC,\ a \prec b \Rightarrow (G^b \Rightarrow \neg(G^a)')$$

Notre exemple, illustré en 6.8, correspond à des rendez-vous binaire bloquant. En effet, les interactions **start** et **halt** de l'émetteur sont équivalentes à des **put** et les interactions **go** et **halt** du récepteur équivalentes à des **get**. Des contraintes supplémentaires apparaissent alors sur les gardes des transitions.

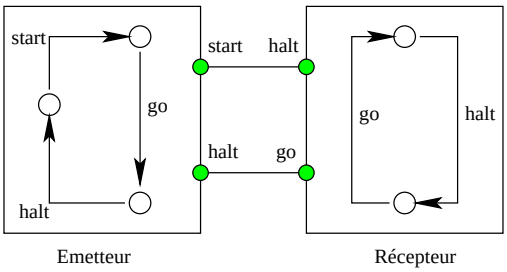


FIG. 6.8 – Le modèle d'interaction de l'exemple

Il reste encore à définir, la composition des systèmes

Définition 7 (||). *La composition de deux systèmes $S[K_1]$ et $S[K_2]$ est le système*

$$\begin{aligned} S[K_1 \cup K_2] &= (B[K_1], IM[K_1])/U_1 \ ||\ (B[K_2], IM[K_2])/U_2 \\ &= (B[K_1] \times B[K_2], IM[K_1] \cup IM[K_2] \cup IM[K_1, K_2])/(U_1 \wedge U_2 \wedge U_{12}) \end{aligned}$$

où $IM[K_1, K_2]$ décrit les interactions entre $K1$ et $K2$ et U_{12} les contraintes.

6.2.2 Résultats de correction

Actuellement, les résultats disponibles portent sur le non-blocage [10], la vivacité et le déterminisme du système composé. Deux aspects du non blocage sont analysés, il s'agit du non-blocage du système composé et du non-blocage des composants dans le système composé.

La preuve du non blocage du système est apporté après la recherche des états bloquants. Le graphe de dépendance permet de trouver ces états s'ils existent.

Pour $k \in K$, Soit dlf_k un invariant non vide sur X_k impliquant l'absence de blocage du composant k, on a $dlf_k \Rightarrow \bigvee_{a \in IC_k} G^a$. On pose $dlf = \bigwedge_{k \in K} dlf_k$.

Le non-blocage du système complet peut être démontré en utilisant la proposition issue de [10].

Proposition 2 (Non Blocage). *$(B, IM)/U$ est sans blocage pour $dlf \wedge U^X$ si $dlf \wedge U^X \Rightarrow \bigvee_{\alpha \in IC^+} (G^\alpha)'$.*

Les résultats pour le non blocage du système ont été donnés mais le fait que le système puisse toujours exécuter une interaction ne guarantit pas que chaque composant le peut. Il faut donc également montrer que tous les composants peuvent effectuer au moins une interaction malgré leurs interactions avec les éléments du système, les composants et les contraintes. La définition 8 donne une condition suffisante pour le non blocage individuel.

Pour cela, une exécution est définie comme une suite infinie d'interactions sur un système B.

Définition 8 (Non blocage individuel). *Pour un système S donné, un composant $k \in K$ est sans blocage dans S si pour toutes les exécutions σ de S et pour tout préfixe σ_n de σ, il existe une exécution σ' telle que $\sigma_n\sigma'$ est une exécution de S, et des interactions de σ' contiennent une interaction de k.*

6.2.3 Prometheus, une implémentation

Nous allons voir comment ce système à transitions a été implémenté.

Pour trouver les différents modèles à utiliser pour modéliser les systèmes SystemC en modèle Prometheus, il faut d'abord bien comprendre comment le modèle Prometheus est construit et comment il est analysé. La modélisation est basée sur l'utilisation de gardes, contraintes et régles de compositions.

À présent, tout cela va être illustré par un petit exemple. Il s'agit d'un exemple très simple différent de celui présenté pour SystemC. Ce dernier sera vu en détail au chapitre 10.

```
1   SYSTEM readwrite
    COMPONENT signal_master
            VAR state;
            TRANSITIONS
5                   ACTION write IF !state DO state:=true;
                    ACTION check IF state DO state:=false;
    END signal_master

    COMPONENT signal_slave
10          VAR signal;
            TRANSITIONS
                    ACTION write IF signal DO signal:=false;
                    ACTION wait  IF !signal DO signal:=true;
    END signal_slave
```

```
15
    INTERACTIONS    signal_master.write|signal_slave.write;
    INCOMPLETE      signal_master.write signal_slave.write;
    ASSERT          signal_master.state=0 or signal_slave.signal=0;
    DISABLE         signal_master.check UNLESS signal_slave.signal=0;
20
    END readwrite
```

Chaque fichier contient un système qui est nommé à la suite du mot clé `SYSTEM` (ligne 1) et qui se termine par le mot clé `END` (ligne 21). Les composants sont ensuite identifiés par le mot clé `COMPONENT` (lignes 2 et 9) et se termine également par le mot clé `END` (lignes 7 et 14). Chaque composant contient une liste de variables. Il s'agit d'une liste après le mot clé `VAR` (lignes 3 et 10). Les variables sont séparées par des espaces. On trouve ensuite la liste des transitions. Chacune est déclarée par le mot clé `ACTION` (lignes 4 à 6 et 11 à 13)suivi de son nom, de la garde précédée de `IF` et de la fonction de transition précédée de `DO`.

Il reste à donner les régles de composition du système. Il faut donc donner dans l'ordre les interactions, les interactions incompletes, les contraintes sur les gardes issues de l'état d'un autre composant et enfin les contraintes d'état entre les composants. Les mots clés sont respectivement `INTERACTIONS`, `INCOMPLETE`, `ASSERT` et `DISABLE <> UNLESS <>` (lignes 16 à 19).

`INTERACTIONS` et `INCOMPLETE` permettent de décrire les interactions du système tandis que `ASSERT` et `DISABLE <> UNLESS <>` permettent de définir ses contraintes.

Les listes d'interactions sont séparées par des espaces et le caractère de composition des interactions est le `|`. Pour exprimer les contraintes complexes, il faut utiliser les mots clés `AND` et `OR`.

Pour accéder à une interaction ou une variable interne, la syntaxe suivante est utilisée :
`<COMPOSANT>.<ACTION ou VARIABLE>`.

Prometheus nous donne le résultat ci-dessous. Il commence par analyser les composants individuellement pour s'assurer qu'ils sont bien sans blocages. Puis, après composition, cette propriété est vérifiée sur le système. L'absence de blocage des composants est revérifié, mais une fois qu'ils sont insérés dans le système. La propriété suivante, qui est vérifiée, est la vivacité des composants. Le déterminisme du système est finalement vérifié ainsi que la confluence locale.

```
1   signal_master deadlock-free in (true)
2   signal_slave deadlock-free in (true)
3   Refining component invariants.. done.
4   signal_master || signal_slave deadlock-free
5   Component signal_master is deadlock-free
6        in signal_master || signal_slave
7   Component signal_slave is deadlock-free
8        in signal_master || signal_slave
```

```
signal_master is live in signal_master || signal_slave
signal_slave is live in signal_master || signal_slave
System is locally confluent and component-deterministic

Terminating...
```

Chapitre 7

L'analyseur syntaxique existant

SystemC étant étant une librairie C++, le code SystemC peut être analysé avec les règles lexicales et syntaxiques du C++ et l'utilisation d'un analyseur existant être envisagée. La sémantique apportée par SystemC est cependant perdue.

Le code SystemC peut également être analysé avec des règles lexicales et syntaxiques très proche du C++. Mais cette approche suppose de (ré)écrire un analyseur syntaxique similaire à celui de C++. C'est donc une approche très coûteuse en temps bien que conceptuellement assez simple puisque il suffit de modifier le lexique et la grammaire de C++ pour y ajouter les éléments que SystemC apporte.

L'approche retenu dans le cadre de la collaboration entre ST Microelectronic et Vérimag est un analyseur plus complexe mais réutilisant les sources de GCC et SystemC avec des modifications mineures.

7.1 Son fonctionnement

Pour utiliser l'analyseur syntaxique, décrit dans le schéma 7.1, le générateur de code appelle la méthode `scp_main` de l'analyseur. Celle-ci appelle la fonction `sc_real_main` qui a été ajoutée au noyau de SystemC et qui remplace la fonction main de SystemC. Juste avant cet appel, le contexte est sauvegardé en vue d'un saut dans le programme. La fonction `sc_real_main` lance alors la phase d'élaboration comme l'aurait fait SystemC en appelant la méthode `sc_main` du modèle utilisateur SystemC à tester. L'élaboration se termine par l'appel au simulateur, ce qui correspond à l'appel de la méthode `sc_start`. Cette méthode a également été déroutée et ne renvoie plus vers le code du noyau SystemC mais vers la fonction `scp_parser_start` de l'analyseur. Cette fonction se contente de restaurer le contexte précédent.

A cet instant, tout le modèle à tester est chargé en mémoire. L'analyseur lance alors la méthode `scp_call_gcc` après avoir mis en place la décoration sur les processus SystemC déjà en mémoire. L'essentiel de la méthode `scp_call_gcc` est l'appel à la méthode `toplev_main` de l'analyseur C++ de GCC.

L'analyseur GCC a lui aussi été modifié et fait des appels à la méthode `scp_gcc_analyse_function_hook` de l'analyseur SystemC. Cette fonction appelle la méthode `scp_simcontext_decorate` qui accède à des données de SystemC. Elle n'est appellé que lors de l'analyse de certain noeud de l'arbre, ceux correspondant à des fonctions membres d'une classe. Il s'agit de faire le lien entre cette branche de l'arbre abstrait et les processus des modules. `scp_simcontext_decorate` cherche également les modules dont hérite le module courant à la recherche de surcharge des fonctions. Actuellement, seul la surcharge des fonctions `read` et `write` nous intéresse. La fonction renvoye alors le type de fonction, le type SystemC ou C++ classique. Si il s'agit d'une fonction SystemC, `analyse_function_body` est appelée. Cette dernière utilise les informations fournies par SystemC et par GCC afin de les unifier et d'améliorer la décoration de l'arbre abstrait. Elle construit un vecteur de paire (noeud de l'arbre, processus). Actuellement, l'essentiel de la décoration concerne les données membres non-static.

La méthode `scp_main` est alors finie et les informations en mémoire peuvent être exploitées dans un générateur de code à notre convenance pour former un compilateur.

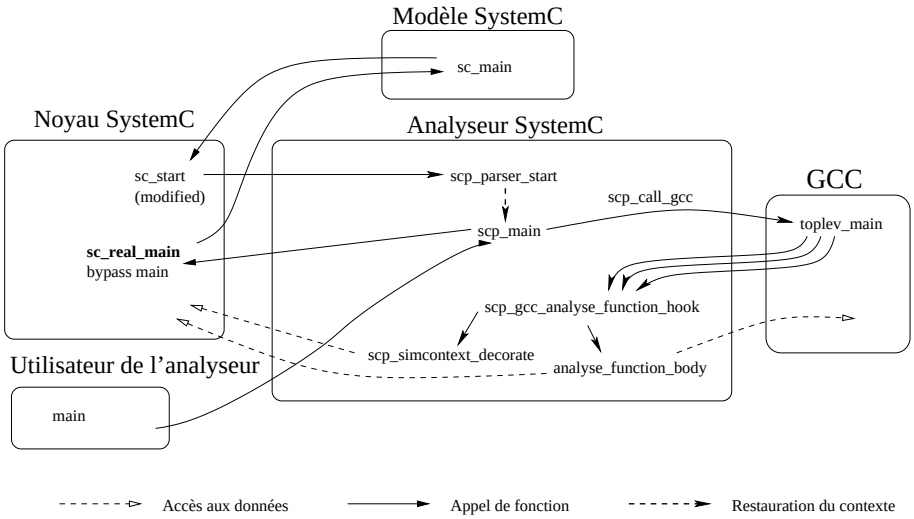


FIG. 7.1 – Architecture de l'analyseur syntaxique SystemC

Les composants du système trouvés sont de types `sc_signal` ou `sc_module`. Un ensemble de listes contenant les objets SystemC peut donc établi.

7.2 La structure d'informations disponible et son utilisation

L'analyseur syntaxique de Matthieu Moy nous fournit la liste des objets SystemC ainsi que la liste des processus. De plus, pour chaque processus, l'arbre abstrait de GCC correspondant complété par une décoration supplémentaire spécifique à Sys-

temC est disponible. tous les éléments cités précédemment et nécessaire pour la suite sont donc bien présents.

La structure de l'arbre est relativement complexe et ne présente pas d'intérêt à être détaillée ici. Elle est disponible en ligne à l'adresse [6] :

http ://gcc.gnu.org/onlinedocs/gccint/Trees.html

Pour comprendre plus facilement la suite, les bases de cet arbre abstrait sont précisées. Il s'agit d'un ensemble de noeuds. Chacun a un type qui permet de savoir quelles sont les informations disponibles. Le nombre d'opérandes se déduit du type de noeud. Tous les accés aux noeuds doivent se faire par des macros pour des raisons de compatibilité. Pour parcourir l'arbre, il existe la macro `TREE_CHAIN` qui renvoie le noeud suivant. Les suites d'instructions sont ainsi parcourues. Le type du noeud est accessible avec la macro `TREE_CODE`, la valeur avec `TREE_VALUE` et les opérandes avec `TREE_OPERAND`.

Certaines instructions sont de type "simple", c'est à dire qu'elles n'ont pour paramètre que des données alors que d'autres comme les instructions de contrôle ont des paramètres qui sont eux-même des suites d'instructions (structure récursive). Ces instructions seront de type "complexe".

