CentraleSupélec

École doctorale STITS

« Sciences et Technologies de l'Information des Télécommunications et des Systèmes »

THÈSE DE DOCTORAT (PhD THESIS)

Domaine : STIC
Spécialité : Informatique

Laboratoire de Recherche en Informatique
Chaire Conception Analogique Avancée Thales/Supélec

Soutenue le (defended on) 10 Juillet 2015

par (by) Daniel Chaves Café

Multi-level modeling for verification and synthesis of complex systems in a multi-physics context

Composition du jury

Rapporteurs :	Yamine AIT AMEUR	Professeur à l'ISAE - ENSMA
	Carlos VALDERRAMA SAKUYAMA	Professeur à l'Université de Mons
Examinateurs :	François PECHEUX	Professeur à l'UPMC/LIP6
	Nicolas SABOURET	Professeur à l'Université Paris Sud/LIMSI
Invité :	Antoine RAUZY	Professeur à CentraleSupélec/Chaire Blériot-Fabre
Directeur de thèse :	Frédéric BOULANGER	Professeur à CentraleSupélec/LRI
Co-encadrant :	Filipe VINCI dos SANTOS	Professeur à CentraleSupélec/Chaire Thales

"Só o sofrimento constroi"

Eli

CentraleSupélec

Abstract

École doctorale STITS
Department of Computer Science

Doctor of Philosophy

Multi-level modeling for verification and synthesis of complex systems in a multi-physics context

by Daniel Café

In the era of highly integrated electronics systems, engineers face the challenge of designing and testing multi-faceted systems with single-domain tools. This is difficult and error-prone. These so called heterogeneous systems have their operation and specifications expressed by several formalisms, each one particular to specific domains or engineering fields (software, digital hardware, analog, etc.). Existing design tools are meant to deal with homogeneous designs using one formalism at a time. In the current state, industry is forced to battle with integration issues at every design step, i.e. specification, simulation, validation and deployment. Common divide-to-conquer approaches do not include cross-domain interface specification from the beginning of the project. This lack is often the cause of issues and rework while trying to connect parts of the system that were not designed with the same formalism. This thesis proposes an approach to deal with heterogeneity by embracing it from the beginning of the project using SysML as the unifying tool. Our proposal hinges on the assignement of well-defined semantics to SysML diagrams, together with semantic adaptation elements. To demonstrate the effectiveness of this concept, a toolchain is built and used to generate systems simulation executable code automatically from SysML specifications for different target languages using model driven engineering techniques.

Keywords: SysML, SystemC-AMS, VHDL-AMS, Semantic Adaptation, Model Transformation.

Acknowledgments

This thesis was a joint collaboration between the Computer Science Department of Supélec and the Thales/Supélec Chair on Advanced Analog System Design under the scientific leadership of Professor Frédéric Boulanger and Professor Filipe Vinci dos Santos.

I would like to thank both Prof. Frédéric Boulanger and Prof. Filipe Vinci for giving me the opportunity to carry out this research and for their valuable advice and support. I would also like to express my gratitude to Christophe Jacquet and Cécile Hardebolle for all of their great research ideas, productive discussions and encouragement that empowered me to pursue the experiments that resulted in this thesis. I am grateful to all of my advisors for helping me to present the results of this work in an understandable way.

This work wouldn't have been possible without the support of many. I would like to thank my dearest friends and colleagues Alexis, Andreea, Ashish, Ana, Rémi, Mickael, Emanuel, Ionela, Iuliana, Gisela, Pierre, Konstantinos, Coriane, Georges, Dimitri, Raul, Luc, Hugo, Nicolas, Safouan, Marc-Antoine, Gianluca, Arpad for their kindness and joyfulness, which made our work environment so pleasurable and took away so much burden off my shoulders.

One special thank to my beloved family, my mother Ângela, my father Fernando and my sister Luísa for their support, encouragement and understanding throughout my dissertation. Last but not least, I would like to thank Letícia Magalhães Costa for her unconditional love and support. Her constant words of encouragement were of inestimable value giving me the strength to carry on.

Contents

List of Figures

Abbreviations

CT	**C**ontinuous **T**ime
DE	**D**iscrete **E**vent
MoC	**M**odel of **C**omputation
SDF	**S**ynchronous **D**ata **F**low
FSM	**F**inite **S**tate **M**achine
TDF	**T**imed **D**ata **F**low
LSF	**L**inear **S**ignal **F**low
ELN	**E**lectrical **L**inear **N**etworks
CSP	**C**ommunication **S**equential **P**rocesses
OMG	**O**bject **M**anagement **G**roup
XMI	**XM**L **I**nterchange
MDE	**M**odel **D**riven **E**ngineering
MDA	**M**odel **D**riven **A**rchitecture
PIM	**P**latform **I**ndependent **M**odel
PSM	**P**latform **S**pecific **M**odel
ATL	**A**tlas **T**ransformation **L**anguage
TLM	**T**ransaction **L**evel **M**odeling
AMS	**A**nalog **M**ixed **S**ignal

To my family

Chapitre 1

Résumé en Français

Cette thèse s'inscrit dans une collaboration entre le Département Informatique de Supélec et la Chaire Thales/Supélec des Systèmes Analogiques Avancés. Ce travail vise à résoudre la problématique des nouvelles méthodologies de conception de circuits intégrés qui deviennent de plus en plus hétérogènes. Cette hétérogénéité provient de l'intégration incessante de nouvelles fonctionnalités et/ou de composants de différentes natures (analogique, numérique ou même mécanique) dans un même circuit intégré. Dans notre équipe, nous élaborons des approches permettant de modéliser chaque composant avec le formalisme qui lui convient le mieux. Mais l'utilisation de plusieurs formalismes pour différentes parties d'un même système pose un problème d'adaptation entre les différentes interfaces. Nous essayons de résoudre ce problème avec le concept d'adaptation sémantique, plus particulièrement appliqué au langage graphique de modélisation de systèmes SysML.

Dans le contexte de la modélisation de systèmes, SysML apparaît comme un langage pivot de spécification et de documentation. Ses diagrammes permettent la définition de la structure et du comportement de systèmes. La flexibilité de SysML a pour inconvénient qu'il n'existe pas de méthode standard pour définir leur sémantique. Ce problème est flagrant dans la conception de systèmes hétérogènes, où différentes sémantiques opérationnelles peuvent être utilisées. Dans cette thèse nous présentons une manière de donner une sémantique opérationnelle aux éléments de SysML sous la forme de transformations vers des langages textuels et exécutables, tels que SystemC-AMS et VHDL-AMS, permettant ainsi la validation par simulation de modèles SysML.

SystemC et son extension de modélisation de systèmes analogiques et mixtes, SystemC-AMS, font partie de l'ensemble d'outils indispensables de l'industrie pour la modélisation et la simulation des systèmes numériques et mixtes. Dans le domaine numérique, SystemC permet une modélisation de plus haut niveau que VHDL et Verilog. Le style de codage

au niveau transactionnel (TLM), par exemple, permet de remplacer les interconnexions numériques au niveau des registres (RTL) des bus de données, par quelques appels de fonctions suivant un protocole donné. Cela diminue considérablement le temps de simulation de ce type de système puisque le noyau de simulation n'a plus besoin d'itérer autant de fois. Cela provient du fait qu'il y a moins de composants à prendre en compte dans la simulation. Ainsi, SystemC, permet le développent conjoint de matériel et de logiciel en s'appuyant sur des spécifications exécutables. N'ayant plus besoin d'attendre que le circuit soit prêt et opérationnel pour que le développement logiciel démarre, l'industrie utilise SystemC pour réduire les délais de commercialisation de ses produits. La partie AMS de ce langage a permis d'aller au delà des systèmes numériques en combinant des processeurs entiers avec des systèmes de transmission de données en radio fréquence, ainsi que des composants mixtes comme des convertisseurs analogiques-numériques.

VHDL, de son coté, est devenu incontournable. Sa large adoption par l'industrie prouve son importance. Avec Verilog, VHDL fait partie des outils les plus répandus de l'industrie pour la modélisation, la simulation et principalement pour la synthèse automatique des systèmes numériques. Son comportement déterministe a donné de la confiance aux ingénieurs qui l'utilisent. L'extension AMS permet la modélisation des systèmes complexes et hétérogènes en s'appuyant sur un solveur d'équations différentielles.

Les contributions de cette thèse peuvent être séparées en trois parties. Dans la première, un méta-modèle pour le langage SystemC-AMS est présenté en y incluant les éléments nécessaires aux différents types d'adaptation, ce qui n'existait pas auparavant. Avec ce méta-modèle, une approche de transformation de SysML vers SystemC-AMS est présentée permettant la génération automatique d'un code SystemC-AMS exécutable. Le code généré a pour but principal de vérifier le comportement d'un système par simulation. Ce travail a été publié dans un article pour la conférence FDL 2013 [14]. Le deuxième travail consiste en une amélioration de l'approche grâce aux retours de la communauté. Nous avons implémenté une nouvelle version de la théorie d'adaptation sémantique appliquée à une transformation de modèles partant de SysML vers VHDL-AMS. Ce travail a été publié dans le workshop de modélisation multi-paradigme MPM 2014 [15]. La forte ressemblance entre ces deux résultats nous a motivés à poursuivre une généralisation de l'approche en y séparant la syntaxe de la sémantique quelque soit le langage de simulation. Cela nécessite une représentation intermédiaire des modèles SysML ainsi que des transformations de modèles dédiées à la traduction de syntaxe et à l'interprétation de la sémantique des modèles SysML.

1.1 Introduction

Un système hétérogène est constitué de composants de différentes natures, qui sont modélisés selon des formalismes distincts. Par exemple, un accéléromètre MEMS a une partie mécanique, une partie analogique et une interface numérique. Dans le domaine numérique, le formalisme à événements discrets est efficace pour la simulation grâce à l'abstraction des phénomènes analogiques qui font qu'une bascule change d'état. Dans le domaine analogique, la modélisation par réseaux de composants électriques permet de décrire la topologie du réseau pour en déduire les équations différentielles. La simulation des systèmes hétérogènes permet de garantir une fabrication correcte dès le premier essai. Le métier d'architecte de systèmes consiste à intégrer différents paradigmes dans un même modèle, ce qui pose des problèmes d'adaptation et fait appel à des compétences pluridisciplinaires et à la maîtrise des outils de simulation.

1.1.1 État de l'art

Les outils de modélisation hétérogène sont encore en phase d'expérimentation et de maturation. Ptolemy II [20] gère l'hétérogénéité par hiérarchie. Chaque composant est considéré comme une boîte noire, et la sémantique d'exécution et de communication est définie par une entité appelée *director*. Cette entité définit le modèle de calcul (MoC) de chaque composant. Pour arriver à cet objectif, Ptolemy II définit un moteur d'exécution générique [42] avec trois phases distinctes : l'initialisation, l'itération (pre-fire, fire et post-fire) et la finalisation (wrapup). Chaque *director* compose le comportement des blocs en redéfinissant ces phases d'exécution.

Inspiré de Ptolemy II, ModHel'X [10, 32] a été créé dans le but de modéliser explicitement l'adaptation sémantique en rajoutant au moteur d'exécution générique de Ptolemy des phases d'adaptation sémantique. Cela donne un moyen efficace de définir la sémantique des interactions entre différents modèles de calcul. Néanmoins, l'implémentation actuelle de ModHel'X est basée sur un méta-modèle non-standard qui rend difficile l'intégration dans les chaînes d'outils existantes. Dans cet thèse, nous présentons une méthode pour définir la sémantique opérationnelle ainsi que l'adaptation sémantique pour des modèles hétérogènes SysML par traduction en SystemC-AMS et VHDL-AMS.

SysML est un langage graphique de spécification de systèmes dont les diagrammes facilitent la communication entre différentes équipes d'un projet pluridisciplinaire. La sémantique opérationnelle de ces diagrammes n'est toutefois pas précisément définie, ce qui est un obstacle à l'exécution de modèles SysML. L'implémentation de cette sémantique peut être réalisée dans un langage capable de simuler des modèles hétérogènes. Comme

une première preuve de concept, nous avons choisi d'utiliser SystemC-AMS [28] pour la modélisation des systèmes mixtes. SystemC-AMS est une bibliothèque C++ contenant un noyau de simulation à événements discrets ainsi qu'un ensemble de blocs de base. Ce langage comprend trois modèles de calcul : Dicrete Event (DE), Timed DataFlow (TDF) et Continuous Time (CT). j

Notre approche consiste à générer du code SystemC-AMS à partir d'un modèle SysML annoté avec des éléments qui donnent la sémantique d'exécution de chaque bloc SysML et la sémantique d'adaptation entre blocs. Nous utilisons des techniques de l'ingénierie dirigée par des modèles (IDM) pour réaliser les transformations de modèles et la génération de code.

1.1.2 Génération de code SystemC à partir de SysML

La génération de code SystemC à partir des diagrammes SysML a été le sujet d'étude de plusieurs travaux de recherche. Raslan et al. [57] ont définit une correspondance entre SysML et SystemC ciblant seulement le formalisme à événements discrets. Prevostini et al. [56] ont proposé un profil de SysML pour la modélisation des circuits intégrés et la génération automatique de code vers SystemC dans le but de réaliser des co-simulations entre matériel et logiciel embarqué. Mischkalla et al. [47] ont travaillé avec un sous-ensemble de SystemC pour créer un outil de synthèse de matériel a partir de SysML.

Toutes ces approches ont abordé le sujet de l'intégration de SysML avec le simulateur à événements discrets de SystemC ou un sous-ensemble synthétisable de SystemC seulement pour les systèmes homogènes. Ils n'ont pas considéré l'aspect multi-domaine des systèmes hétérogènes. Dans cette thèse, nous souhaitons résoudre le problème de la modélisation des systèmes hétérogènes en utilisant des diagrammes SysML avec un ensemble de définitions sémantiques pour chaque domaine ainsi que des mécanismes d'adaptation sémantique dans les interfaces hétérogènes. Il ne s'agit pas seulement d'améliorer la compréhensibilité des projets de systèmes hétérogènes avec des descriptions graphiques de haut niveau, mais aussi fournir une sémantique exécutable aux diagrammes SysML, ce qui nous permettra de réaliser des simulations des modèles à partir de leurs spécifications décrites en SysML.

1.1.3 Génération de code VHDL et VHDL-AMS à partir de SysML

En ce qui concerne VHDL et VHDL-AMS, des travaux considérables ont été réalisé dans le but de créer des outils de génération automatique de code à partir des descriptions

SysML. Nous citons D. Guihal [30] et J. Verriers [64] qui ont étudié et étendu le méta-modèle de VHDL proposé dans [2] et [59] avec des concepts de la partie AMS. Pour la génération de code, ils se sont basés principalement sur des diagrammes BDD (Block Definition Diagram) et IBD (Internal Block Diagram) de SysML. Ils ont utilisé un élément particulier de SysML, les blocs de contrainte, pour définir les équations physiques des modules analogiques.

De manière similaire aux travaux ciblant SystemC, ces travaux n'ont pas traité les incohérences sémantiques introduites par l'hétérogénéité. Nous présenterons, en détails dans le chapitre 4 et résumé ici, une technique pour résoudre ce problème. Nous montrerons comment utiliser les bonnes pratiques de ModHel'X, consistant à définir explicitement une sémantique d'adaptation entre composants de différente nature, pour résoudre ces conflits sémantiques. Dans notre approche, nous utilisons SysML comme un langage pivot pour la génération de code vers SystemC-AMS et VHDL-AMS.

1.2 Notre approche

Notre approche consiste à réaliser deux transformations de modèles. En partant d'un modèle SysML, nous appliquons une transformation "model-to-model" (M2M) écrite en ATL (Atlas Transformation Language)[40] pour générer une représentation intermédiaire du système dans le langage cible. La génération de code se fait juste après, en appliquant une deuxième transformation de type "model-to-text" (M2T) écrite en ACCELEO[49].