Le générateur de code utilise l'arbre abstrait [6] coloré fourni par l'analyseur. Il commence par sélectionner chaque composant pour générer le composant équivalent en Prometheus. Pour chacun, il sélectionne ses processus sous la forme de l'arbre fourni. La racine de cet arbre est le premier élément de la séquence d'instructions. A chaque instruction complexe, le générateur trouve un noeud de l'arbre ayant une ramification de type `COMPOUND_STMT`. Les autres noeuds ont des fils décrivant des données, des expressions arithmétiques ou des appels de fonctions.

Une méthode récursive permet de parcourir les chaînes de l'arbre sur les noeuds non terminaux. Pour chaque noeud terminal de l'arbre, c'est à dire équivalent à une instruction simple, le générateur est en présence d'un cas de base et génère le programme Prometheus équivalent.

Chapitre 8

Notre générateur de code

Nous allons voir la méthode de traduction qui a été mise en place dans le cadre de ce stage et qui permet le passage d'un système modélisé en SystemC vers un système compositionnel en couches à base de systèmes à transitions avec contraintes. Une fois les différentes constructions de SystemC présentées et mises en évidence, nous verrons les structures correspondantes dans le modèle compositionnel.

8.1 Schéma général

Une architecture SystemC donnée, notée A, comporte des signaux Σ_A et des modules M_A. Chaque module m de M_A est composé de ports P_m et de processus Π_m. Chaque processus est un ensemble I d'instructions.

Chaque fois que le contexte le permettra, les éléments M_A, Σ_A et Π_m seront notés respectivement M, Σ et Π.

S est le modèle compositionnel issue du système à transitions B restreint par un modèle d'interaction IM et une contrainte U.

Le modèle étant formé de trois couches, nous allons voir la traduction d'une instruction et de son contexte pour chacune d'elle. La première couche étudiée sera celle du comportement des composants. Il s'agit de définir les variables et les interactions de base ainsi que les gardes et les fonctions de transition. La couche d'interaction entre les composants sera ensuite construite et enfin la couche de contraintes. Des fonctions Φ_B, Φ_{IM} et Φ_U définissent respectivement ces couches du modèle.

Le langage C++ est un langage impératif, les instructions I sont liées et forment un graphe connexe, c'est à dire qu'une instruction peut elle-même être composée de plusieurs blocs d'instructions. Chacune sera associée à une ou des transitions, c'est à dire des tuples interaction, garde, fonction de transition.

Un processus est dans un état donné entre deux transitions, cet état est représenté par la variable s_π. Les variables d'un module et de ses processus, ainsi que leur évolution doivent être prises en compte puisqu'elles caractérisent l'état du processus. Afin de diminuer l'espace d'état, il est possible d'ignorer les variables constantes ou ajouter d'autres optimisations [2] mais cela n'étant pas l'objet de ce stage, ces problèmes ne seront pas abordés ici.

Les gardes contrôlent l'évolution du système et les fonctions de transition le font évoluer. Le principe de la traduction est d'associer à chaque instruction un schéma de code de ces éléments et de le compléter en construisant, pour chaque processus, le graphe connexe et sans blocage dans lequel le complément des gardes et des fonctions de transitions modélisent la structure du modèle SystemC, c'est à dire les sections linéaires et les divergences et convergences liées aux structures de contrôle de C++.

Le principe de construction, illustré par la figure 8.1, est de parcourir l'ensemble des instructions en profondeur. Les instructions de contrôle sont des instructions ayant une condition et un ou plusieurs blocs d'instructions en paramètres. C'est ces blocs qui sont parcourus en profondeur. Dans l'exemple, le `while` est la première instruction du "processus". Les interactions associées partent donc de l'état 0. Dans le cas d'un `while`, il y a un seul bloc paramètre qui est exécuté si la condition est vraie. La transition de l'état 0 vers l'état 1 représente les cas où la condition est vraie. L'état 1 est donc l'état d'entrée du bloc. Les instructions consécutives b et c sont représentées par une chaînes dans le graphe, état 1 à 3. L'instruction `if` a deux blocs, le premier lorsque la condition est vraie, transition de 3 vers 4, le second lorsqu'elle est fausse, transition de 3 vers 5. Le premier bloc est d'abord parcouru, son état de sortie n'est pas encore déterminé et il est donc nécessaire de mémoriser la dernière interaction du bloc. Le second bloc du `if` est ensuite construit. Son état de sortie, état 7, est répercuté vers la dernière interaction du premier bloc, transition de 4 vers 7. Lorsque le bloc du `while` est entièrement traité, une interaction de saut permet de passer de l'état 7 de sortie du bloc vers l'état 0 d'entrée du `while`. La condition est alors retestée. Lorsqu'elle est fausse, l'instruction `while` est terminé et l'automate passe vers l'état 8 qui marque la fin du `while`.

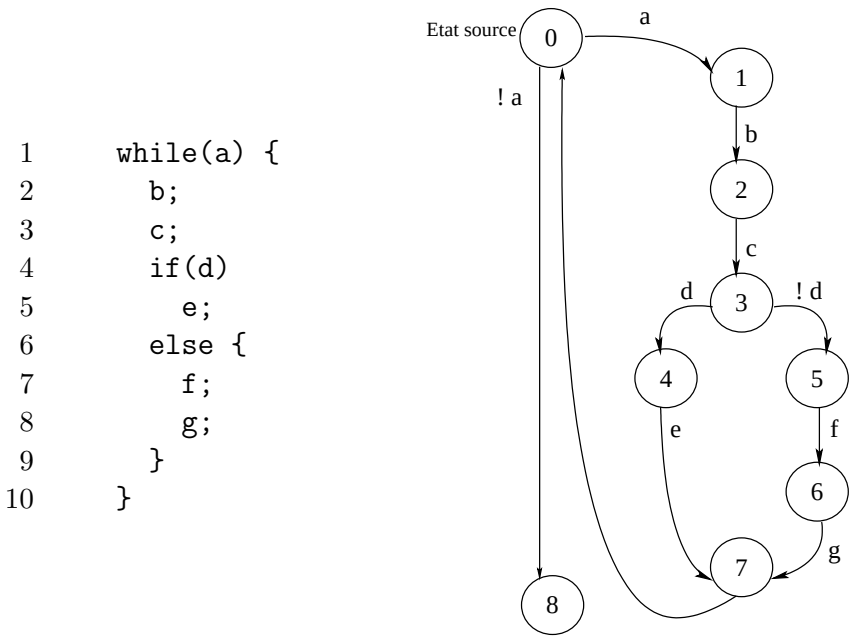


FIG. 8.1 – Principe de construction par parcours en profondeur

Les instructions SystemC peuvent avoir des influences trés variées sur le système, toutefois afin de les traduire aisément dans le modèle compositionnel, elles vont être

classées en différentes catégories selon le type d'interactions qu'elles représentent et leur rôle dans le modèle d'interaction et les contraintes :

- structures de contrôle tels que `while, if, do`
- appels de méthode SystemC ou TLM tels que `write, wait`
- appels de méthode ou de fonction C++ tels que `printf`
- affectations et toutes les expressions arithmétiques tels que `y++, y = x + 1`.
- opérations sur les flots d'entrées/sorties C++ tels que `cout << var << endl`

Selon la classe à laquelle l'instruction est rattachée, il faut :

- créer une interaction dans la couche comportementale qui éventuellement :
  - modifiera, dans sa fonction de transition, une variable du modèle compositionnel qui modélise une variable de SystemC. C'est le cas si l'instruction SystemC affecte une variable.
  - aura une garde plus forte que celle du graphe d'état qui modélise la structure du programme SystemC.
- ignorer l'instruction (pour les cas d'affichage par exemple) lorsqu'elle n'intervient pas dans le comportement du processus vis à vis des autres processus.
- créer une interaction dans la couche du modèle d'interaction.
- ajouter une contrainte au système dans la couche de contrainte

Cas de la librairie TLM

Dans le cas de TLM, l'appel à la méthode `write` dans le composant maître correspond alors à l'appel du processus `write` de l'esclave. Cet appel doit être vu comme un appel de méthode. Il y a donc deux synchronisations entre le maître et l'esclave. La première a lieu à l'appel de la méthode et permet à l'esclave d'exécuter sa tâche pendant que le maître attend. La deuxième a lieu lorsque l'esclave a terminé et que le maître doit reprendre son exécution. TLM est un modèle à base de transaction qui peut être représenté par ces deux rendez-vous comme sur le schéma 8.2.

8.2 La couche comportementale

La couche comportementale est un tuple $(X, IC, \{G^a\}_{a\in IC}, \{F^a\}_{a\in IC})$. La fonction Φ_B est définie sur les modules et les signaux de SystemC et donne pour chacun des deux les composantes du tuple.

8.2.1 Les signaux booléens

Le composant $\Phi_B(\sigma)$ est défini statiquement par le modèle utilisé pour un signal σ donné. Dans le cadre de ce stage, seul les signaux booléens ont étés modélisés :

Définition 9 ($\Phi_B(\sigma_{bool})$). *Nous définissons la traduction d'un signal booléen σ_{bool} par :*

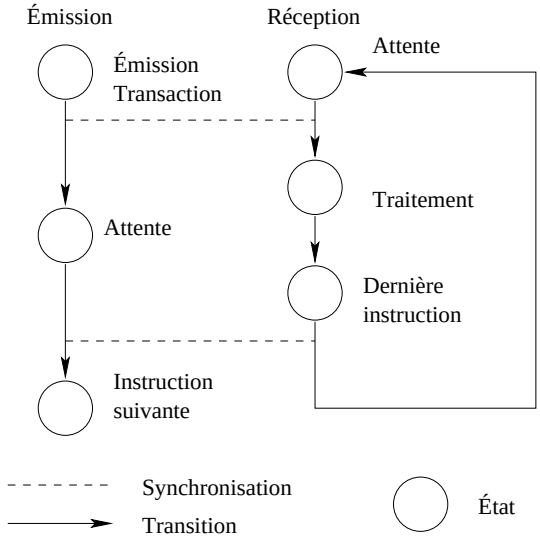


FIG. 8.2 – Graphe d'une transaction

$$B_{\sigma_{bool}} := \begin{cases} X = \{C, N, \delta\} \\ IC = \{WR_F, WR_T, RD_F, RD_T, notify, \delta-, \delta\} \\ G = \{G^a | a \in IC\} \\ F = \{F^a | a \in IC\} \end{cases}$$

où C :=valeur booléenne courante du signal, et N :=valeur future, la variable δ indique si le signal est dans une phase de mise à jour des variables par l'ordonnanceur. avec

$a \in IC$	G^a	F^a
WR_T	$true$	$N := true$
WR_F	$true$	$N := false$
RD_T	C	id
RD_F	$\neg C$	id
$\delta-$	$(C \wedge \neg N) \vee (\neg C \wedge N)$	$\delta := true$
$notify$	δ	id
δ	δ	$C := N, \delta := false$

Il s'agit là du modèle comportemental de l'automate 6.4 à la page 24 qui est exprimé dans le système à transitions. L'action $\delta-$ marque la fin d'un cycle entrée/sortie est le début des notifications et de la mise à jour. L'action `notify` permet de notifier l'ensemble des composants qui sont en écoute sur le signal σ. C'est pour cela que sa fonction de transition est idempotente. L'action δ permet l'évolution du temps et met à jour les variables internes du signal.

Il reste à définir Φ_B sur les modules.

8.2.2 L'ordonnanceur

De même que pour les signaux, la modélisation de l'ordonnanceur doit être réalisé statiquement. La figure 8.3 illustre le modéle utilisé. Il ne comporte que deux états

suivant que ce soit les processus ou l'ordonnanceur qui soit actif. Les processus sont actifs dans qu'un nouveau δ-pas est possible. Lorsque l'évolution maximale des processus est atteinte, l'ordonnanceur prend la main pour effectuer une mise à jour du système. Si la fin d'un cycle implique une modification, l'interaction δ^- est réalisée sinon le δ-cycle se termine implicitement et l'interaction t est réalisée puis les processus reprennent le contrôle. Si l'interaction δ^- a été réalisée, toutes les interactions δ pendantes dans les événements et les signaux sont réalisées puis le contrôle est rendu aux processus par l'interaction δ^+.

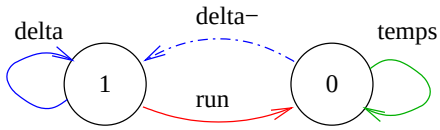


FIG. 8.3 – Modélisation de l'ordonnanceur

8.2.3 Les composants utilisateurs

Les variables X qui caractérisent l'état d'un composant, au niveau du modèle généré, sont issues de la traduction de variables qui sont, soient déclarées dans un module, variables X_m, soient dans un de ses processus, variables X_π. Dans le modèle compositionnel, les variables X sont déclarées pour l'ensemble d'un composant. Il est donc nécessaire de construire une fonction injective entre $X_m \cup X_\pi$ et X. Il faut également gérer les collisions de nom de variables, c'est à dire lorsque $X_m \cap X_\pi \neq \varnothing$, pour que le comportement final soit le même que C++. Nous représenterons X_π dans X de tel sorte que cette intersection soit toujours vide en préfixant les variables de π par "π".

Définition 10 ($\Phi_B(m)$). *Un module inclut des variables et des processus qui contribuent à l'aspect comportemental du système. Pour un module $m \in M$, l'ensemble Π_m des processus de m est défini. Il se traduit en $\phi_B(\Pi_m)$. Chaque traduction des processus d'un module est ensuite composée pour former le comportement du composant.*

$$\phi(\Pi_m) := \bigcup_{\pi \in \Pi_m} \phi(\pi)$$

$$\Phi(m) := \begin{cases} X := \{X_m\} \cup \phi_X(\Pi_m) \\ IC := \phi_{IC}(\Pi_m)) \\ G := G^a \mid a \in IC \\ F := F^a \mid a \in IC \end{cases}$$

où Π_m est l'ensemble des processus du module m, $X_\pi := \phi_X(\Pi_m)$ est la traduction des variables d'état des processus de m et $\phi_{IC}(\Pi_m)$, la traduction des instructions en interactions.