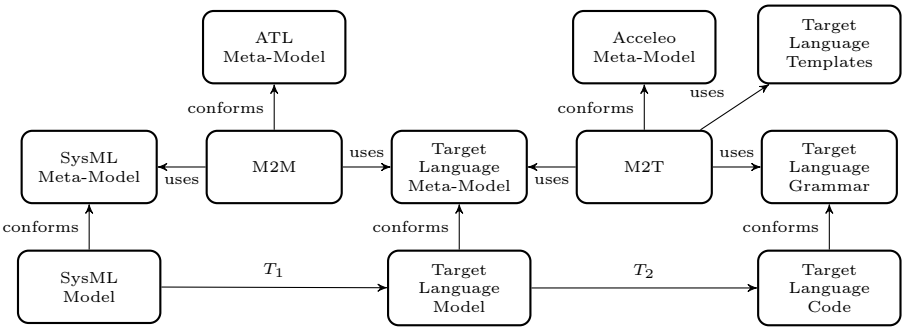

FIGURE 1.1 – Notre approche

Nous avons fait deux expérimentations. Une avec SystemC-AMS et une deuxième avec VHDL-AMS. Dans la première expérimentation, nous utilisons des contraintes SysML pour indiquer le modèle de calcul utilisé par un bloc donné. Les mots clés "CT Block", "DE Block" et "FSM Block" sont utilisés pour spécifier l'utilisation des MoCs Continuous Time, Discrete Event et Finite State Machine, respectivement. La sémantique de ces MoCs est décrite dans [42]. Dans la deuxième expérimentation nous avons fait évoluer cette technique dans un profil de SysML. Cela correspond à une technique courante dans

l'ingénierie dirigée par les modèles qui propose l'extension des éléments d'un langage donné (tel que SysML) avec des notions propres au profil. Notre profil de SysML en particulier introduit des stéréotypes qui peuvent être appliqués aux blocs SysML. Cela remplace la contrainte SysML utilisée dans le premier essai et augmente la sémantique avec des notions spécifiques à chaque MoC. Dans le MoC SDF, par exemple, nous pouvons définir le taux d'activation d'un module ainsi que le nombre d'échantillons nécessaires en entrée pour déclencher le calcul ou le nombre d'échantillons produits en sortie.

Nous appliquons ces MoCs aux diagrammes SysML qui nous semblent leur correspondre. Un automate par exemple, est modélisé par un diagramme d'états-transitions, un modèle à temps continu peut être représenté par un diagramme d'interconnexions de blocs où chaque bloc représente une fonction. D'autres solutions peuvent être imaginées, comme l'utilisation des diagrammes paramétriques pour la définition des équations différentielles.

Nous considérons dans ce travail l'adaptation sémantique entre les différents domaines, par exemple l'échantillonnage entre un sous-système à temps continu et un sous-système à temps discret, que nous exprimons dans des commentaires liés aux ports des modules. Ces commentaires, comme nous verrons dans la prochaine section et dans la figure 1.2, nous permettront de choisir des adaptateurs pré-existants dans les langages cibles (i.e. SystemC-AMS et VHDL-AMS) ou bien de définir des adaptateurs non-standard s'il le faut.

1.3 Application à la génération de code SystemC-AMS

Ce premier travail vise l'utilisation de deux normes industrielles de spécification, modélisation et simulation des systèmes hétérogènes : SysML et SystemC-AMS. SysML fournit un moyen graphique pour modéliser la structure et le comportement de systèmes hétérogènes. Malgré sa flexibilité, SysML manque d'une sémantique pour donner aux éléments de ce langage un sens précis. Les implémentations actuelles de la norme permettent plusieurs interprétations des éléments syntaxiques et peuvent causer des malentendus lors du portage d'un modèle parmi des différents outils. Nous abordons ce problème en ajoutant une sémantique concrète aux diagrammes SysML par l'utilisation de modèles de calcul. Nous illustrons notre approche avec un système composé de deux composants mieux modélisés avec deux MoCs distincts, à savoir FSM et CT, tout les deux disponibles dans un environnement de simulation DE. Nous définissons également des règles explicites d'adaptation sémantique pour les interactions entre ces MoCs et nous générons automatiquement du code SystemC-AMS. Cette génération de code est basée sur les transformations de modèles et sur nos définitions sémantiques. Nous profitons ainsi

de SysML comme outil de modélisation et nous bénéficions en même temps du puissant moteur de simulation que SystemC offre pour valider les systèmes hétérogènes par simulation.

1.3.1 Cas d'étude

Pour illustrer l'approche, prenons l'exemple d'un véhicule à vitesse contrôlée, présenté en détail dans [14] et résumé ici. Les diagrammes de la figure 1.2 montrent comment appliquer une sémantique à un bloc SysML en lui ajoutant des annotations textuelles. Nous nous intéressons à une modélisation haut niveau de la dynamique du système en utilisant des équations de la physique classique. Nous déduisons l'accélération de $F = m \times a$, puis la vitesse et le déplacement en intégrant l'accélération. La dynamique du système est modélisée par le formalisme CT. Nous définissons une équation différentielle dans un diagramme IBD en assemblant des fonctions de base, comme l'intégrateur et le gain (voir figure 1.3 à gauche). Le contrôle est modélisé par un diagramme états-transitions où la rétroaction de la force dépend de l'état du contrôleur (Accelerate, Hold et Brake). Ces deux formalismes sont non seulement exprimés de manières différentes mais ont aussi des sémantiques opérationnelles différentes.

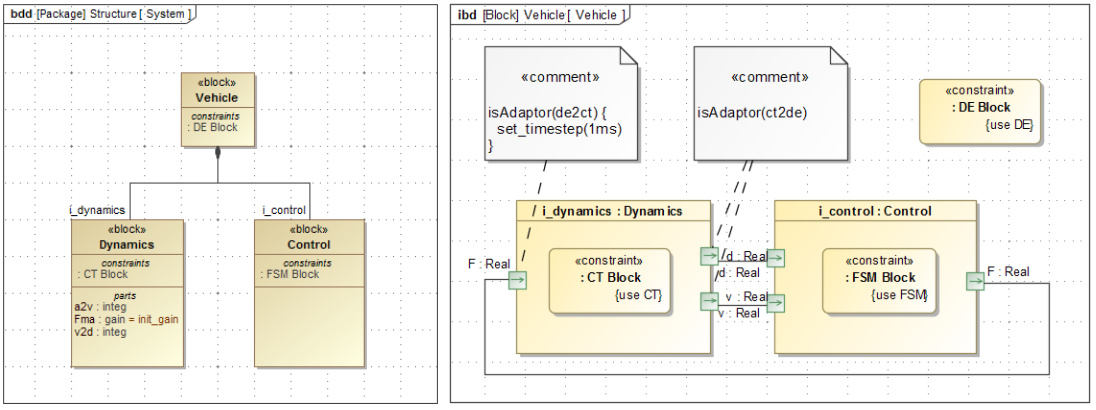

FIGURE 1.2 – Exemple du véhicule à vitesse contrôlée : modèles SysML annotés

L'adaptation entre différents domaines est décrite par des commentaires liés aux ports du bloc *i_ dynamics* dans le diagramme de droite. Une fois l'interface annotée par le mot clé *isAdaptor*, nous lui appliquons une sémantique d'adaptation spécifique. Dans le cas d'un adaptateur *de2ct* le dernier événement capturé par le port est enregistré et sa valeur répétée à un pas d'échantillonnage fixe, donné par la directive *set_ timestep*. En sortie, l'adaptation *ct2de* génère un événement à chaque changement de la valeur produite en sortie.

Cet approche est cependant très dépendante du choix du langage cible (ici SystemC-AMS), ainsi que des modèles de calcul utilisables. Nous ne pouvons implémenter que les

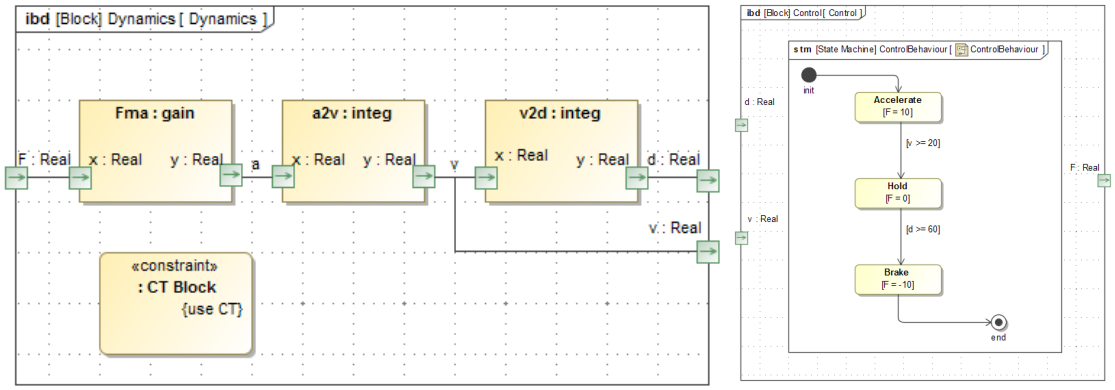

FIGURE 1.3 – Diagrame interne du bloc *i_ dynamics* et du bloc *i_ control*

MoCs que SystemC-AMS est capable d'exécuter. Cette limitation n'est pas contraignante car SystemC possède un moteur d'exécution à événements discrets qui supporte une large gamme de modèles de calcul discrets, et la partie AMS supporte les modèles continus. Il est important de souligner que la sémantique des MoCs est prise en compte de deux manières : soit par utilisation directe de la bibliothèque de base de SystemC, soit par implémentation dans la transformation de modèles. Dans l'exemple, la sémantique *block dynamics* s'appuie sur l'assemblage de blocs de la bibliothèque LSF (Linear Signal Flow) de SystemC-AMS. Dans le contrôleur, la sémantique de la machine à états est codée par la transformation M2M car SystemC n'a pas de MoC FSM. Cela a une influence sur les performances de simulation comme discuté dans [54].

La même réflexion s'applique aux mécanismes d'adaptation. L'échantillonnage des données et la production d'événements discrets sont réalisés par des adaptateurs spécifiques de la bibliothèque de SystemC-AMS. L'adaptation qui est faite à l'entrée du bloc *dynamics* est traduite en un module SystemC capable de transformer un événement discret de la machine à états en échantillons au pas fixe défini par la commande *set_ timestep(1ms)*. L'adaptateur de sortie détecte les changements de valeur pour produire des événements. Une autre adaptation pourrait ne générer que les événements qui correspondent à une transition de l'automate. Cela permettrait une simulation plus performante mais rendrait l'adaptateur dépendant des modules qui lui sont connectés. Les adaptateurs possibles sont également limités par le langage cible. Dans notre exemple, nous n'avons utilisé que les adaptateurs de la bibliothèque SystemC-AMS. La conception d'adaptateurs spécialisés peut être réalisée sous forme de modules dédiés, comme discuté dans [21] avec le concept de *thick adapter*.

1.3.2 Résultats

La simulation de ce système est obtenue par transformation du modèle SysML en code SystemC-AMS. Le code généré par notre outil est ensuite compilé et exécuté. Les résultats de simulation sont affichées dans la figure 1.4

Le bloc qui modélise la dynamique du système a été annoté avec une contrainte “CT Block”, ce qui nous a permit d’interpréter le diagramme interne de la figure 1.3 dans un module LSF de SystemC-AMS. Nous avons utilisé la première transformation de modèles de notre approche (illustrée dans la figure 1.1) pour créer un mapping entre les éléments de SysML et leurs contreparties du côté SystemC-AMS. Ici, l’interprétation des équations différentielles de chaque sous-bloc du modèle de la dynamique a permis l’instantiation des modules LSF du domaine continu de SystemC-AMS, i.e. *sca_ lsf::sca_ integ* pour l’intégrateur et *sca_ lsf::sca_ gain* pour le module de gain.

La machine à états finis a été traduite dans un module SystemC pur à événements discrets. Ce module possède la structure de base d’une machine à états à deux processus, un pour la logique de transition d’état et un autre qui modélise le registre d’état. Cette structure est aussi présente dans les machines à état en VHDL comme dans [62].

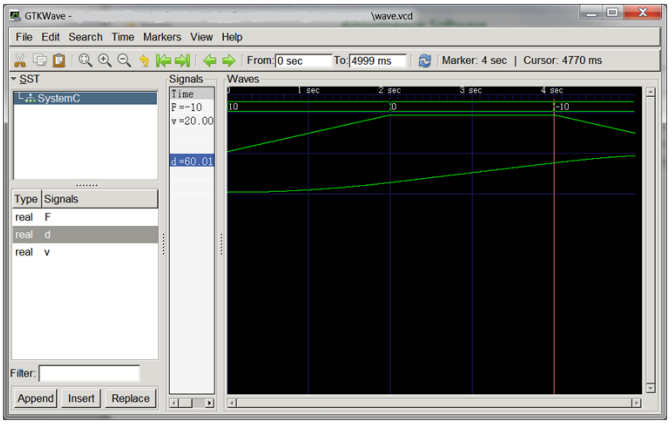

FIGURE 1.4 – Résultats obtenus par génération automatique de code

Le résultat de la figure 1.4 est produit à partir de l’exécution du code compilé et généré à partir de notre outil. Nous y voyons la force appliquée au modèle de la dynamique dans la première ligne. Il s’agit d’un signal de nature discrète adapté correctement pour générer un stimulus dans un domaine continu. Le véhicule accélère et atteint la vitesse de 20m/s comme spécifié par la machine à états du bloc de contrôle de la figure 1.3. Ensuite le contrôle maintient la vitesse jusqu’à ce que la distance parcourue atteigne 60m légèrement après les 4 secondes de simulation. Il termine par un freinage jusqu’à la fin de la simulation.

L'aspect remarquable de cette simulation est que le code SystemC-AMS a été intégralement généré à partir des diagrammes SysML, sans intervention humaine. La sémantique a été définie individuellement pour chaque bloc en les annotant avec des mots clés qui représentent des différents modèles de calcul. Les interactions dans les frontières multi-domaines ont été décrites par des adaptateurs.

1.3.3 Discussion

Ce premier essai a été illustré sur un modèle à trois formalismes (CT, DE et FSM). Nous avons spécifié la sémantique de chaque bloc SysML avec des annotations textuelles et la sémantique d'adaptation dans les commentaires liés aux ports. Le code exécutable est généré par transformation de modèles en utilisant ATL et ACCELEO.

Cette technique est un premier pas vers un framework générique de génération de code pour d'autres langages, par exemple VHDL-AMS. Notre approche en deux étapes (M2M et M2T) permet d'envisager l'extensibilité à d'autres MoCs par ajout de règles de transformation M2M correspondant au MoC désiré, sans devoir changer le générateur de code (partie M2T). Notre objectif est de découpler au maximum l'aspect sémantique des MoCs et l'aspect génération de code afin de supporter plus facilement de nouveaux MoCs et de nouveaux langages cibles. En ce qui concerne la définition des MoCs et l'adaptation sémantique, nous suivons les travaux en cours sur ce sujet au sein de l'initiative GeMoC [23].

1.3.4 Réflexions sur l'approche

Depuis la publication de ce travail exploratoire, nous avons amélioré l'intégration de notre approche avec des techniques de l'IDM. Nous utilisons désormais des stéréotypes pour définir les MoCs au lieu des mots-clés dans les contraintes SysML. Ces contraintes sont désormais réservées à la spécification des équations différentielles dans le formalisme CT. Les annotations des adaptateurs ont aussi évolué vers un langage dédié à la modélisation de leur comportement, ce qui évite de polluer le modèle avec du code spécifique au langage cible. Nous allons présenter ensuite ces améliorations.

1.4 Application à la génération de code VHDL-AMS

Pour cette deuxième itération, nous allons appliquer la même suite de transformations de modèles comme illustré dans la figure 1.1, mais cette fois-ci au langage VHDL-AMS. Comme cas d'étude, nous allons présenter un accéléromètre MEMS. Cet exemple nous

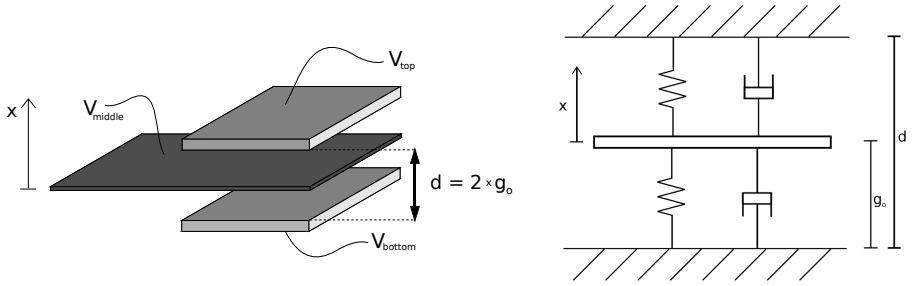

FIGURE 1.5 – Modèles électrique et mécanique d'un MEMS

paraît intéressant car il couvre une partie purement analogique/mécanique et une partie numérique. Les défis de modélisation et simulation de ce type de système se trouvent dans les intersections entre les dispositifs de différentes natures. Nous verrons ici, comment réaliser des simulations mixtes à partir d'un code généré automatiquement provenant d'une spécification écrite en SysML.

Les dispositifs MEMS (Micro Electro Mechanical Systems) sont un bon exemple d'un système hétérogène qui mélange des composants mécaniques, analogiques et numériques dans le même système. Ce type de système offre des caractéristiques importantes pour la mesure de quantités physiques tels que l'accélération, pression, force ou même des concentrations chimiques. Les capteurs MEMS sont basés fortement dans certains mécanismes de transduction comme les dispositifs piézo-résistifs ou capacitifs. Notre cas d'étude se limite à une simplification d'un capteur capacitif avec deux électrodes et une membrane capable de se déplacer dans l'axe vertical, comme illustré dans la figure 1.5. À droite se trouve un modèle mécanique équivalent.

La structure de la figure 1.5 à gauche forme deux capacités entre la membrane et les deux électrodes. Le mouvement vertical de cette membrane provoque une variation dans les deux capacités, sachant que $C \propto 1/(g_0 \pm x)$, ou g_0 est la distance entre la membrane au repos et une électrode, et x est le déplacement par rapport au repos. Nous fixons le courant à zéro et appliquons une tension symétrique sur les deux électrodes (i.e. $V_{top} = -V_{bottom} = V_0$) pour obtenir la relation linéaire 1.1 entre la tension de la membrane et son déplacement. Nous présentons plus de détails du développent mathématique dans la section 4.6.2

$$V_{middle} = V_0 \frac{x}{g_0} \tag{1.1}$$

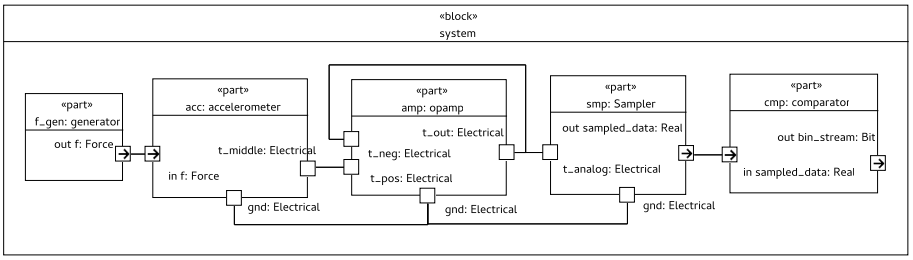

FIGURE 1.6 – Modèle SysML [IBD]

1.4.1 Le Modèle SysML

Nous avons reparti la modélisation de ce système en cinq blocs majeurs comme nous voyons dans les figures 1.6 et 1.7 :

- le bloc **accelerometer** modélise la dynamique électromécanique. Dans ce modèle purement analogique, nous modélisons les équations physiques des capacités variables avec le déplacement de la membrane ainsi que les équations du modèle mécanique en considérant seulement la résistance du ressort et la friction.
- le bloc **opamp** modélise un amplificateur idéal suivant l'équation du gain $V_{out} = gain * V_{in}$. Nous considérons aussi la saturation de cet amplificateur dans une équation par morceaux définit dans une contrainte SysML. Cette équation est affichée dans la figure 1.8.
- le bloc **sampler** modélise l'interface entre le monde analogique et le monde numérique. Il est responsable de la conversion d'une tension analogique en mots binaires. Nous choisissons ici une adaptation de type périodique avec une fréquence d'échantillonnage de 2 micro secondes.
- le bloc **comparator** est un bloc purement numérique qui surveille la sortie de l'échantillonneur. Il génère ainsi un signal en sortie de type flot de bits qui est ultérieurement soumis à un traitement dans le domaine numérique. Les détails du reste du système restent en dehors du cadre de cette discussion.
- enfin, le bloc **source** génère une force d'entrée sinusoïdale qui stimule le modèle. Nous souhaitons ici vérifier si la sortie électrique suit de manière linéaire la force d'entrée.

Nous avons utilisé un profil de SysML pour donner une sémantique à chacun des blocs définis précédemment. Dans cet exemple, l'utilisation des stéréotypes ≪analog≫, ≪digital≫ et ≪adaptor≫ nous permet d'interpréter de manière automatique les expressions et contraintes SysML de chaque bloc. Le signe "=" du bloc **accelerometer** est interprété

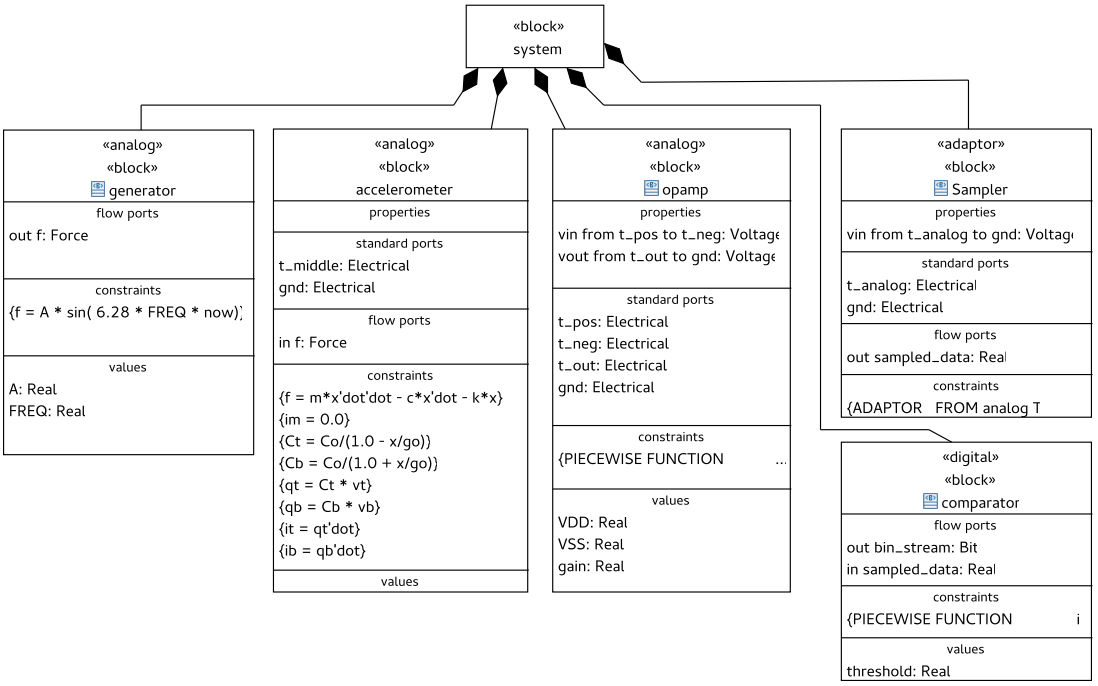

FIGURE 1.7 – Modèle SysML [BDD]

comme une égalité dans le domaine continu. Le même symbole est utilisé dans le domaine numérique pour définir une attribution d'un signal discret. Cette technique élimine l'ambiguïté des éléments SysML permettant ainsi la génération automatique de code par interprétation de ces éléments.

Le bloc d'adaptation, annoté avec le stéréotype ≪adaptor≫ possède une contrainte SysML distincte. La contrainte qui démarre par le mot-clé "ADAPTOR" définit la sémantique d'adaptation entre le monde analogique et le monde numérique. Cette contrainte, affichée intégralement dans la figure 1.8 à droite, montre comment déclarer un adaptateur avec notre mini-DSL de définition d'adaptation. Dans ce cas, un échantillonneur est défini par une liste de paramètres d'adaptation, une entrée, une sortie et une période d'échantillonnage périodique. La liste complète de paramètres d'adaptation se trouve dans le chapitre 4 section 4.7.3. Ici, nous utilisons cet exemple simple pour montrer la technique de transformation d'un modèle SysML en code VHDL-AMS par transformation de modèles.

```
1 PIECEWISE FUNCTION
2   V_in < VSS/gain:
3      V_out = VSS,
4   V_in > VDD/gain:
5      V_out = VDD,
6   elsewhere:
7      V_out = gain * V_in
```

```
1 ADAPTOR
2   FROM analog TO digital
3   IS sampler
4   PARAMS
5     input     : vin,
6     output    : sampled_data,
7     timestep  : 2us
```

FIGURE 1.8 – Fonction définie par morceaux & langage d'instanciation de l'adaptateur

La spécification de l'adaptateur de la figure 1.8 garantit que la sortie de données *sampled_ data* sera échantillonnée à un pas fixe de 2 μs. Ce cas d'adaptation est intéressant puisque nous adaptons non seulement la base de temps mais aussi le type de donnée. D'un côté, *vin* est un nœud de sortie d'un circuit analogique. L'adaptateur doit extraire la valeur de la tension entre ce nœud et la référence et la convertir pour un bloc du domaine à événements discrets. Les paramètres *input* et *output* indiquent la tension analogique d'entrée et le signal discret de sortie.

Certains adaptateurs que nous utilisons ont leur contrepartie dans la bibliothèque de base du langage cible, d'autres ne l'ont pas. Dans le premier cas, notre transformation choisi l'adaptateur correspondant de la bibliothèque. Dans le deuxième cas, nous définitions un nouveau module qui se comporte comme défini par les paramètres d'adaptation.

Dans ce cas d'étude, l'échantillonneur n'est pas présent dans la bibliothèque de base de VHDL-AMS. Nous générons ainsi un module responsable de cette adaptation. Le code qui a été généré peut être séparé en deux processus. Un qui définit le pas d'échantillonnage (un générateur d'horloge) et un deuxième qui modélise la sémantique d'adaptation. Ce deuxième est activé par le premier et copie la valeur des tensions dans le terminal d'entrée dans un signal numérique de sortie. Le deuxième s'active toutes les 2 μs, comme défini par notre spécification.

La sortie de l'échantillonneur est connectée à un comparateur qui génère un flux de bits à partir de son entrée. Lorsque la tension d'entrée analogique franchit une valeur donnée par le paramètre *threshold*, la sortie bascule à '1' ou '0' sinon.

1.4.2 Résultats de simulation

En appliquant les deux transformations de notre approche (figure 1.1) au modèle SysML nous obtenons plusieurs fichiers VHDL-AMS (un par bloc) que nous utiliserons pour faire tourner la simulation. Nous montrons le résultat dans la figure 1.9. Il s'agit de la sortie du simulateur Hamster pour une entrée de force sinusoïdale.

On note que, malgré la variation non-linéaire des deux capacitances, la sortie est linéaire et suit le stimulus présenté en entrée. Cela est conforme à l'équation 1.1. Le côté gauche de la figure 1.9 nous permet de conclure que le mécanisme de détection de passage de seuil fonctionne correctement puisque la sortie numérique suit le signe de la sortie de l'ampli Op.

En regardant de plus près, nous voyons que la donnée numérique est en effet échantillonnée à pas fixe même si le signal analogique ne l'est pas. Le signal d'horloge *clk* génère des événements toutes les 2 μs, sur le front montant et sur le front descendant. Les détails

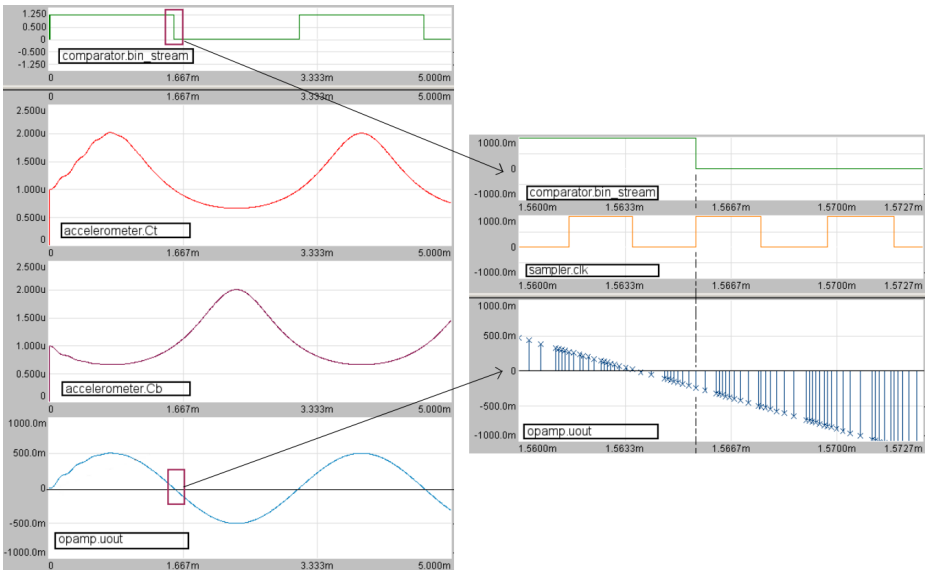

FIGURE 1.9 – Résultats de la simulation

à droite de la figure 1.9 montrent que le signal analogique était déjà négatif pendant plusieurs cycles de simulation avant d'être détecté par le comparateur. Cela se traduit par un délai de détection d'au maximum 2 μs. C'est le comportement attendu puisque notre contrainte d'adaptation impose un pas d'échantillonnage de 2 μs.

1.5 Généralisation de l'approche

1.5.1 Un formalisme intermédiaire

Ces deux derniers travaux ont pavé le chemin vers un formalisme intermédiaire généralisant ce qui a été fait pour SystemC-AMS et VHDL-AMS. Ce formalisme est le lien entre SysML et tout langage textuelle de simulation de système hétérogène. Nous avons essayé de capturer trois aspects importants :

- Modélisation hiérarchique avec des liens de composition et d'agrégation.
- Séparation de la structure et du comportement.
- Modélisation explicite de l'adaptation sémantique.

Le méta-modèle intermédiaire simplifié est affiché dans la figure 1.10. La première contrainte de modélisation hiérarchique est satisfaite par les liens entre les blocs **Module**, **Behavior**, **Composite** et **Atomic**. Un module de comportement atomique représente un bloc qui n'a pas des sous-blocs. Il est considéré comme le point final d'un arbre hiérarchique. Il peut, par contre, être inclus dans un module composite.