Les processus étant des graphes d'instructions, il est nécessaire de préserver cette structure. La garde et la fonction de transition permettent cela. En effet, de la

même façon qu'une instruction ne s'exécute que lorsque l'instruction précédente est terminée, la garde d'une interaction a ne doit être vraie que si une interaction précédente a placé le système dans un état initial du schémas de code de l'interaction a.

8.2.4 Les instructions des processus

La contribution $\varphi(i)$ des instructions qui forment un processus n'est pas immédiate. La structure du processus, c'est à dire l'ordre des instructions, doit être modélisée.

Pour chaque instruction, des schémas de code donnent la traduction dans le modèle compositionnel. Ils vont définir des gardes $G^-_{\varphi_{IC}(i)}$ qui seront renforcées et des fonctions de transition $F^-_{\varphi_{IC}(i)}$ qui seront raffinées pour l'ensemble des interactions associées. Ces deux opérations supplémentaires permettront de modéliser la structure de processus en formant un graphe d'état.

L'opérateur $\uplus$ permet d'obtenir la contribution de deux instructions consécutives en réalisant le renforcement de la garde en $G^{\varphi_{IC}(i)}$ et le raffinement de la fonction de transition en $F^{\varphi_{IC}(i)}$. Il utilise une variable s_π, propre au processus π, qui modélise l'état courant du processus.

Définition 11 (L'opérateur $\uplus$). *$\forall i \in \pi$, l'opération $\uplus$ est définie :*

La variable s_π caractérise l'état du processus. $s_{\varphi(i)}$ est la valeur de l'état prédécesseur de la transition associée à l'instruction i et $s_{\varphi(i+1)}$ est la valeur de l'état successeur.

$$\varphi(i_1) \uplus \varphi(i_2) := \left\{ \begin{array}{rcl} X & := & s_\pi \\ IC & := & \varphi_{IC}(i_1) \cup \varphi_{IC}(i_2) \\ G & := & \{G^-_{\varphi(i_1)} \wedge (s_\pi = s_{\varphi(i_1)})\} \\ & & \cup \{G^-_{\varphi_{IC}(i_2)} \wedge (s_\pi = s_{\varphi(i_2)})\} \\ F & := & \{F^-_{\varphi(i_1)} \cup \{s_\pi := s_{\varphi(i_2)}\}\} \cup \{F^-_{\varphi_{IC}(i_2)}\} \end{array} \right.$$

Remarque 1 (Cas des instructions de contrôle). *Par définition, une instruction de contrôle conduit le programme dans des états différents en fonction de la valuation de sa condition ce qui correspond à l'exécution de bloc d'instructions différents dans un processus.*

Ce qui conduit aux deux inégalités suivantes, $|\varphi_{IC}(i)| > 1$ et $|s_{\varphi(i)}| > 1$. De plus, il existe une bijection entre ces deux ensembles qui permet de construire les fonctions de transition complètes dans tous les cas.

Ceci implique que les instructions de contrôle correspondent à plusieurs interactions qui ont des gardes disjointes et non-bloquantes. La relation suivante est donc observée sur ce type de gardes.

$$\bigvee_{g \in G^{\varphi_{IC}(i)}} g = true$$

Ce problème des instructions de contrôle est illustré par le schéma 8.4. En parcourants les blocs d'instructions en profondeur, le numéro d'état de sortie du bloc "a" est déterminé, le numéro de l'état d'entrée du bloc "b"est le suivant. Par un parcours en profondeur de ce nouveau bloc, son état de sortie est déterminé. L'état de sortie de l'instruction `if` est ainsi obtenu.

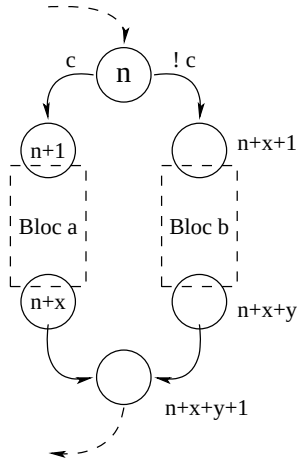


FIG. 8.4 – Détermination de la valeur des états pour un `if`

L'ensemble des instructions d'un processus peut alors être composé en prenant $s_{\varphi(i_0)} = 0$ pour la première instruction du processus notée i_0.

Définition 12 ($\phi_B(\pi)$). *Comme un processus est un ensemble consécutif et récursif d'instructions, $\phi_B(\pi)$ est défini par :*

$$\phi(\pi) := \biguplus_{i \in \pi} \varphi(i)$$

Reprenons la notion de classe vu à la fin de la section 8.1. Il est nécessaire de traiter chacun des cas séparément, les instructions étant de types différents. La classification des instructions est mise en place en vue de construire la fonction φ pour les instructions.

Définition 13 (Classification des instructions). *Soit I, l'ensemble des instructions d'un processus π. I peut être partitionné en trois ensembles, I_D, I_G et I_F.*

Si $i \in I$ est une déclaration de variables, alors $i \in I_D$.

Si $i \in I$ est une instruction de contrôle, alors $i \in I_G$.

Sinon i est une fonction et $i \in I_F$.

i peut alors différencié dans I_F en partitionnant I_F en quatre parties.

- *$I_{F^{id}}$ si la fonction est idempotente.*
- *$I_{F^{ac}}$ si la fonction modifie uniquement l'état interne du composant et qu'elle n'est donc pas synchronisée vis à vis des autres composants. Elle est traduite en une interaction complète. La garde de ce*
- *$I_{F^{si}}$ si la fonction accède à un autre composant et doit se synchroniser avec lui. La traduction est une interaction incomplète du comportement appartenant à une interaction du modèle d'interaction.*

– $I_{F^{sc}}$ *si la fonction accède à un autre composant et peut se synchroniser avec lui. La traduction est une interaction complète car la synchronisation n'est pas obligatoire. C'est le cas d'une écriture ou d'une lecture non bloquante par exemple. Cette interaction fait partie d'une interaction du modèle d'interaction.*

Remarquez l'absence de l'existence d'interaction non synchonisée et incompléte. En effet, une telle interaction ne serait jamais réalisable.

Définition 14 ($\varphi_B(i)$). *$\varphi_B(i)$ est défini pour $i \in I \setminus I_{F^{id}}$ par :*

$$B := \begin{cases} X := \{X(i)\},\ i \in I_D \\ IC := label(i),\ i \in \{I_{F^{ac}} \cup I_{F^{si}} \cup I_{F^{sc}} \cup I_G\} \\ G_a^-,\ a \in IC \\ F_a^-,\ a \in IC \end{cases}$$

label(i) est un nom unique pour désigner chaque interaction dans le composant. Une instruction a plusieurs labels lorsqu'elle se traduit en plusieurs interactions, gardes et fonctions de transition. Ceci est particulièrement utile lorsque une méthode est appelée plusieurs fois dans un même processus, ce qui est souvent le cas. Une méthode possible est indexé les appels de même type. Il est également possible de posfixer les interactions par la valeur de l'état courant.

Dans le cas général, les gardes et fonctions de transition sont les suivantes :
– $G^-_{\varphi_{IC}(i)} = true$ *si* $i \in \{I_{F^{ac}} \cup I_{F^{si}} \cup I_{F^{sc}}\}$
– $F^-_{\varphi_{IC}(i)} = id$ *si* $i \in I_G$

L'ensemble $I_{F^{id}}$ se traduit par des interactions ayant une fonction de transition idempotente dans le schéma de code des instructions. Ces interactions ne sont pas synchonisées avec d'autres et sont complètes. Leur garde non renforcée est toujours vraie, leur fonction de transition est l'identité et elles ne sont pas restreinte par le modèle d'interaction, étant compléte. Elles ne présentent donc aucun intérêt à être modélisées. Pour simplifier le modèle, ces interactions ne sont pas générées.

8.2.5 Gestion des variables

Prometheus utilise des contraintes booléennes. Le codage des états doit être réalisé par des variables booléennes. Il faut mettre en place un système de codage en binaire pour l'ensemble des variables non booléennes dont l'état du processus, s_π. Comme l'usage d'une variable par état produit une solution coûteuse, il est nécessaire de coder les entiers naturels en binaire.

Une partie de notre compilateur aura pour tâche cette transformation. On essayera de la rendre aussi indépendante que possible afin de pourvoir la placer dans le code de Prometheus et autoriser ainsi les entiers naturels directement dans le programme Prometheus qui sera alors plus lisible.

Une représentation exacte de l'état du processus est possible. Cependant gérer un entier sur 16 ou 32 bits peut facilement faire exploser la taille de la représentation, il

serait donc judicieux d'effectuer une analyse du processus afin de trouver le domaine réellement utilisé par la variable. Le nombre de bits utilisés peut ainsi être limiter à $n = \lceil log_2\ d \rceil$ où d est le plus grand entier identifiant un état, c'est à dire le maximum utilisé par la variable s_π.

Pour une question de temps, nous commencerons la programmation du compilateur par une abstraction grossière des variables entières i en variables booléennes b. Les valeurs nulles correspondront à une valeur fausse tandis que les valeurs non nulles correspondront à des valeurs vraies.

Définition 15. *À une variables entières i est associée $b = \phi(i)$, une variable booléenne telle que : $\phi(0) = 0$ et $\forall i \in N^*,\ \phi(i) = 1$*
Par la suite la variable booléenne portera le même nom que la variable entière à laquelle elle est associée.

8.3 Le modèle d'interaction

8.3.1 Définition des relations

Pour déterminer les interactions du système S entre les interactions des composants de B, il est nécessaire de définir des relations dans SystemC. Elles portent sur les modules M et les signaux Σ.

Définition 16 (Relation `bind`). *Pour tous ports p d'un module et tous les signaux σ de l'architecture SystemC, la relation **`bind`** est définie par :*

*le port p est attaché au signal $\sigma \Leftrightarrow$ **`bind`**(p, σ)*

Définition 17 (Relation `use`). *Pour toutes instructions i d'un processus et tous ports p d'un module, la relation **`use`** est définie par :*

*l'instruction i utilise le port $p \Leftrightarrow$ **`use`**(i, p)*
Ceci est valable que le port soit utilisé en entrée ou en sortie.

Les interactions du système peuvent alors être définies entre les interaction des composants.

Définition 18 (Relation `IC`). *Pour un module m et un signal σ donnés, la relation **`IC`** est définie par :*

$$\forall p \in P_m \forall \pi \in \Pi_m \forall i \in \pi.\ \textbf{\texttt{use}}(i, p) \wedge \textbf{\texttt{bind}}(p, \sigma) \Rightarrow \exists ic_\sigma \in \phi(\sigma), \textbf{\texttt{IC}}(\varphi_{IC}(i), ic_\sigma)$$

C'est à dire que toute instruction utilisant un port qui est attaché à un signal a une traduction qui est en interaction avec une interaction de la traduction du signal.
Cette relation participe à la construction du modèle car

$$\texttt{IC}(\varphi_{IC}(i), ic_\sigma) \Rightarrow \varphi_{IC}(i) \shortmid ic_\sigma \in IC$$

Les processus peuvent également communiquer par des événements. Il est donc nécessaire de les définir et de mettre en évidence les relations qu'ils impliquent et qui génèrent des interactions.

Définition 19 (Relation `notify`). *Une instruction peut être une notification d'événement. Une relation entre une instruction i et un événement e est donc définie :*

i appelle la méthode `notify` *de l'objet e* $\Leftrightarrow$ $\texttt{notify}(i, e)$

Définition 20 (Relation `wait`). *Une instruction peut être en attente d'un événement, une seconde relation entre une instruction i et un événement e est donc définie :*

i appelle la méthode `wait` *sur l'objet e* $\Leftrightarrow$ $\texttt{wait}(i, e)$

Remarque 2 (Attente sur un événement). *En SystemC, la liste de sensibilité, événements et signaux, d'une attente peut être déclarée de plusieurs façons. Elle peut être statique ou dynamique. Si l'attente est déclarée dynamiquement, elle peut l'être de différentes façons également. Il convient de bien suivre les spécifications de SystemC.*

Une attente statique utilise la primitive `sensitive` *lors de l'instantiation, c'est à dire que la liste de sensibilité est définie avant la simulation. L'instruction* `wait()` *met en attente sur cette liste.*

Une attente dynamique est une attente sur un objet ou un ensemble d'objets de type événement. L'instruction utilisée est alors du type `wait(e)`, `wait(e1 & e2)`, `wait(e1 | e2)` *ou* `wait(s)`*. Si une liste statique est définie, la liste dynamique la remplace, elle n'est donc plus prise en compte.*

Des alarmes peuvent être également mises en place. Si aucun événement n'arrive avant l'expiration de l'alarme, l'attente est malgré tout terminée. Ce dernier mode d'attente n'est pas pris en compte dans l'analyseur actuellement utilisé.

Il faut à présent mettre en relation les deux instructions qui font références à un même événement.

Définition 21 (Relation `IC`). *La relation* `IC` *entre deux instructions, utilisant un même événement* `e`*, peut alors être définie :*

$$\forall \pi \in \Pi_m \forall \pi' \in \Pi_{m'} \forall i \in \pi \forall i' \in \pi'.\ \texttt{notify}(i, e) \wedge \texttt{wait}(i', e) \Rightarrow \texttt{IC}(\varphi_{IC}(i), \varphi_{IC}(i'))$$

C'est à dire que deux instructions ont leur traduction composée dans le modèle d'interaction si l'une notifie un événement et que l'autre l'attent.

8.3.2 Traduction des relations en interactions

Les connexions entre les différents modules via des canaux impliquent des interactions qui définissent le modèle d'interactions IM.