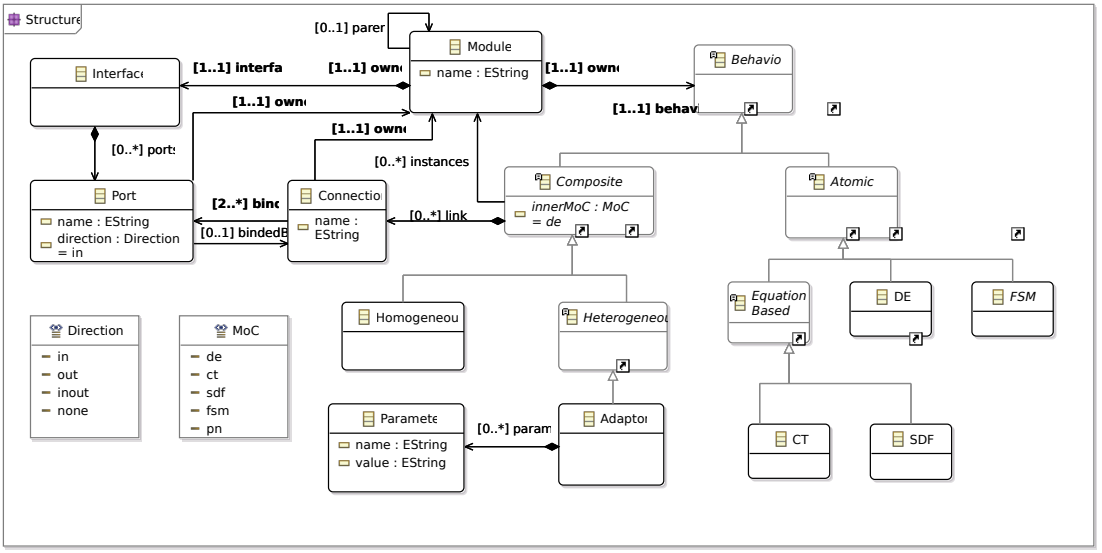

FIGURE 1.10 – Méta-modèle intermédiaire

La séparation de la structure et du comportement se fait par la définition d'une interface à travers la classe **Interface** et du comportement par la classe **Behavior**. L'interface n'est qu'un conteneur pour les ports. La classe de comportement atomique peut être spécialisée. C'est d'ailleurs ce qui va définir le modèle de calcul. Pour des raisons de simplicité je détaillerai ici seul le MoC des machines à état (FSM). Le lecteur intéressé peut consulter la description complète dans la section 4.7 en anglais.

L'adaptation sémantique est prise en compte dans la classe de comportement composite hétérogène. Un adaptateur peut être défini par une liste de paramètres ayant un nom et une valeur quelconque. Cela nous laisse la liberté de créer des paramètres selon chaque cas d'adaptation.

1.5.2 Exemple de définition de comportement

Une des spécialisations de la classe de comportement atomique est la classe FSM qui défini le comportement des machines à états finis. La figure 1.11 montre comment le comportement **FSM** définit le concept d'une machine à états et transitions. Chaque comportement FSM peut définir une ou plusieurs machines à états qui fontionnent en parallèle. Un machine à états possède des états qui ont des transitions sortantes. Chaque transition est activée quand un événement discret de sa garde se produit. Cette transition peut aussi générer une action. Cette action est généralement une attribution de valeur à une variable interne ou une sortie du module.

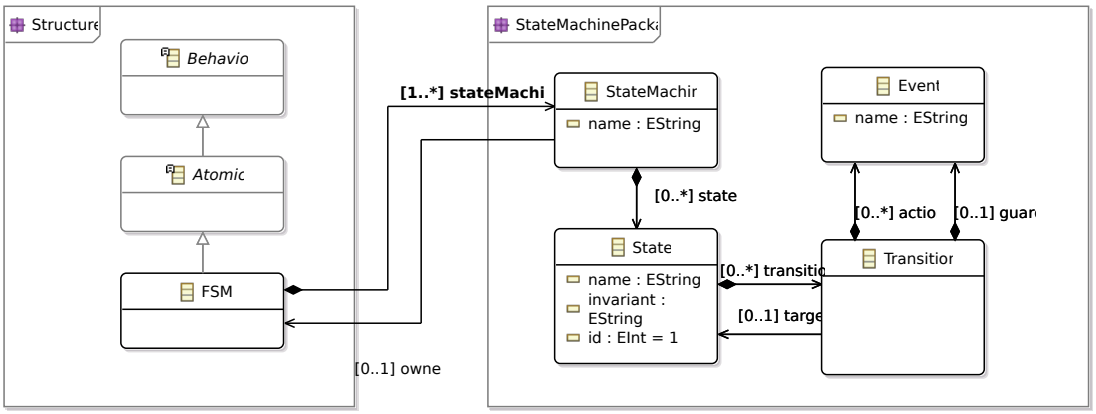

FIGURE 1.11 – Machine à états finis

1.5.3 Adaptation sémantique

L'adaptation sémantique est modélisée par la classe **Adaptor** de notre méta-modèle intermédiaire. Cette classe est définie par une suite de paramètres contenant une paire nom & valeur. Le nom du paramètre fait aussi partie de la sémantique d'adaptation. L'utilisateur doit choisir les paramètres qui lui conviennent le mieux. Par exemple : **Sampling**, qui prend la valeur *fixed* ou *dynamic* ; **Timestep** qui prends une valeur de temps. Les valeurs $15ms$ ou $2ns$ sont acceptées. **Data Resolution** prends un entier comme valeur pour définir le nombre de bits utilisés dans la représentation matérielle d'une donnée quelconque. **Input** et **Output** sont des paramètres pour définir les ports d'entrées et/ou sorties. La liste complète de ces paramètres peut être trouvée dans la section 4.7.3.

1.5.4 Une nouvelle approche

Le méta-modèle intermédiaire de la figure 1.10 est le cœur de notre nouvelle approche. Nous avons ajouté une nouvelle étape de transformation de modèles responsable de l'interprétation sémantique des éléments du langage SysML. En dépendant du modèle de calcul appliqué au bloc, il sera mis en correspondance avec un des différents domaines de notre méta-modèle. De même, les spécifications d'adaptation sémantique seront aussi mises en correspondance avec un ensemble de paramètres d'adaptation.

L'approche complète est affichée dans la figure 1.12. Nous y voyons le nouveau méta-modèle comme l'élément central d'où partent les transformations purement syntaxiques vers chaque langage cible : SystemC-AMS et VHDL-AMS. Les transformations $M2M_{2a}$ et $M2M_{2b}$ sont des versions simplifiées des transformations discutées dans les sections précédentes puisque toute interprétation sémantique a été basculée vers la transformation

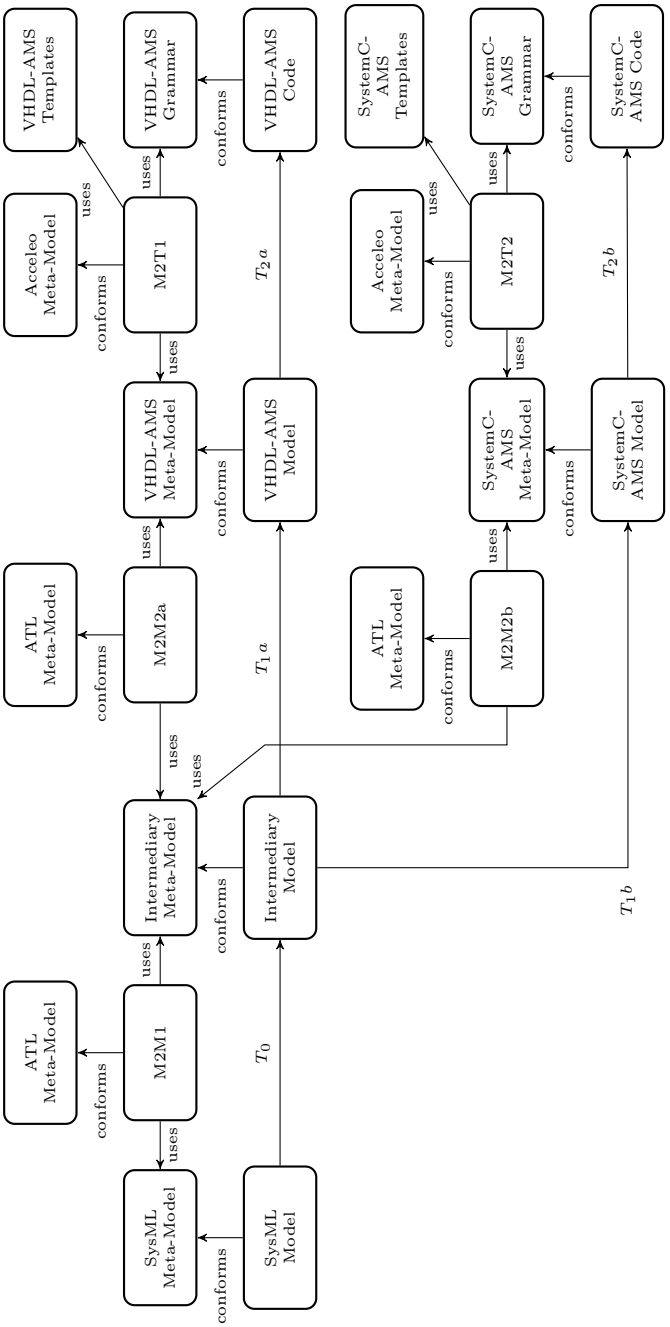

FIGURE 1.12 – L'approche complète

$M2M_1$. Le lecteur intéressé peut suivre un cas d'étude d'application de cet approche dans la section 4.7.6.

1.6 Conclusions

Dans ce résumé, nous avons montré brièvement un moyen de définir la sémantique concrète des diagrammes SysML à travers les stéréotypes de notre profil pour SysML. Nous avons également montré une manière de définir explicitement l'adaptation sémantique entre les blocs de différents domaines. Nous avons introduit le concept de bloc d'interface qui sert comme d'adaptateur pour d'autres blocs d'un autre domaine. Nous avons aussi mis en évidence un langage de spécification d'adaptation. Notre mini-DSL permet l'instanciation des adaptateurs existants ou bien la création d'adaptateurs non-standards. Les trois expériences de ce résumé montrent les améliorations progressives de notre technique.

La première expérience montre une technique qui s'appuie sur les commentaires des diagrammes SysML pour définir l'adaptation sémantique. L'utilisation des commentaires n'est pas considérée comme une technique standard de l'ingénierie dirigée par des modèles et pour cette raison, et grâce aux retours de la communauté, nous avons amélioré notre approche pour remplacer ces commentaires par des stéréotypes définis dans un profil SysML séparé. Cela a été implémenté dans la deuxième expérience ciblant VHDL-AMS avec pour cas d'étude un accéléromètre MEMS.

La sémantique de chaque MoC est donnée par les contraintes SysML et est utilisée par nos transformations de modèles, en particulier, la première étape de notre approche. Cette étape consiste dans une transformation "model-to-model" (M2M). La correspondance entre les éléments de SysML et les éléments du langage cible se fait à travers ces contraintes. Cela nous permet de choisir correctement l'élément approprié pour un MoC donné. L'adaptation sémantique se fait aussi au niveau de cette transformation M2M. Les paramètres d'adaptation sont utilisés pour guider la transformation qui, à son tour, devra choisir un adaptateur de la bibliothèque standard (cas de la première expérience) ou en générer un s'il n'existe pas (cas de la deuxième expérience).

Ensuite, nous avons présenté une généralisation de l'approche. Un formalisme intermédiaire a été présenté en y ajoutant les concepts de chaque modèle de calcul ainsi que des éléments nécessaires pour l'adaptation sémantique. Malgré l'ajout d'une étape additionnelle de transformation, nous croyons que cette modification dans l'approche améliore l'extensibilité et aussi la maintenance en séparant les transformations sémantiques des transformations purement syntaxiques. Avec ce nouveau méta-modèle intermédiaire, nous simplifions les transformations syntaxiques puisque toute l'interprétation sémantique est faite par la première étape. Cela n'est pas vrai pour les deux premiers essais de la section

1.3 et 1.4. Les transformations M2M de ces deux essais sont en même temps, une interprétation sémantique du modèle SysML et aussi une interprétation syntaxique entre SysML et le langage cible (SystemC-AMS pour le premier et VHDL-AMS pour le deuxième).

Ce travail a été implémenté dans l'environnement de développement Eclipse. Il dépend des outils de modélisation d'Eclipse. Tous nos méta-modèles sont basés sur le méta-modèle Ecore et nos transformations écrites en ATL et Acceleo. Nous avons noté un manque d'une base de données communautaire pour la réutilisation des méta-modèles. La plupart des travaux de modélisation des langages SystemC et VHDL faites ici peuvent être réutilisés dans les cas d'usage de génération automatique de code ou bien pour faire des analyses de modèles. De même, le travaux faits auparavant pourraient être réutilisés ou améliorés si les méta-modèles étaient disponibles librement en ligne dans un dépôt commun à la communauté. De même que pour les langues littéraires, tel que l'anglais ou le français, nous ne connaissons qu'une petite partie de la langue. C'est d'ailleurs ce qui compose notre vocabulaire. Une meilleure modélisation des langages de programmation, sous la forme de méta-modèles, pourrait être faite si nous avions à notre disposition des outils de travail collaboratifs de création et déploiement des méta-modèles des langages existants.

Par rapport aux travaux précédents [2, 12, 30, 47, 56, 57, 59, 64] nous nous distinguons par la manière de traiter l'adaptation sémantique. Aucun des travaux précédents n'a traité les inconsistances que l'hétérogénéité introduit. Le langage d'adaptation sémantique, notre mini-DSL, nous permet de traiter cela en modélisant les adaptations par des paramètres divers. HetSC [18, 37] a une approche similaire mais sans traiter le cas de la frontière continu-discret, et donc ne considère pas l'extension AMS de SystemC. UniverCM adopte une vision différente de la notre en essayant d'homogénéiser un système hétérogène dans un seul formalisme. Nous adoptons une stratégie différente puisque nous croyons que l'utilisation des différents formalismes est un atout pour la modélisation des systèmes hétérogènes et que la maîtrise des interfaces multi-domaines fait aussi partie du travail de modélisation. C'est d'ailleurs pour cette raison que nous insistons sur le fait de modéliser *explicitement* les adaptations sémantiques. Cela nous permet d'éliminer les incohérences des simulations dans les simulateurs où la complexité est cachée à l'utilisateur. Sans l'adaptation sémantique, l'utilisateur risque d'avoir de nombreux problèmes qui apparaissent à cause de l'intégration.

1.7 Perspectives

Grâce à sa généralité, cette approche peut être étendue à d'autres langages textuels. L'approche à été développée de telle sorte que la sémantique des modèles de calcul et

les adaptations sémantiques sont prises en compte par la première partie de la chaîne de transformations. Le reste de la chaîne de transformations sont des transformations purement syntaxiques et donc facilement écrites par un expert du langage cible.

L'inclusion d'autres modèles de calcul est aussi possible. Cela nécessite l'inclusion des concepts de chaque modèle de calcul dans le méta-modèle intermédiaire. Différemment d'une simple extension à d'autre langages textuels, ce changement nécessite des modifications plus profondes dans la chaîne de transformations. D'abord, on commence par l'extension de la classe "Atomic" dérivée de la classe "Behavior" qui représente l'ensemble des modèles de calcul. Ensuite, ce changement doit être pris en charge par chaque transformation M2M ciblant un langage textuel et qui supporte le MoC en question. Même si cela peut paraître compliqué, l'ajout d'un MoC ne signifie pas que tout devra changer, au contraire. La séparation des MoCs par extension d'une classe abstraite permet que cette tâche soit non-intrusive, garantissant ainsi qu'il n'y aura aucune modification dans le code, où dans un modèle qui fonctionnait auparavant.