– Les instructions de contrôle sont considérées sur l'état interne du système et elles n'ont pas par conséquent d'interaction avec les autres composants. Il s'agit donc d'actions complètes. $\forall i \in I_G,\ \varphi_{IC}(i) \in IC^+$

– Les éléments $I_{F^{ac}}$ sont associés à des actions qui ne participent pas au modèle d'interaction et qui sont complètes. $\forall i \in I_{F^{ac}},\ \varphi_{IC}(i) \in IC^{+}$

– Les éléments de $I_{F^{si}}$ sont des instructions dont les traductions sont des actions incomplètes participant à au moins une interaction. $\forall i \in I_{F^{si}},\ \varphi_{IC}(i) \in IC^{-}$. Ces actions doivent être composées.
$i_1, \ldots, i_n \in I_{F^{si}}, \varphi_{IC}(i_1) \shortmid \cdots \shortmid \varphi_{IC}(i_n) \in IC^{+}$ ssi $\texttt{IC}(i_1, \ldots, i_n)$.

– Les éléments de $I_{F^{sc}}$ sont des instructions dont les traductions sont des actions complètes participant à au moins une interaction.
$i_1, \ldots, i_n \in I_{F^{sc}}, \varphi_{IC}(i_1) \shortmid \cdots \shortmid \varphi_{IC}(i_n) \in IC^{+}$ ssi $\texttt{IC}(i_1, \ldots, i_n)$.

L'architecture SystemC peut avoir plusieurs signaux, avec plusieurs processus en écoute sur un même signal. De plus un processus peut écouter un même signal ou événement à des instants différents. Des interactions n-aires devraient alors apparaître or les interactions du système peuvent restées binaires. C'est la couche de contraintes qui va permettre, par des priorité et des contraintes d'état, de notifier l'ensemble des processus en écoute compléter par deux interactions n-aires mais avec l'ordonnanceur.

L'évolution du temps implique la synchronisation de l'ensemble des instructions liées au temps `delta` et `temps`. Les notifications `notify` seront synchronisées avec les interactions `wait` respectivement associées via un événement. L'interaction `notify` est synchronisée avec les écouteurs potentiels. Pour une notification immédiate, ceux en écoute sont immédiatement rendus éligibles à la prochaine évaluation, mais la non-préemption permet au processus courant de finir son propre δ-pas en cours. Sinon la notification est différée en fin de δ-cycle.

L'ensemble des interactions `temps` des événements et des attentes temporisées est en interaction avec l'interaction `temps` de l'ordonnanceur, illustré page 41. Il en est de même pour les interactions `delta-`. Cette interaction est issue de la décomposition du changement de δ-cycle en trois phases. `delta-` marque la fin des δ-pas possibles et la possibilité de faire un nouveau δ-cycle. La seconde phase notifie toutes les éléments en attente grâce à l'interaction `delta`. Enfin, l'interaction `run` signale que tous les éléments possibles ont été notifiés et qu'il faut reprendre l'exécution des processus.

La méthode de construction des interactions est appliquée à l'exemple de la figure 8.5, page 48. Cet exemple ne contient pas d'éléments tels que de l'affichage, des calculs ou des instructions de contrôle afin de pouvoir se concentrer sur les aspects modèle d'interaction et contraintes. La figure représente uniquement les états initiaux et les états bloquants, marqués ici par des `wait`.

L'indexation pour les interactions générées par des instructions similaires, est réalisée en postfixant la valeur de l'état. L'appel `e.notify(0)` du processus `P0` génère l'interaction complète `e.mem_d|P0.notify0`. L'appel `wait(e)` du processus `P2` génère l'interaction compléte `e.notify|P2.wait0`. La même opération est appliqué au événement `e0, e1` et `e2`.

```
e0.mem_t|P2.notify1
e1.imm|P1.notify1
```

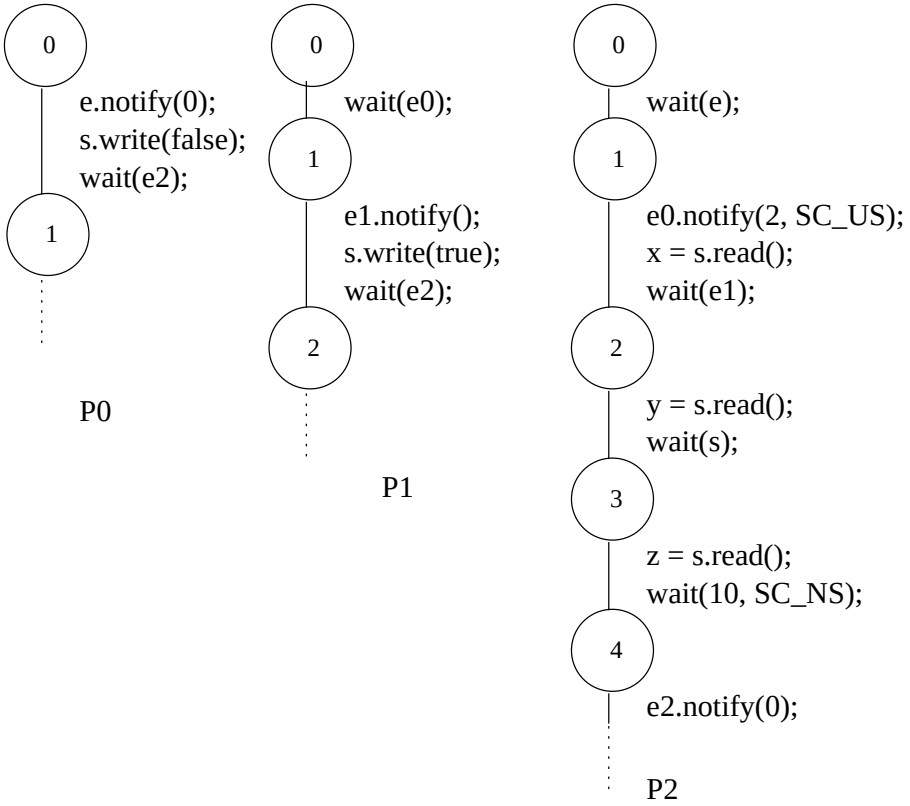


FIG. 8.5 – Exemple pour la construction des interactions et des contraintes

```
e2.mem_d|P2.notify8
e0.notify|P2.wait0
e1.notify|P2.wait3
e2.notify|P1.wait3
e2.notify|P0.wait2
```

L'ensemble IC_e^+ correspond à l'ensemble des interactions précédentes relatives aux événements `e, e0, e1, e2`.

Des interactions complétes sont générées pour les processus utilisant le signal `s`. Elles forment l'ensemble IC_s^+ :

```
P0.write1T|s.writeT P0.write1F|s.writeF
P1.write2T|s.writeT P1.write2F|s.writeF
P2.read2T|s.readT P2.read2F|s.readF
P2.read4T|s.readT P2.read4F|s.readF
P2.read6T|s.readT P2.read6F|s.readF
P2.wait5|s.notify
```

Il reste à définir les interactions de IC_{sched}^+ ayant une action de l'ordonnanceur. L'interaction compléte de la temporisation du processus `P2` forme l'interaction `P2.wait7|sched.temps`. De même, il faut lié les événements et le signal à l'ordonnanceur.

```
s.delta-|sched.delta-
e.delta-|e0.delta-|e1.delta-|e2.delta-|sched.delta-
```

```
e.delta|sched.delta
e0.delta|sched.delta
e1.delta|sched.delta
e2.delta|sched.delta
e.temps|e0.temps|e1.temps|e2.temps|sched.temps
```

L'ensemble IC^+ qui nous intéresse est $IC_e^+ \cup IC_s^+ \cup IC_{sched}^+$.

Cependant toutes les sous-interactions générées ne sont pas complétes, il faut donc préciser les interactions incomplétes. Il faut donc traiter toutes les interactions binaires et les deux interaction n-aires de l'ordonnanceur. Il est nécessaire que l'ordonnanceur participe aux interactions, les interactions incomplètes sont donc toutes celles où l'ordonnanceur est absent.

```
e.delta-|e0.delta-|e1.delta-|e2.delta-
e.temps|e0.temps|e1.temps|e2.temps
```

Dans notre exemple, pour les interactions binaires, les deux interactions qui les composent sont incomplètes à l'exception des interactions :
`P0.notify0, P1.notify1, P2.notify1, P2.notify8`

8.4 La couche de contraintes

Il est vital de prendre en compte le modèle d'exécution de SystemC car il ne doit pas être modifié. En SystemC, il y a non préemption des processus. Notre modèle doit donc avoir un comportement identique.

La non-préemption des processus fait partie du modèle d'exécution de SystemC. Il faut donc rendre atomique les δ-pas. Des contraintes U sur le modèle du système font servir à cela. La représentation la plus aisée de l'atomicité est d'avoir un drapeau pour chaque processus dans chacun des composants. Les signaux n'ont pas de drapeau. Chaque section atomique s'effectue avec le drapeau actif et le libère à la fin. Il suffit d'imposer alors sur le système complet la contrainte d'état qui limite à un drapeau actif maximum l'ensemble des drapeaux du système. C'est à dire qu'il peut y avoir zéro ou un drapeau actif dans tout le système à chaque instant. Il y a ainsi une exclusion mutuelle entre les sections atomiques. Il faut également prendre soin que la phase de mise à jour des signaux et celle de l'ordonnanceur sont bien synchronisées. La contrainte d'états est modifiée en conséquence.

Pour garantir un bon fonctionnement du signal, il faut imposer que les signaux ne soit en phase de mise à jour que si l'ordonnanceur l'est également.

Aucune interaction synchronisée d'attente n'appartient à une section atomique. Ces sections sont un ensemble d'instructions internes aux composants, donc complètes, ou d'opérations d'entrée/sortie qui sont incomplétes mais toujours réalisable, par exemple la lecture d'un signal. Il y aurait sinon un blocage du système. Dans les

deux cas, les instructions appartiennent à un même δ-pas.

Le terme d'interactions `INTERNES` désigne les traductions des instructions C++ standards et des instructions SystemC, à l'exception des interactions bloquantes telle que `wait` et des interactions associés telle que la notification des événements et des signaux, notée `{s, e}.notify`.

L'ordonnancement de SystemC est modélisé par des priorités dans le modèle compositionnel. Le système doit évoluer de manière maximal, les instructions internes sont donc prioritaires sur les instructions de changement de δ-cycle et d'évolution du temps. Une fois l'évolution maximale atteinte, l'ordonnanceur prend le contrôle du système. Il exécute alors de manière atomique les notifications. Pour s'assurer de l'atomicité, le drapeau des processus est également employé par l'ordonnanceur, ce qui bloque les interactions internes.

La phase de notification des δ-cycles peut alors commencer. Cela permet que toutes les notifications `notify`, d'un signal ou d'un événement, possibles soient réalisées avant qu'un nouveau processus ne commence un nouveau pas. Lorsque toutes les notifications ont été effectuées, une interaction `delta`, dans l'ordonnanceur, synchronise et met à jour les signaux et les événements. L'ordonnanceur rend le contrôle aux processus pour faire évoluer certains des processus qui étaient en attente grâce à l'interaction `run` qui libère le drapeau.

Si cette phase n'a pas rendu une interaction interne possible, l'ordonnanceur reprend le contrôle du système et commence la phase de notification des événements temporisés. Cette phase est similaire à la phase de notification des δ-cycles et est elle-aussi atomique. A l'issu des notifications, le temps simulé évolue. L'ordonnanceur libère le système et un processus notifié démarre un nouveau pas. Nous nous limiterons pour l'instant à la représentation des temporisations limitée à un temps minimal non nul. Pour généraliser, il faudra représenter dans l'ordonnanceur le temps simulé de SystemC par un ensemble de variables booléennes pour modéliser une échéancier.

Définition 22 (Priorités de début de changement de δ-cycle). *Afin que l´évolution soit maximale, la priorité suivante est définie :*

$$delta\text{-} \prec INTERNES$$

Définition 23 (Priorités durant le changement de δ-cycle). *Pour chaque* `delta` *d'un événement ou d'un signal, et le* `notify` *associé, la relation suivante est définie :*

$$delta \prec notify$$

Définition 24 (Priorités de fin de changement de δ-cycle). *La reprise d'un nouveau δ-cycle par l'interaction* `run` *de l'ordonnanceur, est dominée par l'interaction* `delta` *afin que tous les événements et les signaux soient mis à jour.*

$$run \prec delta$$

Définition 25 (Priorité pour la gestion du temps simulé). *L'évolution des δ-cycles représente un temps nul, la relation de priorité suivante est donc définie :*

$$\mathit{temps} \prec \mathit{delta}\text{-}$$

En résumé et en tenant compte de l'atomicité, l'ordre partiel suivant est réalisé :

$$\texttt{delta-} \prec \texttt{INTERNES}$$

$$\texttt{sched.delta|\{s, e\}.delta} \prec \texttt{\{s, e\}.notify|<comp>.wait\_x}$$

$$\texttt{sched.run} \prec \texttt{sched.delta|\{s, e\}.delta}$$

$$\texttt{sched.temps|<comp}_1\texttt{>.tick|<comp}_n\texttt{>.tick} \prec \texttt{sched.delta-|\{s, e\}.delta-}$$

Il ne faut pas définir de priorité telle que :

$$\texttt{sched.delta} \prec \texttt{sched.delta-}$$

En effet, il y aurait dans ce cas un blocage, car les actions internes étant prioritaires, elles ajouteraient un drapeau au système or l'ordonnanceur en phase de mise à jour à son propre drapeau déjà actif.

L'exemple de la figure 8.5, page 48, est réutilisé pour illustrer la construction des priorités.

Il suffit d'inventorier les interactions d'instructions internes et les interactions entre les notifications d'événements ou de signaux, `{s, e}.notify`, d'une part et les instructions bloquantes, ici `wait`, d'autre part.

Pour chaque `delta-` du système, la priorité avec chaque élément du premier groupe est définie. De même, la priorité entre les éléments du second groupe et `delta` est définie. Enfin les deux priorités de l'ordonnanceur sont définie.

La classe des instructions internes et des interactions de notification dans les composants, est la suivante :

```
P0.notify0
P0.write1T P0.write1F
P1.notify1
P1.write2T P1.write2F
P2.notify1
P2.read2T P2.read2F
P2.read4T P2.read4F
P2.read6T P2.read6F
P2.notify8
```

Et enfin les `wait` forment la seconde classe. Chacun des éléments est en interaction avec le `notify` d'un événement ou d'un signal :

```
P0.wait2
P1.wait0 P1.wait3
P2.wait0 P2.wait3 P2.wait5
```

L'interaction `P2.wait7` est synchronisée avec `sched.temps` puisqu'il s'agit d'une temporisation.

8.5 Composition du modèle

Les paragraphes précédents ont permis de voir comment chaque structure de SystemC et chaque instruction d'un processus participe à la construction du modèle à base de système à transitions. Il faut désormais composer chaque traduction afin de les unifier et de les lier pour qu'elle coopère en un modèle équivalent. L'application Γ est définie à cet effet et reprend une partie des définitions vu précédemment.

Définition 26 (Application Γ). *La transformation Γ d'une architecture SystemC A vers un système à composants S est définie par :*

$$\Gamma : A \rightarrow S$$

$$\Gamma := \left\{ \begin{array}{l} B := \left(\bigcup_{m \in M_A} \Phi_B(m)\right) \cup \left(\bigcup_{\sigma \in \Sigma_A} \Phi_B(\sigma)\right) \cup \Phi_B(scheduler) \\ IM := \left\{ \begin{array}{l} \bigcup_{m \in M_A} \bigcup_{\sigma \in \Sigma_A} \Phi_{IC}(\Pi_m, \sigma) \cup \\ \bigcup_{m \in M_A} \Phi_{IC}(\Pi_m, scheduler) \ \cup \ \bigcup_{\sigma \in \Sigma_A} \Phi_{IC}(\sigma, scheduler), \\ \bigcup_{m \in M_A} \Phi_{IC^-}(\Pi_m) \cup \bigcup_{\sigma \in \Sigma_A} \Phi_{IC^-}(\sigma) \cup \Phi_{IC^-}(scheduler) \end{array} \right\} \\ \\ U := U^X \wedge \bigwedge_{a \in IC} t^a(U^a) \wedge \{a \prec a'\}, \ \forall a, a' \in IC \end{array} \right.$$

8.6 Schémas de traduction

La structure d'arbre décrite en 7.2, page 34 est utilisée. Pour chaque processus de la table des processus de SystemC, le noeud de l'arbre correspondant est récupéré.

La transformation est appliquée au noeud puis récursivement sur le reste de la chaîne attaché à ce noeud. Cette chaîne décrit une séquence d'instruction.

8.6.1 Interactions de contrôle

Elles sont issues des instructions de contrôle, c'est à dire de la classe I_G.

Les instructions de contrôle sont par exemple les mots clés `while, if, do` présents dans le programme SystemC. La condition qui est exprimée entre parenthèses à la suite doit être exprimée dans la garde du programme Prometheus. La fonction de transition est alors considérée comme vide, sauf si une telle fonction peut être détectée dans la condition elle-même. C'est le cas par exemple dans l'expression `if(x--) call_true() ;` où une fonction est incluse dans l'expression à tester. Pour des raisons de simplicité, ce cas ne sera pas pris en charge par le compilateur dès le début de son développement.

Comme il s'agit d'un test, deux branches du programme sont possibles, selon que le test soit vrai ou faux, il y a donc un deuxième arc associé à cet état dans l'automate d'états fini qui représente le processus étudié. Sa garde doit vérifiée que l'automate se trouve bien dans le même état mais que la condition du test est fausse. Comme il n'y a pas d'autre choix possible, chaque sommet ne peut être la source que de deux arcs au plus.

Les deux lignes suivantes donnent la traduction de l'exemple précédent :

```
ACTION act_x IF state=x AND x DO state:=state+1 ;
ACTION act_y IF state=x AND not x DO state:=state_y ;
```

Lorsque des fonctions dans les expressions de test sont autorisées, il faut ajouter dans notre exemple `x=x-1 ;` dans les deux expressions `DO` précédentes. Dans le cas de prédécrémentation, il faut ajouter l'interaction sur l'arc précédent, celui dont l'état destination est l'état courant.

Equivalence pour les relations

Étant donné la représentation et l'abstraction des données, la régle d'équivalence des expressions sur les variables peut être écrite.

Définition 27 (ϕ d'une expression à deux termes entiers à résultat dans $\{0,1\}$). *Soit e une expression entre deux variables entières (x, y), éventuellement constantes, dont l'évaluation est booléenne.*

$e = xRy$, avec $R \in \{<; <=; =; != ; >=; >\}$

La traduction de e est donnée par l'expression $p = \phi(x)\phi(R)\phi(y)$. $\phi(x)$ et $\phi(y)$ ont été définis précédemment en 8.2.5. L'application $\phi(R)$ est donnée par les lignes 2 et 4 du tableau 8.1.

x	y	<	<=	==
ϕ(x)	ϕ(y)	(not x) and y	(not x) or y	x == y
x	y	!=	>=	>
ϕ(x)	ϕ(y)	not (x == y)	x or (not y)	x and (not y)

TAB. 8.1 – Transformation des relations entre variables entieres

Définition 28 (ϕ d'une expression booléenne ET ou OU). *Soit eb une expression booléenne de la forme x and y, respectivement x or y, $\phi(eb) = \phi(x)\ AND\ \phi(y)$, respectivement $\phi(eb) = (\phi(x)\ OR\ \phi(y))$.*

Les parenthèses sur la forme en OR permettent de préserver l'expression de la priorité du AND. Enfin une expression à plus de deux termes entiers à résultat dans $\{0,1\}$ est équivalente à une expression booléenne dont les termes sont des expressions à deux termes entiers.

Instruction `while`

La structure C++ est : `while(<cond>) do {<bloc d'instructions>}`. Le cas simple où la condition ne comporte pas d'instruction modifiant l'état du système et où le bloc d'instructions est de taille n est d'abord considéré. La traduction en Prometheus est la suivante :

```
ACTION whileT_{x_1} IF state=x AND φ(<cond>) AND flag
DO state:=state+1, tick:=false ;
<code Prometheus associé au bloc>
ACTION while_ret_{x_2} IF state=x+n AND flag
DO state:=x, tick:=false ;
ACTION whileF_{x_3} IF state=x AND not φ(<cond>) AND flag
DO state:=x+n+1, tick:=false ;
```

$\texttt{act}_{x_1}$ est le label pour l'interaction associée à la condition vraie, c'est à dire non nulle. L'état du système est incrémenté normalement dans ce cas. $\texttt{act}_{x_3}$ est celui pour l'interaction où la condition est fausse. L'état d'entrée pour le bloc est la valeur $x + 1$. $\texttt{act}_{x_3}$ permet de revenir sur le test du `while` lorsque l'exécution du bloc est finie.

Instruction `if [else]`

La structure C++ est :

```
if(<cond>) {<bloc d'instructions 1>}
[else {<bloc d'instructions 2>}]
```

Le cas simple où la condition ne comporte pas d'instruction modifiant l'état du système et où les blocs d'instructions sont de tailles n_1 et n_2 respectivement est d'abord considéré.

La traduction en Prometheus est la suivante :

```
ACTION ifT_{x_1} IF state=x AND φ(<cond>) AND flag
DO state:=state+1, tick:=false ;
<code Prometheus associé au bloc 1>
ACTION ifF_{x_2} IF state=x AND not φ(<cond>) AND flag
DO state:=x+n_1+1, tick:=false ;
<code Prometheus associé au bloc 2>
```

$\texttt{act}_{x_1}$ est le label pour l'interaction associée à la condition vraie, c'est à dire non nulle. L'état du système est incrémenté normalement dans ce cas. $\texttt{act}_{x_2}$ est celui pour l'interaction où la condition est fausse. L'état d'entrée pour le bloc 1 est la valeur $x + 1$, et $x + n_1 + 1$ pour le bloc 2.

Instructions `do while` et `do until`

Les structures étant similaire pour les instructions `do while` et `do until` à celle vu pour le `while`, elles ne seront pas présentée ici.

8.6.2 Interactions idempotentes pour l'état du système

Des interactions telles que `cout << ...` ou `printf(...)` ne sont là que pour l'utilisateur qui effectue du test et du débuggage. Ces instructions là appartiennent à la classe $I_{F^{id}}$ et peuvent être tout simplement ignorées. L'arc associée et l'état successeur, qui est identique au prédécesseur, sont alors supprimés du graphe. L'arc du successeur est alors rattaché à l'état prédécesseur.

Un autre cas est celui de la déclaration de variables. Dans ce cas, l'état du système n'est pas modifié mais il faut ajouter la variable déclarée à l'ensemble de X des variables du système à transitions.

8.6.3 Interactions qui modifient l'état d'un composant

Il s'agit des interactions qui modifient les variables du composant mais pas non pas d'effet de bord immédiat sur les autres composants. Elles sont générées à partir des instructions de la classe $I_{F^{ac}}$. Elles peuvent être de deux formes : appels de méthodes internes au composant, ou opérations arithmétiques classiques du C++. Ces interactions ne modifie pas la garde de la transition à laquelle elles sont associées mais modifient la fonction de transition. Dans le cas des opérateurs mathématiques, il faut convertir la fonction en utilisant les opérateurs mathématiques disponibles dans Prometheus, uniquement des booléens.

Dans le cas de l'appel de méthode, il faut faire de même mais avec l'ensemble des instructions de la méthode. Cela peut vite devenir ingérable si la méthode est longue ou réalise des tests ou des boucles. Il est donc plus simple de voir la méthode comme un processus fils synchronisé avec le processus principal. Ce processus est synchronisé, en entrée et en sortie, par des rendez-vous binaire avec le processus principal. Celui se poursuit en récupérant l'état dans lequel c'est terminé le processus fils (valeur de retour de la méthode).

Ce cas ne sera pas implémenté dans un premier temps dans le compilateur cas il est relativement complexe.

Il reste donc à convertir les expressions mathématiques en SystemC en leur équivalent en Prometheus.

8.6.4 Interactions qui interagissent avec le reste du système

Ces interactions sont issues des instructions synchronisantes appartenant donc aux classes $I_{F^{s}{}_{c}}$ et $I_{F^{s}{}_{i}}$.

Il faut traiter les méthodes qui utilisent les signaux extérieurs, tels que `write`, `wait`, et qui ne modifient pas l'état interne du composant, et `read` qui modifie l'état interne d'un composant. Ces méthodes sont associées à des interactions incompletes du système qui interviennent dans des interactions.

Il est donc nécessaire de trouver avec quel(s) autre(s) composant(s) elles interagissent afin de pouvoir déterminer les contraintes du système dans le modèle Prometheus.

Dans la table de connexion de l'exemple que nous avons choisi, les composants `emitter` et `receiver` sont connectés au signal `wire`. De plus, `emitter` y est connecté via un port de type `sc_in` tandis que `receiver` l'est par un port de type `sc_out`. Les rôles de chacun peuvent ainsi être déterminés et donc les interactions à mettre en place. Les instructions sur un port `<port>` définissent les interactions avec le signal `<signal>` auquel est attaché le port.

Les appels aux fonctions SystemC vont maintenant être détaillés. Pour chacun, sa classe d'appartenance sera précisée.

Méthode `write`, classe $I_{F^s i}$

La méthode `write`, du processus du module `<module>`, est susceptible de modifier la valeur du signal `<signal>`. Elle sera donc en interaction avec les actions `write` du signal. Comme la valeur écrite n'est pas, à priori, connue, deux interactions sont possibles.

La forme générale de l'instruction est `<port>.write(<var>)`.

```
ACTION write<var>=T IF state=x AND <var> AND flag
DO state:=state+1, tick:=false ;
ACTION write<var>=F IF state=x AND ! <var> AND flag
DO state:=state+1, tick:=false ;
```

Les interactions de write$_{<var>=T}$ et write$_{<var>=F}$ sont définies respectivement avec les actions write$_T$, write$_F$.

```
INTERACTIONS
<module>.write<var>=T|<signal>.writeT
<module>.write<var>=F|<signal>.writeF

INCOMPLETE <module>.writeT <module>.writeF
```

Méthode `read`, classe $I_{F^s i}$

La méthode `read`, du processus du module `<module>`, lit la valeur du signal `<signal>`. Elle sera donc en interaction avec les actions `read` du signal. Comme la valeur lue n'est pas, à priori, connue, deux interactions sont possibles.

La forme générale de l'instruction est `<var> = <port>.read()`.

```
ACTION readT IF state=x AND flag
DO state:=state+1, <var>:=true ;
ACTION readF IF state=x AND flag
DO state:=state+1, <var>:=false ;

INTERACTIONS <module>.readT|<signal>.RDT <module>.readF|<signal>.RDF

INCOMPLETE <module>.readT <module>.readF
```

Méthode `wait()` sur un signal, classe $I_{F^s i}$

La méthode `wait` est en interactions avec l'ensemble des signaux `<signal`$_i$`>` sur lesquels est sensible le processus du module `<module>`. Un seul signal modifié débloque l'instruction et le changement de δ-pas est alors nécessaire. Si il n'y a pas de modification, le composant reste oisif.

```
ACTION wait IF state=x AND !flag
DO state:=state+1, tick:=false ;

INTERACTIONS <module>.wait|<signal_i>.notify

INCOMPLETE <module>.wait
```

Il se produit un problème si l'action suivante nécessite d'avoir le drapeau actif, il faut donc ajouter une action idempotente qui prend le drapeau.

```
ACTION run IF state=x+1 AND !flag
DO flag:=true ;
```

Méthode `wait(<T>,<U>)` pour le temps, classe $I_{F^s i}$

La méthode `wait`,dans le processus du module `<module>`, est en interactions avec l'ensemble des signaux `<signal`$_i$`>`. Comme il s'agit d'un délai, le changement de δ-pas est forcé. `<T>` désigne le temps d'attente et `<U>` son unité. Prometheus n'est pas temporisé, ces paramètres sont donc ignorés.

```
ACTION tick IF true DO tick:=true ;
ACTION delai IF state=x AND !flag AND tick
DO state:=state+1, flag:=true, tick:=false ;

INTERACTIONS <scheduler>.temps|<module_1>.tick|···|<module_n>.tick

INCOMPLETE <module>.delai
```

Chapitre 9

Réalisation du compilateur

9.1 Architecture

La réalisation du compilateur est faite en adéquation avec la philosophie de SystemC et des systèmes compositionnels. Le choix du langage, C++, a été dicté par le contexte, environnement C et C++, et par l'aspect objet du langage qui correspond à l'aspect composant et modularité des systèmes étudiés. Le générateur est supposé, ainsi construit, être extensible.