Nous pouvons aussi imaginer l'usage de cette représentation intermédiaire pour réaliser des analyses tel que la validation des modèles ou bien faire du "model checking". Cette représentation intermédiaire a toutes les informations nécessaires pour, par exemple, calculer les ordonnancements possibles d'un modèle SDF. Ces vérifications peuvent détecter des problèmes de conception dans une phase précoce de développement.

Chapter 2

Systems Modeling Overview

2.1 Introduction

Designers face a rapid increase in the complexity of electronic systems nowadays provoked by two main factors: on the one hand we have the continuous drive for smaller devices with more sophisticated computational resources, embedded sensors, etc., while on the other hand there are new demands from the market concerning power consumption, safety, reliability and faster product cycles. To meet these challenges, industry requires new project methodologies that respond to the need for testing as soon as possible, thus extending the classical "V cycle" to higher levels of abstraction.

Up to now, electronic designs would start from specifications and proceed to an architectural exploration. Once the architecture is selected, a detailed conception with optimization of parameters would lead to the final implementation in a given target technology. All of those steps are evidently followed by corresponding tests checking compliance to strict specifications and quality assurance verifications. When this flow is applied to designing today's complex systems, things can get more complicated because of their intrinsic multi-domain nature. One has to coordinate the work of specialists of many different domains in order to successfully build systems of this kind. A good example of this kind of situation is the design of System-On-a-Chip products.

The advent of a System-On-a-Chip (or just SoC) is the natural result of the evolution of systems towards smaller and fully integrated products. A SoC is a single integrated circuit encapsulating many components in an electronic system. It may include digital processors, analog blocks such as Radio Frequency (RF) transceivers, and mixed-signal blocks such as analog/digital converters or Phase-Locked Loops (PLLs). To be considered as a SoC, all of these subsystems must be manufactured in a single die.

It appears that we are reaching a technological barrier for the miniaturization of transistors. Up to a few years back, computational power of integrated circuits has evolved following the trend that the number of transistors for a given silicon surface doubles every 18 months. This observation was first made by Moore (CEO of Intel) in 1965, and has remained valid until recently. The diminishing gains expected to be achieved in raw integration density in the near future has pushed researchers to invent new concepts to replace Moore's Law as the driver of electronic systems advances. This evolution challenge has been called "more-than Moore" and it purports to maintain the computational power evolution observed by Moore in 1965. The path for this is to add to the system new functions that do not necessarily scale according to Moore's Law, but provide additional value to the end customer in different ways. This functional diversification comes in the form of power control, multiple processors architectures and non-digital functionalities such as RF communication, tactile displays, sensors and actuators.

These systems clearly have heterogeneous interactions since the base components are modeled in different domains. For instance, a simple system composed of an analog receiver coupled to a digital signal processing unit can be classified as heterogeneous because the digital part will be modeled probably by a discrete event model, while the analog part will be modeled by equations in the continuous time domain. The synchronization among those models can be quite challenging and demands extra effort from the designers.

When designing a system, one must first start from an architectural exploration to find the optimal algorithm/hardware solutions for the specific system. It's paramount to obtain preliminary estimates of power consumption and timing constraints imposed by the real-time embedded applications. That might be carried out using an executable model from the very starting point of the project: that is what we call an executable specification. In addition, sometimes the best way to get familiarized with the functionality of a block is by using it and trying it out [4].

2.1.1 Motivation & Problems in industry

Automated electronic systems synthesis started simultaneously with the appearance of hardware description languages in the 80's, following efforts such as the United States government's Very High Speed Integrated Circuits (VHSIC) program. This program gave birth to the VHSIC Hardware Description Language (VHDL). VHDL first purpose was to serve as a documentation language for integrated circuits with the goal to replace huge and complex manuals. Its structured and hierarchical form of describing concurrent processing architectures, intended for conveying executable specifications of

digital systems, made it also useful for automatic synthesis of logic circuits. This was a first step toward abstraction. Switching from transistor level to gate level and later, Register-Transfer Level (RTL), was a remarkable advance for electronic circuit designers. It allowed the development of more complex designs with less work. The use of VHDL as a systems description language was further strengthened when it was approved as an IEEE standard in 1987, and the release of the IEEE 1076-1987 VHDL language reference manual. The latest version at the time of this writing is the 2008 review [5]. In parallel to the birth and growth of VHDL, there was the development of Verilog, another hardware description language that at first was a proprietary verification and simulation product. Verilog also became an IEEE standard, in 1995 (IEEE 1364-1995) and its latest version is the 2005 review [63]. Both languages are now firmly entrenched as digital systems description languages for simulation, synthesis and verification.

With the continuous increase of complexity and level of integration in nowadays digital systems, the same motivation as before has pushed designers to gradually migrate to newer design methodologies that would allow them to create systems capable of handling complex functions either in hardware or software, without distinction. Several approaches and languages have been introduced, but concerning digital systems design it can be argued that industry and academic researchers have converged to the SystemC language and simulation kernel.

SystemC is a C++ library for designing and simulating complex digital systems. It provides an extensive set of constructs and extensions to C++ in order to enable easy and fast modeling at various levels of abstraction. The main motivation for the creation of this language is that complex digital systems are difficult to simulate due to the enormous quantity of gates. This leads to unacceptably long simulations, making it unfeasible to verify complex functions relying on software. Using a C-based approach enables designers to employ higher levels of abstraction such as the transaction level modeling (TLM) methodology released in 2005 by the OSCI group. The TLM philosophy replaces all pin connections by a simple transaction, in other words by a function call. For example, imagine that you have a processor connected to a 32-bit wide bus making the communication to the memory and other devices. This would be very time-consuming to model and as well as to simulate at the RTL level using a language such as VHDL or Verilog. Replacing all the wiring and logic blocks by a simple function call speeds up the simulation and facilitates the modeling task. This is the key advantage brought by SystemC.

A similar change in the way of designing digital systems had already happened before, when designers moved from gate level design to RTL (Register Transfer Level) design. I am not here saying that RTL is no longer useful, it is. Automated synthesis from TLM

abstraction layer to a physical implementation is still an unresolved challenge. Today, TLM abstraction is mainly used to speed-up the time to market of SoC products, by concurrent design and verification of hardware and software components. With this new level of abstraction it is now possible to start the development of embedded software at the same time as the hardware design. This allows a faster design flow for a SoC [24]. The widespread adoption of this new design process was only possible at a large scale because SystemC is an open source library available to everyone.

The work leading to the SystemC language was inspired by earlier ideas, such as the Signal Processing library Using C++ (SPUC) from Tony Kirke [41]. The SPUC library was created specifically for system designers to allow a simple transition from system design to implementation on a defined hardware platform. This open source library provided a much faster simulation environment than Matlab and offers a large library of reusable DSP building block objects written in C++.

2.1.2 Models of Computation

Significant efforts have been aimed at creating formalisms to concisely describe the structure and behavior of systems in the form of models in many different domains (chemical, electrical, digital logic, programming, etc.). These different modeling paradigms are tuned to the engineering practices of the designers and restricted to their view of the system under consideration. As a consequence, each modeling paradigm can be said to follow a model of computation (MoC). Hence, a MoC is nothing more than a set of rules that gives a semantic to a structure and thus allows the evaluation of the model. This evaluation is expected to give some insight on the properties of the real system, once it's built. The most frequently used MoCs are described below.

Continuous-Time (CT) models are based on differential equations that describe the time variant function of a model. They are generally used for modeling mechanical dynamics, analog circuits and other physical systems. A continuous time model simulator is typically based on ODE (ordinary differential equations) and DAE (differential algebraic equations) numerical solvers. One of the most famous example in the analog circuit design field is the SPICE simulator created in 1975 by Laurence W. Nagel [50].

Discrete event (DE) models are well-suited for digital circuits modeling. This MoC is based on a global notion of time-stamped events where any action is taken only when an event happens. VHDL and Verilog are two hardware description languages that are based on this MoC. They are usually associated with a delta-cycle simulator with distinct phases of evaluation and update.

Finite State Machine (FSM) formalism is used for describing control sequences. It is composed of states, transitions, inputs and outputs. States are generally represented by circles and represent a stable position of the automaton. Transitions are represented by arrows and are triggered by events or conditions. It's well-suited for the description, verification and automatic synthesis of digital logic implementing flow of control structures. There are well-understood techniques for handling FSM descriptions, hence all automatic synthesis tools are capable of creating a logic circuit from a FSM chart [26]. FSM charts do not have a notion of time, continuous nor discrete. States become active or inactive instantaneously, according to guards and inputs.

There are many other MoCs where the notion of time is abstracted away, which is the case of **Synchronous Reactive (SR)** models. In this MoC, time is simply an ordered sequence for the evaluation of inputs and outputs, which are initiated by environmental events. The SR MoC is widely used in software engineering for modeling real-time applications. Examples of this MoC can be found in languages like Esterel, Lustre, Signal, Argos and Statecharts [7, 31].

We can move one step higher in the abstraction process by considering that the ordering relation of processes only exists among a subset of events in the system. This is the case of Synchronous message-passing models such as Hoare's **communicating sequential processes (CSP)** and Milner's calculus of communication systems. These MoCs where created in order to study the non-deterministic nature of distributed programs. We all know that a program may not respond the same way to the same inputs because of possible reordering of messages in different executions times. The synchronous Message-passing MoC was created to ensure synchronous ordering of messages for any algorithm. Examples of implementations of this MoC are the languages Lotos and Occam.

Other efforts have focused on asynchronous modeling such as **Kahn Process Networks (KPN)** [25]. This MoC was originally created for the study of parallel computing where processes communicate through FIFO queues. KPNs are now used for modeling signal processing systems. For example, some researchers have used KPN networks to directly map an algorithm into FPGA or multiprocessor platforms [6, 51].

DataFlow MoCs also use FIFO queues as the main communication channel between processes. They are extensively used for specifying signal processing algorithms, because their properties allows one to analyze memory requirements and detect deadlock situations. One popular kind of Data Flow is the **Synchronous Data Flow (SDF)** MoC where every process is represented in a graph composed of arcs and nodes. Each node represents a function and each arc represents a signal path. SDF is a special case of data flow that has the property that the number of tokens (information units) produced and

consumed by a process is fixed, hence SDF network evaluation can be scheduled statically, that is, at compile time [44]. Other common examples of Data flow are Boolean DataFlow and Cyclo-static dataflow (CSDF).

All of these MoCs are focused to specific aspects or potential problems of the whole system, but we are interested in solving the heterogeneity problem. It's useful to remind that modern embedded systems are intrinsically heterogeneous because of the diverse natures of components: digital, analog, mechanical, optical, thermal, etc.

Suppose we have a system with an analog RF transceiver, one or more Digital Signal Processors and embedded software. For each part of the system there is a MoC that fits the best. The analog part would be modeled using a continuous time MoC, the digital part would be modeled in a discrete event MoC because of its amenability for automatic synthesis, and the algorithms of the embedded software could be modeled either with KPN networks or SDF. All of this diversity is present in our complete system and small local modifications to optimize a subsystem could have an impact on the conception of the whole. The interaction between the subsystems, modeled with different MoCs, is not always intuitive and requires a lot of work to get right.

2.1.3 Heterogeneous modeling languages & tools

Modeling heterogeneous systems is not an easy task. The obstacles and issues arising when trying to implement such models have been the focus of considerable research. One of the most cited developments on this field is the Ptolemy project [20]. The Ptolemy approach handles the heterogeneity problem with what they call hierarchical heterogeneity. This is actually a model structure and semantic framework that treats heterogeneity in a structured manner. This is done by dividing a complex model into a tree of nested sub-models which run locally homogeneous. The interactions between models are allowed through mechanisms specified at different levels in the hierarchy. These mechanisms cover the flow of data and control among the models. This concept gave birth to the Ptolemy II software environment that provides support for modeling, simulation and design of complex heterogeneous systems with a high level of confidence [43].

Ptolemy II [20] handles heterogeneity by hierarchy. Components are nested in black boxes called actors for which the semantics of execution and communication are defined by an entity called *Director*. A director defines how a model should behave and how its components communicate, in other words, it defines the model of computation (MoC). An actor can be transparent or opaque regarding its parent: if the child actor does not have its own director it is considered to be transparent and will inherit its parent director,

if it has its own director, it is considered to be opaque (like a black box). In Ptolemy, computation and communication semantics are defined for a large set of MoCs. These include Process Networks, Dataflow, Discrete Event, Finite State Machines, Continuous Time and others. Unfortunately, Ptolemy does not provide explicit ways to define adaptations between models that use different MoCs. For example, interactions between discrete event (DE) and synchronous dataflow (SDF) models can result in redundant events in the DE domain if a given value does not change. In the same way, an SDF model might not be regularly activated as discussed in [11]. This can cause some confusion if the user is not aware of the default adaptation performed by Ptolemy. Extra modeling effort may be required if a specific behavior is expected.

Ptolemy's approach on heterogeneous systems inspired other works such as ModHel'X [32]. ModHel'X was developed to explore semantic adaptations in heterogeneous models. It proposes a flexible framework for the development of heterogeneous systems separating the model's definition from the MoC's definition. In ModHel'X, a generic execution environment was created to allow the definition of several models of computation. Following the same principle of actor-based modeling, ModHel'X defines blocks whose behavior is determined by a MoC (equivalent to Ptolemy's *Director*) and introduces an interface entity capable of making the necessary adaptations among different MoCs (i.e. data, control and time). To do so, ModHel'X improves upon the execution algorithm of Ptolemy by the introduction of an adaptation phase right before and after the "fire" phase. This yields an effective way to define the semantics of the interactions between different models of computation. However, the current implementation of ModHel'X is based on a non-standard metamodel which makes it hard to integrate with existing toolchains.

2.1.4 From Ptolemy and ownward

Ptolemy inflenced many developments on the field of systems modeling and simulation. Among them one can count the popular SystemC standard [39]. This C++ library has firmly established itself as the most important system-level specification language for electronics, providing simulation capabilities in an early phase of development. Another main feature of SystemC is the ability to describe and perform automatic synthesis of digital hardware from a subset of the language [60]. Internally, SystemC is based on a dedicated discrete event (DE) simulation kernel capable of modeling concurrent processes. However, using only a DE MoC turns out to be unsuitable for modeling several application domains.

Despite its flexibility, researchers realized that SystemC alone was not suitable for modeling heterogeneous systems and thus, many extensions were developed to address this

shortcoming. One of the first efforts in this direction was the mixed signal extension for SystemC described in [8]. The authors of this work have built on top of the SystemC core a suite of C++ classes which they call "Analog Extension Classes" (AEC). Those would enable the integration of analog and mixed-signal systems with simple SystemC syntax. Their main goal was to use it in applications such as sigma-delta converters, image sensors and discrete time analog filters.

The interactions among different MoCs are hard to model and may result in semantic conflicts. One interesting solution was shown in the HetSC library [37]. HetSC is an extension library for SystemC to support the modeling of several MoCs. HetSC deals with heterogeneity by the use of configurable converter channels where time and data adaptations are clearly specified. HetSC allows the user to choose if there will be data loss, interpolation or even an exception thrown [18]. These choices are necessary to resolve semantic conflicts among different MoCs.

2.1.5 Multiple MoCs with SystemC

There are many attempts to extend SystemC capabilities in order to allow a heterogeneous simulation environment where different MoCs would communicate properly. For example, we can cite Hiren D. Patel and Sandeep K. Shukla work on "SystemC Kernel Extensions for Heterogeneous System Modeling" [54]. In this book, they explain that the original Discrete Event simulation kernel of SystemC may not be appropriate to execute other models of computation such as CSP, FSM or SDF. Their work consisted in creating three different kernels for the following MoCs: FSM, CSP and SDF. Those MoCs have their own properties that could enhance simulation speed as shown for the SDF example in pages 88-92. Benchmark results for using specific execution kernels on SDF models improved the simulation speed up to 70%. They have also created an API that would allow each kernel to gain access to its counterpart kernels. Each execution kernel runs independently of the others and they are all able to access every other kernel for a multi-domain heterogeneous simulation. Unfortunately, there was no support for the continuous-time domain.