La classe `sc2prom` représente l'application, c'est à dire notre générateur. Elle instancie un système, représenté par la classe `PromSystem`. Ce système est l'unique instance possible de la classe car il n'est possible d'analyser qu'un seul système SystemC à la fois.

Un objet de la classe `PromSystem` est composé d'objets `Component` qui modélisent les composants. Les signaux sont des composants puisqu'ils ont un comportement propre et ont des interactions avec le reste du système. La classe `Signal` hérite donc de la classe `Component`. De même, les classes `Module`, pour les modules de SystemC, et `Sched`, pour l'ordonnanceur, dérivent de la classe `Component`. Le système est également composé d'un modèle d'interaction et d'un ensemble de contraintes.

Le modèle d'interaction est représenté par la classe `Interaction_Model` qui contient deux ensembles d'interactions, IC et IC^{-}. Chaque interaction est un objet de la classe du même nom et est un ensemble d'actions liées.

La couche de contraintes est modèlisée par la classe `Constraints` qui regroupe trois ensembles, un ensemble de priorité issues de la classe `Priority`, un ensemble de contraintes d'état, classe `State_Constraint` et un ensemble de contraintes sur les interactions, classe `Interaction_Constraint`.

Pour des raisons de simplicité, les modules comportent des processus ce qui permet une transition progressive du modèle SystemC vers le modèle compositionnel à base de système à transitions.

Les variables des composants sont modélisées par la classe `Variable`.

Enfin l'objet le plus complexe est une action, une interaction de base, et est utilisé pour décrire le comportement des composants. Chaque objet est décoré durant l'analyse. La décoration consiste à définir un nom unique au sein du composant ainsi que la garde et la fonction de transition et respectivement le renforcement et le raffinement de ces deux derniers pour définir l'automate propre au processus en cours

d'analyse. Les actions ainsi définies sont celles utilisées pour créer les interactions.

L'organisation des classes peut être représentée par le diagramme UML suivant :

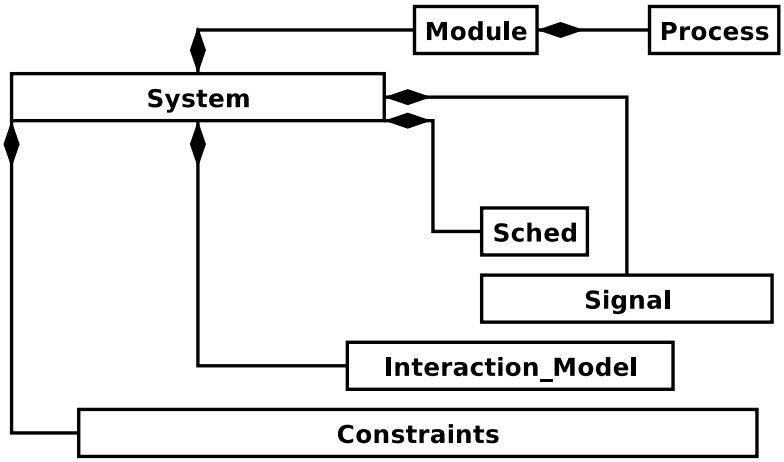


FIG. 9.1 – Diagramme de classe du générateur de code

Un ensemble de diagrammes de classes plus détaillés est disponible en annexe.

9.2 Fonctionnement

Au lancement, le programme SystemC à analyser est passé en paramètre. L'analyseur de Matthieu Moy est exécuté avec ce paramètre. Puis une instance de la classe PromSystem est créée à partir de l'ensemble des objets du système analysé.

Pour chaque objet de SystemC, un objet équivalent dans le système compositionnel est créé avec comme paramètre l'objet SystemC à modéliser lui-même. Une partie des interactions et des contraintes sont générées à la volée lors de l'analyse de l'architecture.

Une fois l'analyse de tous les composants effectuée, le système génère les interactions et les contraintes restantes, notamemment celles liées à l'ordonnancement de SystemC et non pas à l'architecture du système.

Enfin, l'application lance une opération de génération de code sur le système en mémoire qui lance récursivement cette opération sur les éléments qui le composent.

L'implémentation de l'analyseur utilisée est la version 0 révision 10. La dernière version disponible est la révison 32 mais pour l'utiliser des modifications sur le générateur sont nécessaires. Il a donc été préféré de poursuivre le développement avec la révision 10. Les révisions 11 et 12 modifient l'interface entre l'analyseur et le générateur. Les révisions 13 à 15 sont une migration vers GCC 3.4.1. Enfin, les révisions 16 à 32 sont des évolutions et des corrections.

Chapitre 10

Étude de cas

La méthode présentée au chapitre 8 va être appliquée à l'exemple donné en 6.1, page 19. Tout d'abord le générateur de code construit l'ordonnanceur puis parcourt les objets pour construire les signaux de l'architecture. Ensuite, il reparcourt les objets pour construire les modules et leurs processus. Il trouve alors :

- Un signal booléen,
- Deux composants et leur unique processus :
  - `my_emitter` est son processus `emitter_process`,
  - `my_receiver` est son processus `receiver_process`.

Le nom du projet SystemC détermine le nom du système compositionnel. Il comporte quatre composants :

- les deux modules de SystemC,
- le signal, qui est modélisé comme un composant,
- l'ordonnanceur, qui est modélisé comme un composant.

Chaque composant issu d'un module comporte des variables d'état, une variable `tick` pour modéliser le temps et un drapeau par processus qui porte le nom du processus dont il dépend. La traduction des processus étant assez verbeuse, elle ne sera pas donnée dans ce rapport.

Le signal est généré statiquement ainsi que l'ordonnanceur tels qu'ils ont été définis dans les chapitres précédents.

```
COMPONENT signal_0
/* -- This channel is a boolean signal -- */
VAR current futur delta;
TRANSITIONS
        ACTION writeT IF true DO futur:=true, delta:=false;
        ACTION writeF IF true DO futur:=false, delta:=false;
        ACTION delta_ IF current!=futur DO delta:=true;
        ACTION notify IF delta DO ;
        ACTION delta IF delta DO delta:=false, current:=futur;
END signal_0

COMPONENT sched
VAR flag;
TRANSITIONS
```

```
        ACTION temps IF !flag DO ;
        ACTION delta_ IF !flag DO flag:=true;
        ACTION delta IF flag DO ;
        ACTION run IF flag DO flag:=false;
END sched
```

Les interactions IC, non unaires, sont construites à la volée lors de la construction des interactions des processus. Il s'agit des interactions entre le signal et l'ordonnanceur, l'émetteur et le récepteur pour respectivement la mise à jour, les deux écritures et les deux attentes. L'évolution du temps génère une interaction entre les trois composants.

```
INTERACTIONS
sched.temps|my_emitter.aTick|my_receiver.aTick
sched.delta_|signal_0.delta_
signal_0.delta|sched.delta
my_emitter.writeT2|signal_0.writeT
my_emitter.writeF2|signal_0.writeF
my_emitter.writeT6|signal_0.writeT
my_emitter.writeF6|signal_0.writeF
my_receiver.wait1|signal_0.notify
my_receiver.wait2|signal_0.notify;
```

Les interactions IC^- sont déclarées. IC^+ est alors implicitement défini. Dans notre exemple, toutes les interactions n-aires définies précédemment ont leurs membres appartenant à IC^-.

```
INCOMPLETE
sched.delta_
sched.delta
sched.temps
signal_0.delta_
signal_0.writeT
signal_0.writeF
signal_0.notify
signal_0.delta
my_emitter.writeT2
my_emitter.writeF2
my_emitter.aTick|my_receiver.aTick
my_emitter.writeT6
my_emitter.writeF6
my_receiver.wait1
my_receiver.wait2;
```

Il faut garantir la non préemption, la contrainte d'état impose donc qu' il y est au plus un drapeau actif dans les composants actifs, ordonnanceur compris, et le synchronisme entre la phase de mise à jour de l'ordonnanceur et celle du signal.

```
ASSERT
/* No more than one flag enable */
   (my_emitter.emitter_process AND
             !my_receiver.receiver_process AND !sched.flag
OR !my_emitter.emitter_process AND
              my_receiver.receiver_process AND !sched.flag
OR !my_emitter.emitter_process AND
             !my_receiver.receiver_process AND  sched.flag
```

```
OR !my_emitter.emitter_process AND
             !my_receiver.receiver_process AND !sched.flag)
/* Can enable tick only when waiting for */
AND  (!my_emitter.s3 AND !my_emitter.s2 AND  my_emitter.s1 AND
       my_emitter.s0 AND my_emitter.tick AND !my_emitter.emitter_process
   OR !my_emitter.s3 AND  my_emitter.s2 AND !my_emitter.s1 AND
       my_emitter.s0 AND my_emitter.tick AND !my_emitter.emitter_process
   OR !my_emitter.s3 AND  my_emitter.s2 AND  my_emitter.s1 AND
       my_emitter.s0 AND my_emitter.tick AND !my_emitter.emitter_process
   OR !my_emitter.tick)
/* When waiting, don't have the flag */
AND !(!my_receiver.s2 AND !my_receiver.s1 AND  my_receiver.s0
       AND my_receiver.receiver_process
      OR !my_receiver.s2 AND  my_receiver.s1 AND !my_receiver.s0
          AND my_receiver.receiver_process)
/* Non-existent state of my_emitter */
AND !( my_emitter.s0 AND !my_emitter.s1 AND !my_emitter.s2 AND my_emitter.s3)
AND !(!my_emitter.s0 AND  my_emitter.s1 AND !my_emitter.s2 AND my_emitter.s3)
AND !( my_emitter.s0 AND  my_emitter.s1 AND !my_emitter.s2 AND my_emitter.s3)
AND !(!my_emitter.s0 AND !my_emitter.s1 AND  my_emitter.s2 AND my_emitter.s3)
AND !( my_emitter.s0 AND !my_emitter.s1 AND  my_emitter.s2 AND my_emitter.s3)
AND !(!my_emitter.s0 AND  my_emitter.s1 AND  my_emitter.s2 AND my_emitter.s3)
AND !( my_emitter.s0 AND  my_emitter.s1 AND  my_emitter.s2 AND my_emitter.s3)
/* Non-existent state of my_receiver */
AND !(my_receiver.s2 AND !my_receiver.s1 AND !my_receiver.s0)
AND !(my_receiver.s2 AND !my_receiver.s1 AND  my_receiver.s0)
AND !(my_receiver.s2 AND  my_receiver.s1 AND !my_receiver.s0)
AND !(my_receiver.s2 AND  my_receiver.s1 AND  my_receiver.s0)
/* Non-existent state of signal_0 */
AND (sched.flag AND signal_0.delta AND
      (signal_0.current AND !signal_0.futur OR
      !signal_0.current AND signal_0.futur)
     OR !signal_0.delta);
```

Enfin, les priorités permettent de rendre déterministe et localement confluent le système. Les interactions internes sont donc prioritaires sur la mise à jour par l'ordonnanceur.

```
sched.delta_|signal_0.delta_ < my_emitter.modify0;
sched.delta_|signal_0.delta_ < my_emitter.whileT1;
sched.delta_|signal_0.delta_ < my_emitter.writeT2|signal_0.writeT;
sched.delta_|signal_0.delta_ < my_emitter.writeF2|signal_0.writeF;
sched.delta_|signal_0.delta_ < my_emitter.wait3;
sched.delta_|signal_0.delta_ < my_emitter.modify4;
sched.delta_|signal_0.delta_ < my_emitter.wait5;
sched.delta_|signal_0.delta_ < my_emitter.writeT6|signal_0.writeT;
sched.delta_|signal_0.delta_ < my_emitter.writeF6|signal_0.writeF;
sched.delta_|signal_0.delta_ < my_emitter.wait7;
sched.delta_|signal_0.delta_ < my_emitter.while_ret8;
sched.delta_|signal_0.delta_ < my_emitter.whileF1;

sched.delta_|signal_0.delta_ < my_receiver.whileT0;
/*sched.delta_|signal_0.delta_ < my_receiver.run2;*/
sched.delta_|signal_0.delta_ < my_receiver.run3;
sched.delta_|signal_0.delta_ < my_receiver.while_ret3;
```

```
sched.delta_|signal_0.delta_ < my_receiver.whileF0;

sched.delta|signal_0.delta < my_receiver.wait1|signal_0.notify;
sched.delta|signal_0.delta < my_receiver.wait2|signal_0.notify;

sched.run < signal_0.delta|sched.delta;

sched.temps|my_emitter.aTick|my_receiver.aTick < sched.delta_|signal_0.delta_;

my_emitter.wait3 < my_receiver.run3;
my_emitter.wait5 < my_receiver.run3;
my_emitter.wait7 < my_receiver.run3;
```

Prometheus est alors exécuté sur le programme généré et la trace suivante est obtenue.

```
1   [palix@lama Pro-example]$ ./prometheus etude.system
2   Warning: G(my_emitter.writeT6)' = false
3   Warning: G(my_emitter.whileF1)' = false
4           my_emitter deadlock-free in (true)
5   Warning: G(my_receiver.whileF0)' = false
6           my_receiver deadlock-free in (true)
7           signal_0 deadlock-free in (true)
8           sched deadlock-free in (true)
9           Refining component invariants.
10  Warning: G(my_emitter.writeT6|signal_0.writeT)' = false
11  ....... done.
12          my_emitter || my_receiver || signal_0 || sched deadlock-free
13          Component my_receiver is not controllable wrt ...
14          [...]
15          Component my_emitter is not controllable wrt ...
16          [...]
17          Component my_emitter is maybe not deadlock-free ...
18          ... in my_emitter || my_receiver || signal_0 || sched
19          Component my_receiver is maybe not deadlock-free ...
20          Component signal_0 is maybe not deadlock-free ...
21          Component sched is maybe not deadlock-free ...
22          my_emitter is maybe not live ...
23          ... in my_emitter || my_receiver || signal_0 || sched
24                  possibly unfair components: my_receiver sched signal_0
25          my_emitter is (non-compositionally) live ...
26          my_receiver is maybe not live ...
27                  possibly unfair components: my_emitter sched signal_0
28          my_receiver is (non-compositionally) live ...
29          signal_0 is maybe not live ...
30                  possibly unfair components: my_emitter my_receiver sched
31          signal_0 is (non-compositionally) live ...
32          sched is maybe not live ...
33                  possibly unfair components: my_emitter my_receiver signal_0
34          sched is (non-compositionally) live ...
35          System is locally confluent and component-deterministic
36  Terminating...
```

Chapitre 11

Conclusion

L'objectif annoncé était la conception et l'implémentation d'un outil de traduction d'architecture SystemC vers un modèle compositionnel à base de systèmes à transitions avec contraintes.

La méthode utilisée par cet outil reste simple et son implémentation fournit un outil exploitable qui associé à Prometheus permet de donner des résultats sur la validité d'architecture SystemC.

L'important temps nécessaire à la mise en oeuvre de l'analyseur SystemC et à la compréhension de SystemC et des systèmes à transitions, ne m'a pas permis de tester l'outil sur des exemples complexes. Il faut à présent évaluer l'impact des abstractions notamment sur les variables entières.

L'objectif d'intégration de TLM est fortement dépendant de la représentation des données. En effet, le comportement des composants va dépendre des données qui transitent. Or si la représentation de celles-ci est trop abstraite, le modèle ne sera plus pertinent. Le problème est que justement les données de TLM sont des structures de données. Ce problème sort alors du cadre d'un stage. Le type d'interaction peut cependant facilement être modélisé puisqu'il s'agit de deux rendez-vous binaires, l'un entre le point d'entrée de la méthode de l'esclave et l'appel de celle-ci par le maître, l'autre entre le point de sortie et l'instruction qui suit l'appel dans le maître.

Actuellement, seul les signaux booléens sont pris en compte. Il apparait intéressant de généraliser pour les signaux entiers puis pour les signaux de type arbitraire. Le problème de complexité des données vu avac TLM est alors de nouveau rencontré.

Pour disposer d'un outils susceptible d'être utilisé dans l'industrie, il est nécessaire d'étendre la méthode, pour des architectures SystemC de plus en plus complexe. Il faudrait, entre autre, implémenter la gestion d'expression C++ comme les affectations lors d'instructions de contrôle.

Bibliographie

[1] Compositionality : The significant difference. volume 1536 of *LNCS*. Springer-Verlag, Septembre 1997.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers - Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, 1986.

[3] K. Altisen, G. Gößler, and J. Sifakis. Scheduler modeling based on the controller synthesis paradigm. *Real-Time Systems Journal, special issue on "Control-theoretical Approaches to Real-Time Computing"*, 23(1/2) :55–84, 2002.

[4] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2) :244–263, April 1986. Introduction of Model-checking ; impartiality, justice, fairness.

[5] J.-C. Fernandez, C. Jard, T. Jéron, and C. Viho. An experiment in automatic generation of test suites for protocols with verification technology. *Science of Computer Programming*, 29, 1997.

[6] GNU GCC. Trees : The intermediate representation used by the c and c++ front ends, 2004. http ://gcc.gnu.org/onlinedocs/gccint/Trees.html.

[7] G. Gößler. *Compositional Modelling of Real-Time Systems — Theory and Practice*. PhD thesis, Université Joseph Fourier, Grenoble, France, 2001.

[8] G. Gößler. PROMETHEUS — a compositional modeling tool for real-time systems. In P. Pettersson and S. Yovine, editors, *Proc. Workshop RT-TOOLS 2001*, Aalborg, Denmark, August 2001. Technical report 2001-014, Uppsala University, Department of Information Technology.

[9] G. Gößler and A. Sangiovanni-Vincentelli. Compositional modeling in metropolis. volume 2491 of *LNCS*. Springer-Verlag, 2002.

[10] G. Gößler and J. Sifakis. Component-based construction of deadlock-free systems. In *FSTTCS'03 proceedings*, volume 2914 of *LNCS*. Springer-Verlag, 2003.

[11] G. Gößler and J. Sifakis. Composition for component-based construction modeling. In *proc. FMCO'02*, volume 2852 of *LNCS*. Springer-Verlag, 2003.

[12] Thorsten Grötker, Stan Liao, Grant Martin, and Stuart Swan. *System Design with SystemC(TM)*, volume 1. Kluwer Academic Publishers, 3 edition, 2003.

[13] N. Halbwachs. A tutorial of lustre. Technical report, Vérimag, 1993.

[14] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language lustre. *Proceeding of the IEEE*, 79(9) :1305–1320, September 1991.

[15] Open SystemC Initiative. Systemc v2.0.1 language reference manual, June 2004. http ://www.systemc.org/.

[16] L. Mazaré. Architectures réactives synchrones. Master's thesis, Université Joseph Fourier, Grenoble, France, September 2003.

[17] R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25(3) :267–310, 1983. SCCS.

[18] M. Moy. Automatic verification of soc transactional models. Technical report, STMicroelectronics, 2003.

Annexe A

Installation de l'analyseur

Pré-requis :

Pour installer l'analyseur, il est nécessaire d'avoir les sources de OSCI SystemC et GNU GCC. Elles sont téléchargeables sur Internet. Les versions doivent être **exactement** GCC 3.2.3 et SystemC 2.0.1. Il est également nécessaire d'avoir au moins :

- autoconf >= 2.57
- automake >= 1.6.3
- GNU Make
- Une installation d'origine de GCC 3.2.x
- Une installation d'origine de SystemC 2.0.1

Configuration :

```
mkdir SC2Prom SC2Prom/gcc-3.2.3
cd SC2Prom/gcc-3.2.3
tar zxvf gcc-3.2.3.tar.gz
mv gcc-3.2.3 src
cd ..
tar zxvf systemc-2.0.1.tgz
tar zxvf sc-parser.tgz (ou sc-parser_patched.tgz)
tar zxvf tlminfra_1.4_20040429.tar.gz
tar zxvf TLM_Projects.tgz
tar zxvf my-example.tgz
tar zxvf SC2Prom.tgz
```

Il est important de respecter l'ordre car certains éléments écrase des fichiers précédemment extrait d'une autre archive.

Copiez le fichier `depcomp` de automake de :
`/usr/share/automake-<version>/depcomp`
vers `$SC2Prom/systemc-2.0.1/config`

Le fichier `missing` peut également être copier de la même manière afin d'éviter un message d'alerte.

Vérifiez d'avoir les variables d'environnement `CXX` et `LD_LIBRARY_PATH` initialisées.

Elles peuvent être mise dans votre fichier `.cshrc` avant `source setup.csh`

Sinon initialisez les :

- `setenv CXX g++`
- `setenv LD_LIBRARY_PATH`
- `setenv SC2Prom ‘pwd‘`

Changez de répertoire courant :

```
cd $SC2Prom/sc-parser
```

Paramétrez les répertoire du fichier `setup.csh`, et chargez le avec la commande `source setup.csh`. Vous devez être dans un shell (t)csh.

Par défaut :

```
setenv SCP_TLM_HOME ${SC2Prom}/TLM_Projects
setenv SYSTEMC_HOME ${SC2Prom}/systemc-2.0.1
setenv TLM_INFRA_INST ${SC2Prom}/tlminfra_1.4_20040429
setenv GCC_SSA_HOME_DIR
setenv GCC_MAIN_HOME_DIR  ${SC2Prom}/gcc-3.2.3
```

N'oubliez pas d'indiquer l'emplacement des versions d'origine de SystemC et GCC, par exemple :

```
setenv ORIGINAL_SYSTEMC_HOME /usr/local/systemc-2.0.1
setenv GCC_COMPILER_HOME   /usr
```

Des configurations alternatives, éventuellement partielles, peuvent être définies dans d'autres fichiers :

- `setup.csh.<os>` pour d'autres systèmes d'exploitation
- `setup.csh.<host>` pour une configuration en fonction de l'hôte

Ce mécanisme n'est utile que dans un environnement réseau hétérogène où plusieurs hôtes fonctionnant sur des systèmes d'exploitation différents partagent le même projet.

Installation : Premièrement, modifiez GCC :

```
make gcc-patch
make gcc-configure
make gcc-bootstrap (Prend beaucoup de temps !)
make gcc-make (Pour construire la librairie libfullgcc)
```

Modifiez ensuite SystemC :

```
make sc-patch
make sc-configure
make sc-make
```

Il est nécessaire de modifier les fichiers :
`$SC2Prom/tlminfra_1.4_20040429/bin/tlm_configure`
et `$SC2Prom/sc-parser/parser/include/scp-config.h`.

Pour le premier, il faut mettre en commentaire les lignes 992 et 999 concernant la suppression d'include de SystemC et pour le second, il faut mettre en commentaire la ligne 69, `#define SCP_USING_TAC`.

A présent, il faut reconfigurer la librairie TLM :

```
make tlm-patch
make tlm-configure
```

Il est cependant possible d'utiliser la version non modifiée de SystemC :

```
make tlm-configure-original
make tlm-make-original
make tlm-run-original
```

Il est désormais possible de construire le compilateur

```
make sc2prom
make run-sc2prom
```

L'exécutable est placé dans `bin-$TARGET_ARCH/sc2prom`.

Pour une utilisation normale de TLM

La configuration de `TLM_INFRA` se fait par les fichiers du répertoire :
`${SC2Prom}/tlminfra_1.4_20040429/etc`

Il s'agit essentiellement des fichiers `TLM_Compiler` et `TLM_SystemC`.

Pour construire une architecture, il faut se déplacer dans le répertoire de la plateforme. Par exemple :
`${SC2Prom}/TLM_Projects/SC_PARSER/plateform/my_example`

L'exécution en shell `tcsh` de `./plateforme` crée un répertoire temporaire dans le répertoire des projets. Celui-ci contient un sous-répertoire pour chaque couple projet–plateforme, ici :
`SC_PARSER-my_example`

Il contient un fichier `TLMsetup.csh` et un `makefile`. Pour compiler l'architecture, il faut exécuter les commandes :

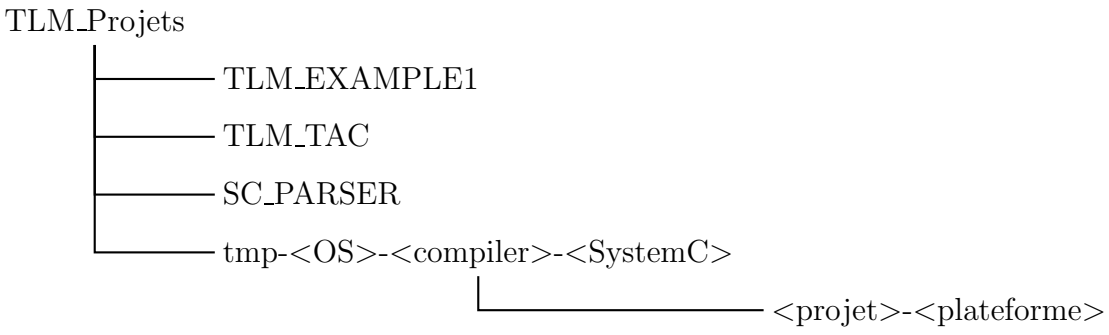
- `source TLMsetup.csh`
- `make`

L'exécution de la plateforme est obtenue par :
`./TLMrun`

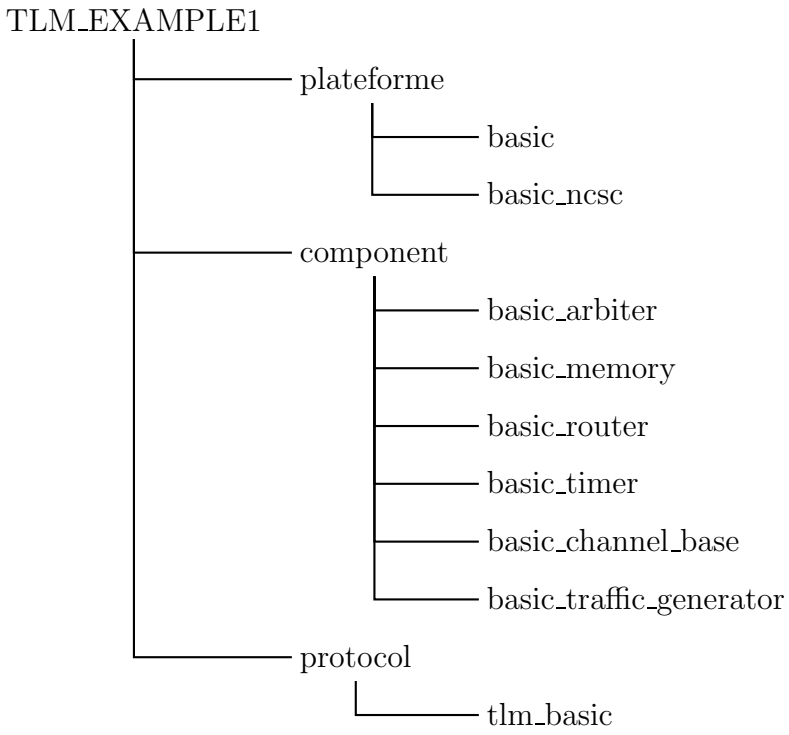
Annexe B

Structure des projets TLM

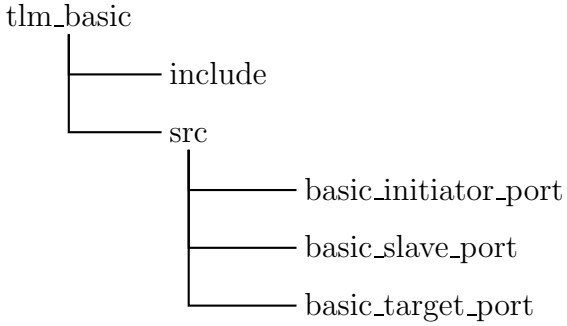
Le répertoire `TLM_Projets` contient l'ensemble des projets ainsi que les répertoires temporaires générés lors de la compilation d'un projet. Dans notre cas, il y a trois projets, `TLM_EXAMPLE1`, `TLM_TAC`, `SC_PARSER`.