One approach to integrate the analog continuous time MoC extension on top of the SystemC library was the SEAMS project [3]. SEAMS stands for SystemC Environment with Analog and Mixed Signal extensions. This library adds a general-purpose analog solver to provide analog modeling capabilities to SystemC. It provides not only an analog kernel simulator but also the link to the standard SystemC DE environment. This connection is necessary because analog solvers do not work the same way as the digital simulation kernel. Analog simulators don't use events to synchronize, instead they use

time control, namely continuous step-size adjustment to optimize the simulation time against numerical precision trade-off. To communicate between the analog and the DE kernel, the solution adopted by SEAMS was to use the lock-step algorithm. The analog kernel advances until the current simulation time and schedules an event to the next time step before suspending. Then, the digital kernel takes control and advances simulation time until it reaches the scheduled analog generated event. This analog-driven synchronization is very similar to earlier mixed-signal SPICE-like simulator implementations, now applied to the SystemC DE simulation kernel.

All of these ideas influenced the developement of SystemC-AMS [28], the Analog and Mixed-Signals (AMS) extension for SystemC. SystemC-AMS provides pre-built MoCs allowing co-simulation of continuous and discrete components. As an answer to the heterogeneity problem, SystemC-AMS provides support for MoCs closer to the continuous-time domain such as the Linear Signal Flow (LSF) and Electrical Linear Networks ELN. These are two different ways of representing differential equations on the continuous-time domain, both built on top of a linear differential algebraic equation solver synchronized to the discrete event simulation kernel of SystemC. The Electrical Linear Network (ELN) method uses the topology of the circuit to determine Kirchhoff equations. The Linear Signal Flow (LSF) method is similar to the Simulink approach. It allows the use of base blocs such as adders, mixers, integrators and differentiators to model a system using the Laplace transfer function. There is also embedded support for a timed variation of the SDF MoC : the Timed Data Flow (TDF) which remains the SDF MoC with time tags for every sample. TDF models are based on the Discrete Event simulator of SystemC and are synchronized through a scheduler. Figure 2.1 illustrates these different modeling paradigms.

The earliest prototype of SystemC-AMS was called ASC-library [29]. It provided a set of classes to model analog behavior. Later, in 2005, after several enhancements, SystemC-AMS was submitted to the OSCI (Open SystemC Initiative), and became a standard in 2007. At the time of this writing, the latest version of SystemC-AMS library is the 2.0-review. This means that SystemC-AMS 2.0 is in the form of a proposed open standard undergoing review by the public. There is a proof-of-concept of the 1.0 version released by the Fraunhofer Institute for Integrated Circuit IIS, which can be downloaded on their web site [http://systemc-ams.eas.iis.fraunhofer.de].

Since the SystemC-AMS analog extension does not provide support for many MoCs, there are other efforts trying to fill this gap, such as the work by M. Damm, J. Haase, C. Grimm, F. Herrera, E. Villar entitled “Bridging MoCs in SystemC Specifications of Heterogeneous Systems” [18]. They have also worked on adapting different MoCs, but this time, only using the SystemC base kernel. For that, they have used a methodology called HetSC

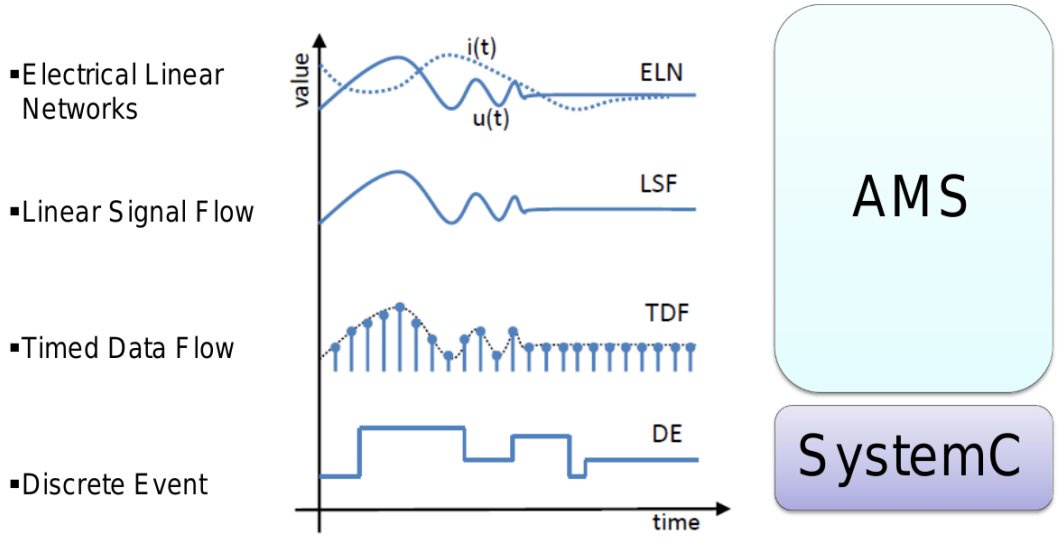

FIGURE 2.1 – Different SystemC-AMS data models

which is a set of rules and a library for enabling heterogeneous specifications of complex embedded systems. HetSC [35] defines coding guidelines for each specific MoC thus facilitating the system designer task. The HetSC library facilitates the specification of interfaces and channels for all the connections between the MoCs. The main contribution of their work consists in a converter channel between SystemC-AMS T-SDF MoC and HetSC BKPN/KPN (timed or untimed) MoC. SystemC-AMS could be used together with HetSC if one needs to support a wide range of MoCs [38].

A different approach is UniverCM. Instead of embracing heterogeneity, UniverCM [19] tries to convert an heterogeneous design into an homogeneous one using an intermediary model of computation that would, in theory, be capable of representing models in the continuous time domain and discrete event. They have proposed an heterogeneous intermediary format that much resembles an hybrid automata where continuous and discrete behavior can be equally modeled. This strategy does not comes for free. For HDL processes for instance, a model of the update-evaluate and time-advancing scheduler is required to obtain the same behavior as HDL simulators.

From UniverCM, it is possible to generate executable C++ code using the SystemC simulation library. Different models written in different languages can also be mapped to UniverCM. These are : Software models written in C, discrete event models written in VHDL and even analog models written in the Compositional Interchange Format (CIF). With UniverCM, there is no guarantee that generated models are identical to original models converted to this formalism. Results obtained from generated code are not expected to match exactly simulations ran with the original models. This can be a drawback if simulation consistency across tools is a must.

From all of these published results, we can see that there is a growing interest from the industry and from the academy on increasing the modeling abstraction level to improve the effectiveness and efficiency of electronics systems design workflow. The field is currently aiming at filling the gap between the system-level specifications (functional requirements) and hardware/software implementation. One idea is to use UML diagrams as a starting point to specify and generate detailed documentation for the whole project. We believe we can go further if we use similar diagrams to automatically generate executable models directly from system-level specifications.

2.2 Graphical Modeling Languages

Model Driven Engineering started in 1999 with the UML standard. Because of it's extremely generic semantics, many profiles were created in order to specify its usage either by using a limited number of features or by extending the UML base meta-model.

One of the first initiatives for the specification of SoCs using graphical languages was "UML for SystemC", based on UML 1.4. This standard was not adopted because it was too generic, preventing designers from creating detailed models. The Object Management Group (OMG) tried to resolve this issue, encouraging the Model-Driven Engineering (MDE).

Model-Driven Engineering appeared as an attempt to standardize this new way of developing systems starting from a higher abstraction layer: a model based on the MOF metamodel.

2.2.1 SysML

SysML appeared as a response to OMG MDE initiative to focus on the modeling of systems. It is now an industry standard for a graphical system-level specification language with the purpose of modeling complex systems in different domains. It extends some of the UML 2.0 features, notably with the new requirements diagram and the parametric diagram. SysML is used to manage complexity while improving communication among different teams. The use of SysML diagrams facilitates the documentation and specification regarding system requirements and constraints on property values [22].

SysML contains the necessary framework for modeling heterogeneous system. SysML was conceived to serve as the essential support for system engineering, including considerations of multi-paradigm modeling with different views and requirements. Unfortunately, SysML is very generic and lacks semantics to execute models. Consequently,

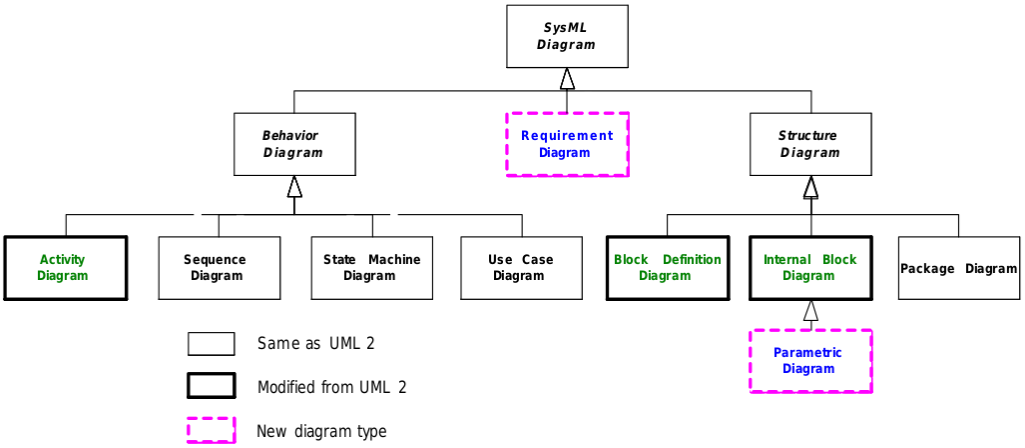

FIGURE 2.2 – SysML Diagrams

the interpretation of the modeling elements is left for the users, that typically use it for documentation and communication among members of an organization.

Indeed, SysML does not provide built-in simulation capabilities but offers great flexibility for modeling large heterogeneous systems. For our purposes, heterogeneous systems are systems that present components of different nature which are modeled using distinct formalisms. A heterogeneous model thus involves at least two different modeling formalisms with bound interfaces, i.e. exchanging data, control or even sharing different time scales. The challenge that we face is to give SysML diagrams explicit and executable semantics so that heterogeneous models can have coherent behavior for simulations across different tools.

The meaning of SysML structural elements is not given by the standard itself. Users of SysML commonly give their own (often implicit) interpretations of how each language element behaves. Block definition diagrams and internal block diagrams are examples of purely structural diagrams with no behavioral semantics. A SysML connector, for example, can represent a wire, a physical chain or even a function call. State machine diagrams and activity diagrams have behavioral semantics, but they can lead to non-deterministic behaviors if not defined carefully. For example, a state machine in SysML lacks the definition of priority for state transitions. For a given state, if two possible transitions happen simultaneously, one cannot know which state to choose next. The problem of the lack of clear semantics is even more severe in heterogeneous models. The interaction of a finite state machine monitoring inputs from a continuous time model is a classical example. From the SysML perspective, there is nothing that defines explicitly when a guard should be evaluated or with what precision continuous time data should be monitored.

One way to describe the behavior of a model is by defining its MoC [45]. The MoC details how components of a given system interact, how they exchange data, control and notions of time. A large set of MoCs have been well detailed in [13]. Some examples of commonly used MoCs are: DE, CT and FSM. The challenge is how to combine them seamlessly so as to run simulations of the whole system with predictable results.

2.2.2 MARTE & Gaspard 2

MARTE [61] is a standard proposal of the OMG. It stands for Modeling and Analysis of Real Time Embedded systems. This profile adds some capabilities to UML in order to support Model-Driven Development of Embedded Systems. From OMG : "MARTE consists in defining foundations for model-based description of real time and embedded systems. These core concepts are then refined for both modeling and analyzing concerns. [...] Especially, it focuses on performance and schedulability analysis. But, it defines also a general framework for quantitative analysis which intends to refine/specialize any other kind of analysis."

In this thesis though, our focus is automated code generation and semantic adaptation for simulation of heterogeneous models. Some researchers have applied the MARTE profile for similar purposes. One of the most interesting ones in this field is Gaspard 2. Gaspard 2 [55] is a design environment for the development of Multi-Processors System-On-Chip (MPSoC). Gaspard 2 allows the execution of a subset of MARTE models. This is achieved by a chain of model transformations from the MPSoC specification to SystemC code as shown in figure 2.3.

Gaspard 2 stands for Graphical array specification for parallel and distributed computing. This design environment is a subset of the MARTE profile aimed at signal processing systems. Code generation is done using a library of elementary components that are linked to existing code. In [55] an H.263 encoder in implemented using the Gaspard 2 approach. This framework was used to explore the different configurations of a MPSoC and check the impact on the encoder performance (for instance, by changing the number of processors of the architecture). The Gaspard 2 approach is interesting when we want to do architecture explorations, but it still requires manual writing of code templates. The framework is not capable of automatically generating the tests (checking code). Those have also to be written manually by designers.

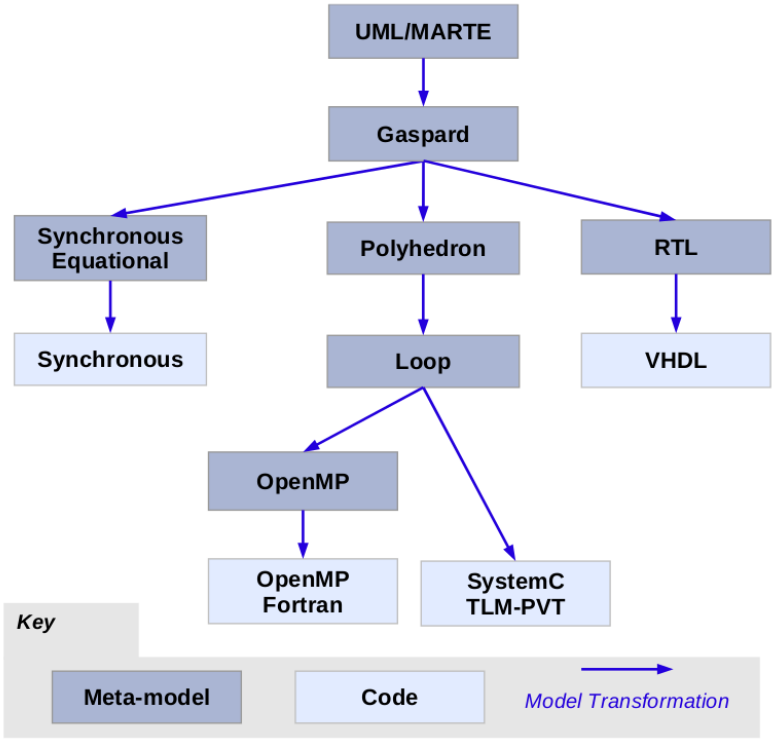

FIGURE 2.3 – The Gaspard 2 transformation chain

2.2.3 From SysML/UML to SystemC

Several publications have addressed the translation from modeling languages to executable code. Raslan et al. [57] have defined a mapping between SysML and discrete event SystemC in order to raise the abstraction level of electronics designs and speed up the design process. Prevostini et al. [56] proposed a SysML profile to model SoCs and provided an engine to automatically generate SystemC code. By using parametric diagrams they were capable of defining equation constraints, and by using allocations in activity diagrams they managed to co-simulate hardware and software together. Mischkalla et al. [47] came up with a methodology based on an emulated processor using SystemC TLM and reported being capable to automatically synthesize combined hardware and software through a series of code generations and tools synchronizations. They support only the synthesizable subset of SystemC to guarantee that the hardware modeled with SysML is synthesizable.

All of these approaches have addressed the integration of SysML with the SystemC discrete event simulator or a synthesizable subset of SystemC, but have not considered the intrinsic multi-domain characteristic of heterogeneous systems. We intend to raise the abstraction level of heterogeneous systems designs using SysML diagrams together with a set of semantic definitions for each MoC. That will not only improve systems design

comprehensibility with high-level graphical descriptions but also provide executable semantics to SysML diagrams, allowing us to run simulations from the specification models.

2.2.4 From SysML/UML to VHDL and VHDL-AMS

Substantial work has been carried out to apply SysML/UML to the design of electronic (analog and digital) systems. Many researchers focused on the generation of VHDL-AMS code from SysML diagrams. D. Guihal [30] and J. Verriers [64] extended the VHDL metamodel proposed in [2] and [59] to use AMS constructions in their code generators. J.-M. Gautier et al. [12] used model transformations to generate VHDL-AMS code from SysML Block Definition Diagrams and Internal Block Diagrams. They have used block constraints to define physical equations in VHDL-AMS modules.

Although these previous efforts have shown methods to generate VHDL-AMS code from SysML diagrams, they have not dealt with the semantic inconsistencies that heterogeneity introduces. We will present in chapter 3 a technique to deal with this problem. We will show how to use ModHel'X's good practices of stating the semantics of different components and explicit modeling of the semantic adaptation between heterogeneous components into an industry standard modeling language: SysML. In our approach SysML acts as a pivot language from which we generate executable code for widely deployed languages, such as SystemC-AMS and VHDL-AMS.

2.3 Textual vs Graphical Modeling Languages

Before describing what are the main differences between these two classes of languages, I'd like to emphasize that we are here discussing modeling languages and not programming languages. Modeling languages are mostly declarative languages with clear syntax but not always with clear semantics as we shall see further in this text. Programming languages on the other hand offer more freedom to model behavior and have clear semantics. That's because they are tightly related to an executable code that will run in a machine. Models, on the other hand, try to capture a concept or a point of view of a given reality. They tend to be more specialized languages with a syntax that reflects elements of that reality. This kind of approach would not be possible in programming languages because of the need to create any kind of behavior.

With that in mind, I'd like to start this comparison by a simple but very elucidative citation: "You will never strike oil by drilling through the map" by Solomon Golomb *Mathematical models – Uses and limitations* in the Aeronautical Journal, 1968 and latter

in the IEEE transactions in 1971. [27]. This refers to the simple fact that the models that we design are in fact a representation of a reality limited by what we wish to describe and by what we wish to analyze. They are meant to simplify the complex reality that we live in so that we can evaluate certain properties or obtain some sort of information that wouldn't be obvious otherwise. The example of the map makes this evident. A map can give a lot of informations about a terrain like altitude, mineral resources or even population density. These informations are easily accessible without the need of actually visiting the terrain. Another example, maybe more interesting for engineers, is the Fourier transform. The Fourier transform can take a time representation of a signal to a frequency representation of the same signal. That simple transformation give us knowledge such as the frequency components of a given signal, that otherwise, in the time domain, would be hard or impossible to discern.

In any case, the properties we assert about systems are in fact not properties of the implemented system but only properties of the model. Anything you prove, by simulations or analytically, will be valid only for the model, not the real system. The question that arises naturally is how can we build complex systems using models and how do we build confidence on a given modeling language? To answer that question I'll give the example of VHDL. The language started as a documentation language, but quickly gave rise to a simulation platform were engineers could test and experiment digital designs without actually having to fabricate any circuit. Automatic synthesis tools came along to translate hardware description written in VHDL into actual hardware, both in ASICs and FPGAs. The confidence built around VHDL came from two main factors. The first one was that the deterministic behavior of the simulator guaranteed consistent results. This was crucial as an IP (intellectual property) market arose for VHDL designs both from independent design houses and big foundries. The second important factor that gave confidence to VHDL was that the behavior of implemented hardware could be brought back to the simulator itself. Thus, a discrete event language with timing annotations on delays of digital gates would yield a much more precise simulation, allowing designers to correctly determine the maximum clock frequency for a given circuit among other things.

Deterministic models builds confidence, allowing us to create very complex systems. But it isn't because a language is associated with a simulator or that it is executable that it will have a deterministic behavior. Most programming languages, when executed as concurrent threads, can behave in a non-deterministic way. That's because there is another factor included in the mix: the execution environment. It will much depend on the scheduler of the processes and the work load of the machine at the time of execution to determine the sequence of events. The same thing happens with modeling languages when we start to model the environment. The uncertainty of events require a probabilistic approach that will add non-determinism to the model.

So why do we need graphical languages? Can't we solve all modeling issues with textual deterministic languages? Well, there is not a definitive answer to this question simply because we don't know all the problems yet to solve. But one thing has been proven over the years: Graphical languages tend to be more elucidative than textual languages. As we say, a picture is worth a thousand words. For that reason, graphical languages are used mostly for communication purposes, to exchange ideas, to determine system requirements and also to document complex systems with ease. The way I see it, graphical languages are following the same path that VHDL once did. It started as a documentation language, mostly for communication purposes then it became a simulation tool and finally a synthesis tool. UML and it's variants, for instance, are following the same path. They are now being used not only to determine system's requirements and structure, but also are the base for code generation for many textual languages. Some may even have an executable semantics like fUML.

2.4 Problem definition

We still have a long path to travel until graphical languages become the default simulation and design languages. There are still many unsolved problems when we try to use graphical languages to model and simulate systems where different domains are present. These, so called complex systems or heterogeneous systems, are nothing but a set of homogeneous sub-systems made to work together. Each subsystem is developed by a group of experts in a particular field, and may be modeled with different languages and formalisms. Airplanes and cars are examples of large systems that fit into that definition. They have, at the same time, mechanical, electrical and software systems working together seamlessly. What we don't usually know is that under-the-hood there are many heterogeneous interfaces that were meticulously specified so that the integration of these subsystems could happen.

There is a lot yet to learn on how to integrate different subsystems that are modeled in different languages. The most problematic issue is undoubtedly how to simulate systems that are modeled with different languages.

2.4.1 Simulation semantics

A language is defined by its syntax, semantics and semantic mapping [34]. The syntax is a set of rules that define the elements of the language, how they are represented and the combinations between these elements. The meaning of the syntactic elements is defined by the semantics which is usually related to a well-defined domain, such as

algebra for instance. Semantics is the hardest part of a language to be defined. It usually only exists in someone's head and will depend on the knowledge, creativity and ability to create semantic associations to be fully understood. Semantic mapping is the link between the syntactic elements and their corresponding meaning.

Simulation semantics differs from language semantics. The first defines how a given model should operate in a given environment and what are the repeatable required steps to calculate an output from arbitrary inputs while the second defines the meaning of the language elements and what do they represent in a given context. An example of simulation semantics is given in [48] where the formalism of Abstract State Machines (ASM) is used to describe the simulation algorithm and thus the simulation semantics of the SystemC execution kernel.

Language semantics are a bit harder to be defined. For example, the SysML standard [53] barely defines the semantics of the language. instead it focus on syntax definitions (the abstract syntax) and graphical representations (the concrete syntax). The semantic mapping is usually done by the user of the language, usually engineers when trying to specify systems requirements. The lack of semantics in SysML has both a positive and a negative aspect. The negative aspect is very clear. It is hard to work on something we don't know the exact meaning. Most of the times, engineers have to agree on a specific meaning before modeling systems requirements. This issue is by itself a big obstacle in the learning process and is probably the biggest challenge that this language has yet to overcome. The positive aspect of the lack of semantics is that we can explore this to give SysML a powerful feature not yet explored before: heterogeneous systems specification and simulation. If we provide a way to define precisely and without ambiguity the semantics of a the language elements regarding a model of computation, it would be possible to automate the generation of executable code. This is what we will mainly explore in the following chapters of this thesis. Of course, if we propose a broad number of possible semantics for the same language elements, another issue arises from this proposal. The interaction among different models with different semantics will be again source of ambiguity and will lack semantics. That is why we should provide not only the semantics for the modeling of different models of computation, but also the semantics for adaptation and communication among different models of computation.

2.4.2 Interactions among models

The problem of semantic adaptation, highlighted in works like [11], are typical of heterogeneous models. When designing heterogeneous systems one must deal with several data types and communication patterns like the ones shown in fig. 2.1. If we are not

aware of these differences and how to correctly adapt different models of computation, we might end up with erroneous simulations results and with no clue from where these errors are coming from. If homogeneous parts work perfectly, why wouldn't the integration of these different parts also work? Just like a bad contact for electrical engineers, integration issues for system engineers are the hardest to detect and track. In this thesis we propose the use of semantic adaptation techniques to avoid integration issues.

2.5 Conclusions

After taking a look at all of these different approaches, we have first selected SysML and SystemC-AMS as a promising couple to model heterogeneous systems. On one side, we have a complete set of easily understandable SysML diagrams to model multidisciplinary systems. On the other side, our backend would be the powerful and flexible SystemC-AMS language so that we can take advantage of its simulation capabilities, giving the designer a full set of tools comprising specification, architecture exploration and simulation capabilities.

As stated before, SysML is a set of diagrams without a specific meaning related to it. Therefore it cannot be considered as a methodology. Our goal is to add specific semantics to SysML diagrams in such a way that we could automatically generate an equivalent SystemC-AMS code and thereby run specialized simulations. This approach is in accordance with the OMG MDE proposal where we take the development phase to a level of abstraction where there is no dependency on the target platform.

2.5.1 Our objective

We aim at using SysML diagrams to model heterogeneous systems and automatically generate executable code either in SystemC-AMS or any multi-MoC simulation-capable language (like VHDL-AMS for instance). The problem we are trying to solve lies in the communication interface between different models, each one using different modeling paradigms. In this thesis, we will show how to achieve this by providing enough semantics to enable automatic code generation. We have used techniques from the state-of-the-art of model driven engineering such as model transformations and code generation.

2.6 Thesis outline

This thesis is organized as follows: Chapter 3 introduces tools and methods used in model transformations and code generation shown in chapter 4. These are the fundamental pieces of model driven engineering. A detailed walkthrough is shown for both transformation languages, namely ATL and ACCELEO, with simplified and didactic examples.

The main research contributions are then shown in three parts in chapter 4. The first contribution is an experiment of semantic adaptation theory applyed to the generation of SystemC-AMS code from SysML specification diagrams. The second contribution improves upon the first one by materializing parts of the semantic adaptation theory into a SysML profile. Furthermore, these two developments gave us the key ideas for a later generalization of the approach, which is described in the third part of chapter 4. An intermediary representation is shown containing the necessary elements for code generation for any textual executable language. This intermediary representation leverages model transformations by separating syntax from semantics.

We then conclude with a review of results and discussions in chapter 5. We also provide perspectives of future research that arose out of the work pursued during this thesis.

Chapter 3

Tools & Methods

3.1 Model Transformation

Models are the base of Model Driven Engineering (MDE). They belong to an considerable number of engineering processes. Software engineering for instance is based on automatic model transformations. The benefits of usign models can vary from documentation and communtication facilities to a decreased time to market delivery compared to conventional developmente techniques.

In Model Driven Engnieering, model transformations are a central operation for handling models. They allow deeper analisys of models in some cases and allow us to simplify models in other cases. Transformations can be of different nature, depending on their ability to increase or decrease the abstraction level of the model. *Refinement* transformations will take a model from a higher level of abstraction to a lower level. The inverse is called *Abstraction*. *Synthesis* transformations are defined by the creation of a new model through the combination of existing elements; the opposite of synthesis is simply a *reverse engineering*. *Approximation* transformations yield a simpler model, but contrary to *Abstraction* transformation, they do not ensure containment relations between the behavior or properties of the models. For instance, approximating real numbers with fixed point values may lead to very different behaviors because of rounding errors. *Migration* is also a kind of transformation where the level of abstraction is maintained but the language (thus the meta-model) is changed.

A transformation language must provide means to allow all those kinds of tranformations. In the following section I will detail one transformation language suited for our needs.

3.1.1 ATL

ATL stands for Atlas Transformation Language. This model transformation language was an initiative from the AtlanMod (Atlantic Modeling) team. As a joint effort from Inria, École des Mines de Nantes and LINA, ATL reflects years of research on Model Driven Engineering.

ATL is an answer to the QVT (Query, View, Transformation) request for proposal from the Object Management Group (OMG). It enables the process of converting a model to another model of the same system in an automatic fashion. It can also provide ways of manipulating models for specific needs.

ATL is now being developed and maintained by OBEO, a french company member of the Eclipse Foundation specialized in MDE techniques and tools. OBEO is also the provider of ACCELEO, a code generator based on templates.

3.1.1.1 Anatomy of the language

ATL is a declarative language. Transformation rules are written in such a way that a read-only input model is transformed into a write-only output model. Both of them conforms to their respective meta-models. Rationale: A model that respects the semantics defined by a metamodel is said to conform to this metamodel.

An ATL code is basically composed of unidirectional transformation rules. Each rule is fully described with two parts. An input element (referred by the keyword **from**) and an output element (referred by the keyword **to**) or many output elements to be created. The input element may be further filtered with conditions on properties. This first part of the transformation will only be triggered if the input element is found (and the respective conditions are met). The output elements may be further described in a third part of the rule, referred by the keyword **do**.

```
1  rule AtoB {
2    from
3      a : METAMODEL_A!ClassA
4    to
5      b : METAMODEL_B!ClassB (
6        name <- a.name,
7        type <- a.type,
8        annotation <- "created from class A")
9   ...
10 }
```

FIGURE 3.1 – Example ATL rule

The example of figure 3.1 shows a simple rule that generates a model element "*b*" from an input element "*a*" of the *METAMODEL_A*. Every element of *ClassA* found in the

read-only input model will trigger this rule to be run. The attributes *name* and *type* of the element *b* are taken from attributes of the input element *a*. A third element *annotation* will be created for every element *b*.

We can change the rule to filter input elements using an input condition. For instance, in figure 3.2, we show the same rule as before but filtered for instances of type "Adaptor".

```
1  rule AtoB {
2    from
3      a : METAMODEL_A!ClassA (a.type == "Adaptor")
4    to
5      b : METAMODEL_B!ClassB (
6        name <- a.name,
7        type <- a.type,
8        annotation <- "created from class A with type Adaptor")
9   ...
10 }
```

FIGURE 3.2 – Filtered transformation

We can also use more elaborate transformations using the **do** block. Figure 3.3 shows a for loop running over all the ports of the interface of object *a* and calling a function to create a **Port** element and attributing it to the relation *ports* of the object *b*.

```
1  rule AtoB {
2    from
3      a : METAMODEL_A!ClassA
4    to
5      b : METAMODEL_B!ClassB (
6        name <- a.name,
7        type <- a.type,
8        annotation <- "created from class A")
9    do {
10     for (port in a.interface.ports) {
11       b.ports <- thisModule.createPort(port);
12     }
13   }
14 }
```

FIGURE 3.3 – Use of the **do** block in ATL

ATL generates a model from a model, it does not generates actual code even if the metamodel represents elements of the target language. In order to run simulation we need the actual textual code. In model driven engineering, code generation is the job for a model to text generator. A tool that works well with ATL is ACCELEO.

3.1.2 ACCELEO

ACCELEO is a template-based language for generating code from a model. This language enables model-to-text generation by allowing the user to write configurable templates for each element of an input model. ACCELEO is an implementation of the MOF model-to-text language defined by the OMG [52]. ACCELEO is also a transformation

language but the target is text instead of another model. By defining templates for each component of the input meta-model, ACCELEO generates a set of files conforming to the target grammar.