Typiquement, un projet est composé de trois éléments :
– des plateformes, qui constituent le coeur du projet et définit des architectures,
– des composants, qui sont utilisés par les plateformes,
– des protocoles, pour la communication inter-composants.



Un protocole définit un ensemble de port et d'interface qui sont utilisés par les composants pour communiquer.



Annexe C

Détail des classes

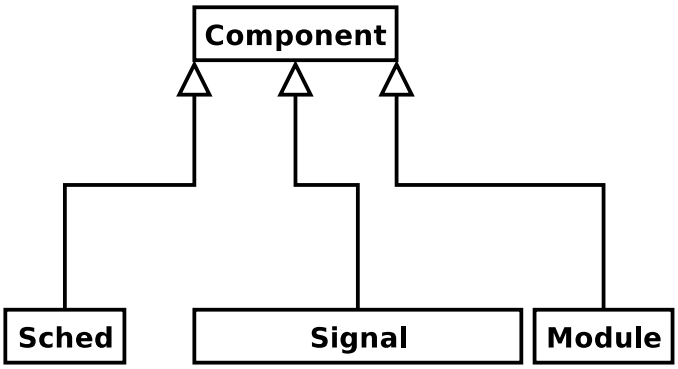


Fig. C.1 – Héritages de la classes Component

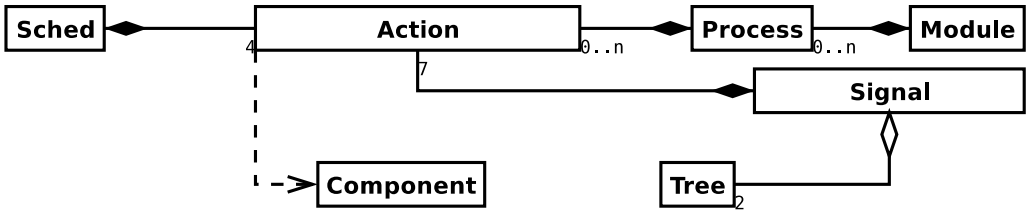


Fig. C.2 – Classes relatives à la classe Action

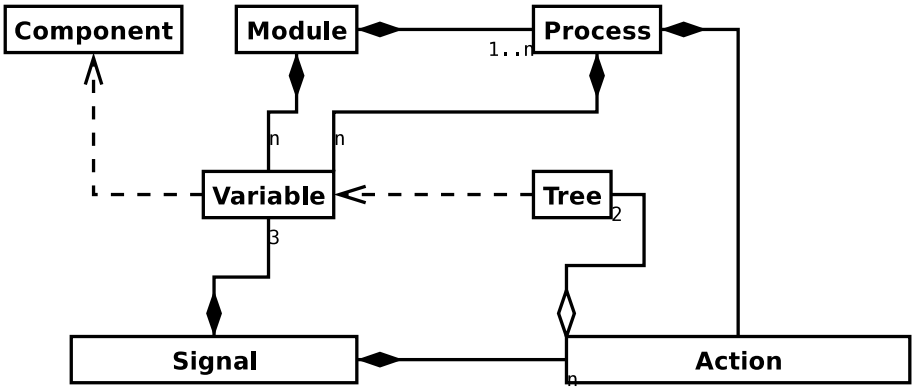


FIG. C.3 – Classes relatives à la classe Variable

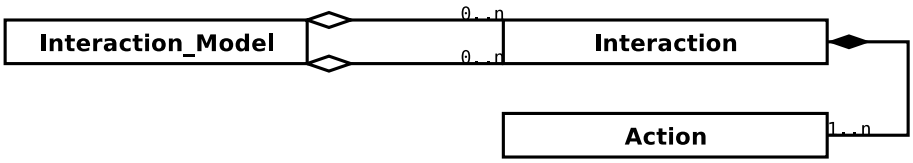


FIG. C.4 – Représentation du modèle d'interaction

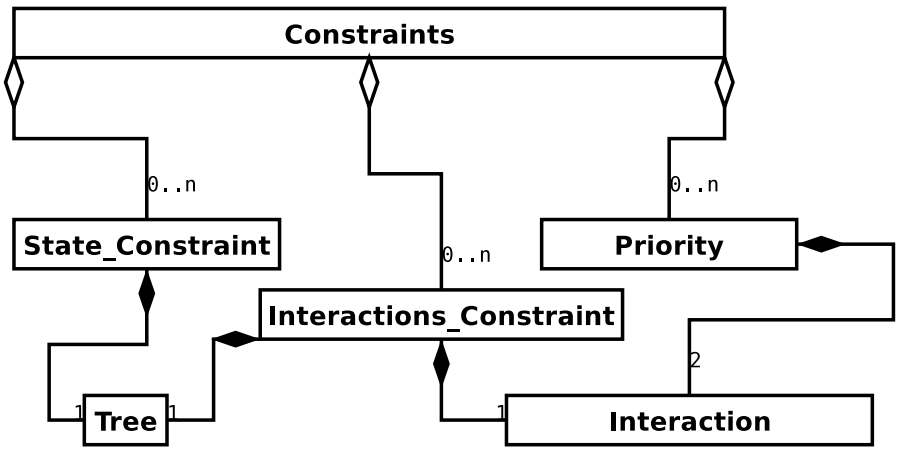


FIG. C.5 – Représentation des contraintes

Annexe D

Organigramme de l'INRIA

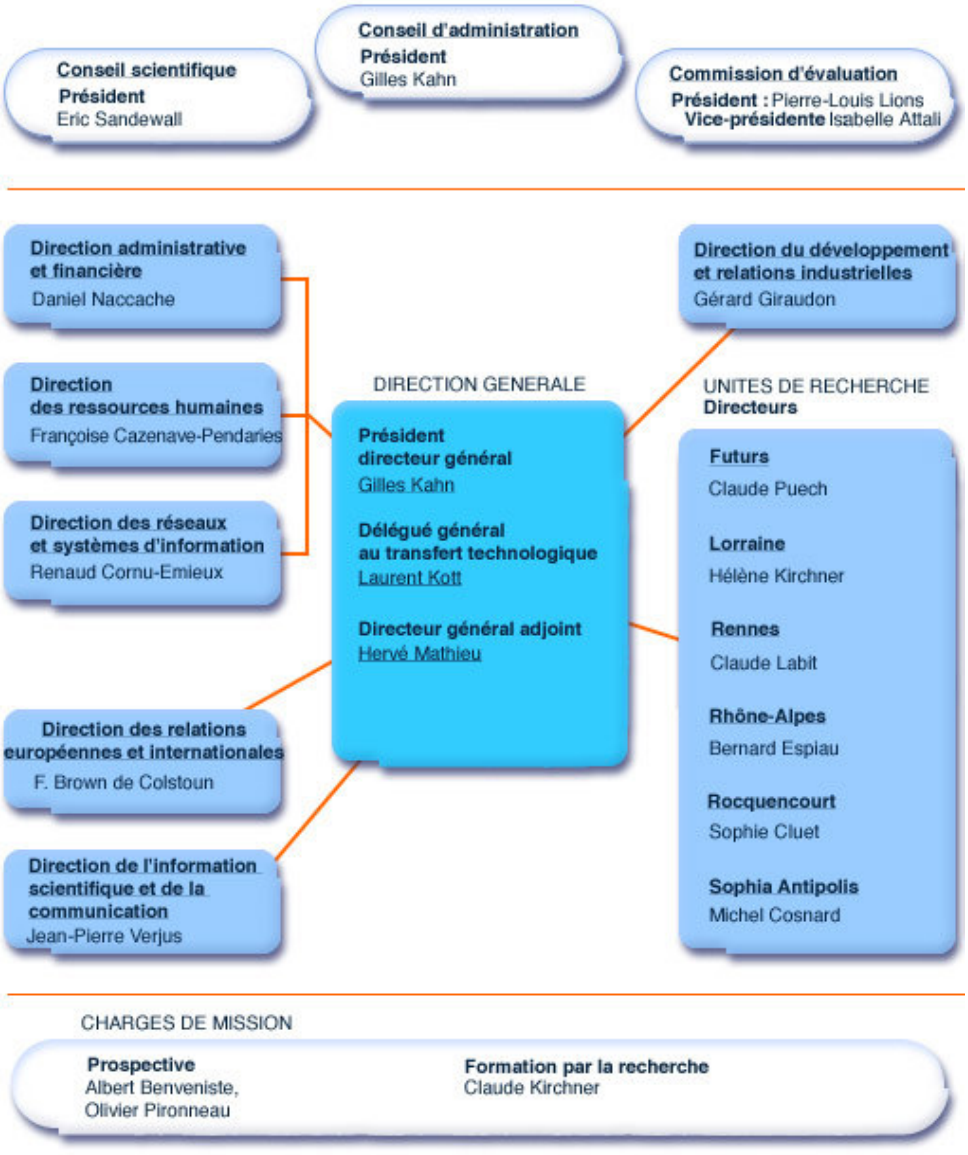


FIG. D.1 – Organigramme hiéarchique de l'INRIA

Annexe E

Compte d'exploitation du projet

Cette estimation financière prend en compte les trois postes de dépense suivants :
- les ressources humaines
- les frais de fonctionnement
- les investissements propres au projet

E.1 Ressources humaines

La rémunération du stage était une indemnité. Celle-ci n'est pas soumise à des charges patronales. Le salaire, charges patronales comprises, pour de l'encadrant est d'environ 3 500 €e tle temps cumulé d'encadrement est estimé à un mois temps complet.

Intitulé	Coût unitaire	Nombre de mois	Coût total
Indemnité de stage	307,65 €/mois	6	1 846 €
Encadrement tuteur	3 500 €/mois	1	3 500 €
Total des dépenses salariales et d'indemnité			5 346 €

E.2 Frais de fonctionnement

Le coût environnent est de 26,70 €/h/personne. Il comprend les locaux, l'électricité, la prestation pour le ménage, le téléphone, l'accés Internet, les abonnements de la bibliothéque, les frais administratifs... La base de travail est de 151,6 h/mois. Ceci s'applique au 6 mois de stage mais également au mois d'encadrement passé par le tuteur.

Il n'y a pas eu de frais de déplacement dans le cadre de ce stage.

Poste	Coût unitaire	Quantité	Coût total
Coût environnent	26,70 €/h/personne	151,6 × 7	28 334 €
Subvention repas	5,89 €/j/personne	22 × 6	777 €
Total des frais de fonctionnement			29 111 €

E.3 Investissements

Aucun investissement n'a été nécessaire pour ce projet. Le projet Pop Art disposais en effet de tous les logiciels et de tout le matériel requis.

Il n'y a pas eu non plus de prestation extérieure facturée. L'aide, apportée par Matthieu Moy et ses stagiaires, a été réalisée à titre gracieux.

E.4 Synthèse

Après cumul, le budget obtenu pour ce projet est d'environ 34 460 €. Il est intéressant de comparer ce coût avec celui d'un projet équivalent effectué en dehors du cadre d'un stage. Le coût aurait alors été d'environ 53 000 €soit 54 % de plus.

Poste	**Coût**
Coût salarial	5 346 €
Frais de fonctionnement	29 111 €
Investissements	0 €
Total des dépenses	34 457 €

RAPPORT DE PROJET DE FIN D'ÉTUDES
ESISAR 2003/2004

Mots clés :
SystemC, système en couche à base de composants, systèmes à transitions avec contraintes, vérification formelle, compilation, C et C++

Résumé :
Ce rapport présente une méthodologie permettant de traduire la description en SystemC d'un système sur puce en un système en couches à base de composants utilisant des systèmes à transitions avec contraintes. Cette traduction est formalisée autant que possible et doit être encore étendue. Seules les structures les plus simples et les plus courantes sont actuellement gérées.

L'aspect modulaire permet une réutilisation accrue des composants tant en SystemC qu'en Prometheus, outils implémentant la vérification compositionnelle des systèmes à transitions avec contraintes. La structure du compilateur respecte la structure modulaire et est considérée comme aisément extensible pour gérer les structures les plus complexes de SystemC et C++.

Key words :
SystemC, layered component-based system, transitions system with constraints, formal checking, compilation, C and C++

Abstract :
This report describes a methodology allowing to translate a SystemC description of a system on chip (SoC) in a layered component-based system consisting of transitions systems with constraints. This translation is formalized as much as possible but must be extended yet. Only simple and frequently used constructs are currently taken into account.

The compositionnal aspect allows to reuse components in SystemC but also in Prometheus, tool implementing the component-based verification. The compiler framework respect the modular and component-based framework and is regarded as easily extensible to manage the more complex constructs of SystemC and C++.