ACCELEO offers a much simpler transformation then ATL does. It cannot add behavior if not specified by the input model. It can only work with the concepts of the target grammar. The input model must be written in the same language as the target textual language, thus it must conform to a metamodel that represents the target textual language. In some cases, if the input model conforms to a language that is close to the target textual language some adaptation must be done. Take for example a UML to Java code generator. UML is not supposed to be a graphical version of java. It intends to be more generic. Because of its generic expressiveness, some of the UML elements cannot be translated directly to Java. UML association-classes for example do not have an equivalent on the Java side. They can be translated to other java concepts that may imitate the expected functionality of UML's association classes, such as a full class and two associations.

3.1.2.1 A template-based language

A template written in the ACCELEO language is very close to the target language. Everything between brackets will be replaced by what is in the model. An example is shown in figure 3.4. Here we show the generation of a systemc header file declaring a module and its ports. In this example, for every element "*Module*" found in the input model will trigger the creation of a file with the name of the module concatenated with the extension ".hpp". Inside it, a module is declared with the macro "SC_MODULE" and we use a for loop to iterate over all ports and declare them accordingly, with its type, direction and name. Note that ACCELEO will replace only the code inside the brackets except for internal commands such as **template**, **file**, or **for** loops.

```
1  [template public genHeader(a : Module)]
2  [file (a.name.concat('.hpp'), false, 'UTF-8')]
3  ...
4  SC_MODULE([a.name/])
5  {
6    ...
7    [for (p : Port | a.ports)]
8      sc_[p.direction/]<[p.type/]> [p.name/];
9    [/for]
10   ...
11
12 [/file]
13 [/template]
```

FIGURE 3.4 – Example Acceleo template

3.1.3 Purpose of model transformations

It is clear that ACCELEO can only generate skeleton code, i.e. only structural elements from the model. It cannot go beyond that, like for instance generate the body of a complex functions or infer an equation from a dataflow diagram. That's because UML's and SysML's generic expressiveness does not cover functional semantics. With those languages, we can go as far as describing a state machine or generic algorithms in activity diagrams. Complex behavior like data flow charts or differential equations are clearly a limitation of the language. But then again, this is not something these languages were designed to do.

Despite that limitation, there are some artifacts that we can play with in order to tackle this issue. Constraint elements have light semantics or almost none. They are generic elements that may be used for many purposes. One of them is to impose a constraint on any other element. The word constraint means limitation or restriction. When used in a model, it can impose a behavior to an element or a group of elements in a same set. These elements will share the same behavior and may communicate in a standard way.

We can use this to adjust, specify or modify the generated code for a given object. When ATL encounters a specific constraint, for example a constraint that sets the model of computation, it can create a model that reflects the specified constraint. We will show in details how to do this in the following chapter.

Chapter 4

Contributions

In this chapter I'll describe the theory behind semantic adaptations and its implementation in the form of model transformations, which are the main contribution of my thesis work.

4.1 Problem of mixed interfaces

As stated before, the issue of systems integration becomes utterly difficult when there are different modeling paradigms in the interface of two sub-systems. It appears that, in the domain of systems simulation, even thought each engineering field is well-known and well-described by its domain specific languages, the integration of domain-specific components into a system is still a challenge on it's own. It will depend on the set of languages used. In our research group, we believe that the cause for all of these issues comes from incomplete interfaces definition. More specifically the semantic adaptation is usually not specified.

It is not evident to think about semantic adaptation when the semantics of languages is not clearly defined. In simulation-capable languages, the execution semantics comes from trial and experimentation of the language. Most users of simulation engines do not think about the inner details of the simulation kernel and that is not a reproachable behavior. One must focus on the model in order to solve engineering problems and should not focus on the details of simulation. Unfortunately the devil hides in the details and thus, questions like : "How will a block communicate and exchange data with its neighbors?" or "When will this block be activated?" are now unavoidable when trying to integrate systems together.

It is also common to see a misconception between semantics and behavior. Many believe that to understand the meaning of a model, one should know how it runs. This is not true, even though knowing how the model runs helps getting familiar with the executional semantics of the underlying simulation engine. In systems modeling, both the systems behavior and its structure are important views of models. Both are represented by syntactic concepts and both need semantics. [34]

Some industrial initiatives show how urgent this problem is. The definition of IP-XACT [1] in the form of an IEEE standard for instance is one attempt to motivate better interface definitions so that different IP's from different companies can communicate better. That may sound interesting, but it doesn't solve the inner issue of semantic incompatibilities of different formalisms.

4.2 Simulation : Continuous vs Discrete

A simple example of this issue is the simulation of systems that mix continuous formalisms with discrete ones. Any system that can be modeled by differential equations can be seen as continuous devices. Mechanical parts and motion can be described using Newton's laws. Analog circuits can be modeled using Kirchhoff laws. Many complex systems can be modeled using state space equations. All of these systems are examples of continuous-time models that are represented by equations that are continuous in time. Even thought the simulator may use sampled time (periodically or not), the models of these systems are continuous by principle.

Discrete models try to capture the essential parts of a communication process, usually leaving the continuous aspect of time aside. Events may happen with time tags or not. In Synchronous Data Flows, for example, calculations are made by steps. This formalism is independent of the notion of time. Each step represents a calculation made by a processing unit. These calculations may not take the same time amount to be processed. With this formalism, we are not so interested in the exact amount of time a process will take. We are more interested in the order of execution of the processes and what are the possible schedules for the execution of this model.

When you combine these two formalisms to model heterogeneous systems, such as a digital-to-analog converter or a micro-electromechanical sensor, modeling problems may appear simply from the integration of these formalisms. Some simulation questions naturally arise from this union.

- How will continuous data be converted to discrete and vice-versa?

  - Will there be interpolation?
  - If yes, what algorithm should we use to interpolate?

- What should be the semantics of the absence of data?
  - First-order hold?

- How should we synchronize data from these two formalisms?
  - Is there a common representation of time?
  - Or should we allow different time notions?

- When should we activate modules in the interface of these two formalisms?
  - Should we handle Zeno effect?

These are the kind of questions that motivated this work. Without saying it explicitly, I've introduced three different axes of semantic adaptation theory : adaptation of data, time and control. I'll explain and detail these concepts in the following section.

4.3 Semantic adaptation

Semantic adaptation deals with communication patterns of different formalisms and how we glue them together for interoperability. Different models may use different notions of time and data. Let me first introduce the reader to the problem. Consider a model of an analog signal being read by a digital system for further analysis. Depending on the level of detail that we wish to model we can either have a very detailed model with all the physics of the system modeled at the gate level, which will render the simulation extremely precise but heavy, or we can abstract away a few concepts, like voltage level and frequency response of the transistors (as long as we don't extrapolate certain limits such as the clock speed) wich will leverage by a considerable factor the simulation time of this system.

With a higher abstraction level, some considerations must be taken into account when simulating analog components together with digital circuitry. In a real world implementation, an analog-to-digital converter would definitely be applied in this situation but in a simulation environment at a higher level of abstraction it would be replaced by an adaptor logic that will convert analog data into understandable digital events. Even if this is a fairly simple example a few considerations must be taken into account.

We have to determine :

- The numerical representation of the data in the digital world: fixed point, float, how many bits and saturation levels
- The way data will be sampled: periodically or dynamically. And what will be the time step considered. In the case of a dynamical time step, the minimal and maximal time step.
- Will there be a delay between data acquisition and the moment the data is available to the digital domain? and of how much?

All of these questions are part of the adaptation semantics theory and are only needed because we have integrated these two different formalisms together. The first item is part of one axis of the semantic adaptation theory: Data adaptation. The second and third item are part of the second axis : Time adaptation. We have not yet introduced the third axis wich is control adaptation i.e. when should components of an heterogeneous model be activated.

Regarding SystemC and SystemC-AMS, one particular work has drawn the community's attention with regard to the issue of semantic adaptation: HetSC [36]. This work was mainly motivated by the need of heterogeneous systems specification specifically in SystemC. The authors provide a library with SystemC macros, channels and interfaces for the integration of several models of computation. Some of them are CSP, SDF, SR, KPN and PN. For example, for a KPN MoC, they provide an infinite FIFO for KPN processes to communicate among themselves. Heterogeneous interactions are provided by interface processes (in the SystemC meaning) and interface channels. In this work, the semantic adaptation is hidden away from the user by pre-coded modules. Semantic adaptation is reduced to searching the appropriate interface and plugging it on your module. For this to work perfectly, one must follow coding guidelines provided by the HetSC library. Almost like with a Lego, if a block doesn't fit, it shouldn't be used.

In this research though, I defend the explicit use of adaptation semantics mainly because most of the issues of system's integration come from the lack of knowledge of subsystems interfaces. Explicit definitions of how a system should adapt to inputs from another model of computation would make it clear and evident to the user what is happening under-the-hood of the simulator engine. It would help the understanding of systems integration. Semantic adaptation should not be hidden away from the user, instead it should be part of the system's specification. In some ways it could be the base definition of a future block implementation. For instance, digital-to-analog and analog-to-digital adaptors are a clear example of this. When we define the semantic adaptation from the digital world to/from the analog world we are almost fully describing an AD/DA module. From this semantical adaptation specification we can, for instance, automatically

synthesize a AD/DA module. If a full synthesis is not possible, some choices could be offered to the modeler of the system.

4.4 Implementation

In the following section, I will describe the implementation details for each language to which we applied the theory of semantic adaptation. I made a total of three important implementations that I'll describe here. All of them focus on how to embed semantic adaptation concepts into SysML and how we generate executable code from these annotated diagrams. As said before, the way we define semantics of a given language is by describing a mapping from the language abstract syntax to a chosen semantic domain. In the following works, I will use extensively model driven techniques such as model-to-model and model-to-text transformations. It is interesting to note here that the model-to-model transformation implements not only a syntax-to-syntax transformation but also the semantic adaptations we cited before. These will map a non-existent concepts from SysML to a semantic domain of one of the target languages. So, without further delay, let's dive into these implementations:

4.5 SysML to SystemC-AMS Transformation

This first work targets two industrial standards for heterogeneous systems specification, modeling and simulation: SysML and SystemC-AMS. SysML provides a graphical way to model structure and behavior. Despite its flexibility, SysML lacks semantics to give language elements a precise meaning. Current implementations of the standard allow multiple interpretations of syntactical elements and can cause misunderstandings when porting a model among tools. This work focuses on the definition of concrete semantics for SysML to enable correct interpretation of heterogeneous models. We also add semantic adaptation elements to guarantee that interactions among different formalisms are unambiguous. We demonstrate this approach by generating SystemC-AMS code automatically from SysML diagrams for a case study with two distinct formalisms. This kind of translation allows the validation of systems behavior through simulation.

4.5.1 Introduction

Component-based design [46] is today's standard way to design systems. It consist of breaking down the system in a set of objects composed hierarchically with interactions restricted to tightly-defined interfaces. Heterogeneous systems require extra modeling

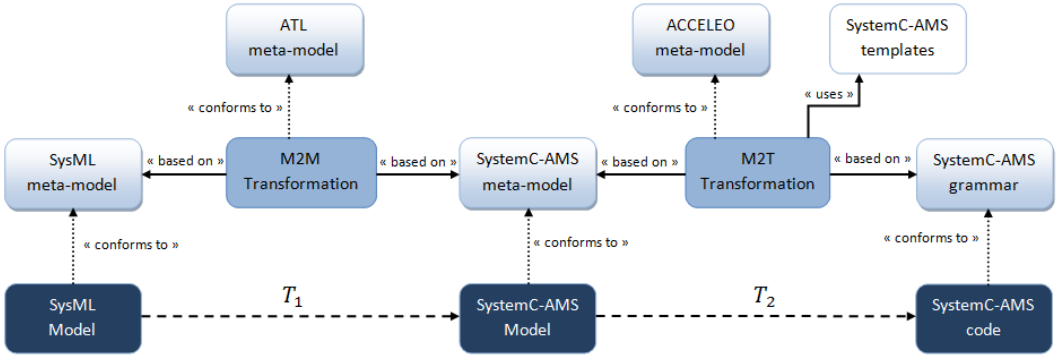

FIGURE 4.1 – Transformation chain

effort because the interactions must be well defined when crossing the boundaries of different domains. Even if the interfaces are completely specified, a problem remains: the execution semantics of components of different domains may differ and without a formal definition of how these components should interact, it is most likely that the overall system, once integrated, behaves unpredictably or in an implementation-dependent way.

During the design phase of a system, it is rather common to use simulations of models as a support to validate correct functionality of a component before deploying it to fabrication. Those models can be either described in textual (like Verilog, VHDL, SystemC) or graphical languages (such as UML, SysML). The latter is generally accepted to be easier to use and more comprehensible.

The present work tackles the problem of simulating heterogeneous systems described in SysML by adding concrete semantics to SysML diagrams through the use of MoCs. We exemplify our approach with a system composed of two components best modeled with two distinct MoCs, i.e. FSM and CT, both embedded in a DE simulation environment. We also define explicit rules that we call *semantic adaptation* for the interactions between these MoCs and we automatically generate SystemC-AMS code using model transformations based on our semantic definitions. We thereby take advantage of the SysML language as our front-end modeling environment and simultaneously benefit from SystemC-AMS powerful simulation engine to validate heterogeneous systems by simulation. In the following section I will detail our approach by showing an automated way to generate SystemC-AMS code from annotated SysML diagrams.

4.5.2 The approach

Models are an abstraction of a part of the reality around us. We can model the same thing in different ways in order to emphasize some aspects of this reality. Consider for example a sound signal. We can visualize it in the time domain to see the time variation

and evaluate properties such as zero-crossing and maximum amplitude but if we look at it in the frequency domain, we can verify other properties that were hardly visible in the time domain, such as harmonics and bandwidth. To go from one domain to the other, we must apply a transform operation which corresponds to a set of rules and calculations allowing the conversion from one domain to the other.

In Model Driven Engineering, a transformation is *a set of rules capable of generating a target model from an input model (or a set of input models) in an automated way.* It can be considered as *an engine capable of translating a model among different abstraction layers or different languages.* Transformations can be of different nature and are classified depending on their ability to increase or decrease the abstraction level of a given model. For example, *synthesis* transformations are defined by the creation of a new model through the combination of existing elements thus decreasing the abstraction level of the model. In the opposite way, we have the *abstraction* transformation. *Approximation* transformations yield a simpler model, but contrary to *Abstractions*, they do not ensure equal behavior. For instance, approximating real numbers with fixed point values may lead to very different behaviors for some models because of rounding errors. *Migration* is also a kind of transformation where the level of abstraction is maintained but the language (thus the meta-model) is changed.

Our approach, illustrated in figure 4.1 consists in two separated phases. Starting from the SysML model, we first do a model-to-model (M2M) transformation in order to have an equivalent model in the SystemC-AMS language. We then generate SystemC-AMS code through a model-to-text (M2T) transformation using templates of SystemC-AMS.

The M2M transformation takes into consideration the constructions of the input and output languages, thus their meta-models. Since models conform to their meta-models, the transformation can be applied to any instance of the input meta-model. This step is responsible for the translation of every SysML element into its equivalent SystemC-AMS. For example, applying the transformation T_1 of figure 4.1 to a SysML Block composed of several parts results in the creation of a SystemC-AMS module with its corresponding sub-modules.

We have used the ATL [40] to define the M2M transformation (i.e. from SysML models to SystemC-AMS). ATL is a language to define model transformations by a set of transformation rules. Being a model itself, the transformation has its own meta-model as well. ATL is based on pattern recognition of *from/to* rules. Every element of the source model that matches any *from* rule triggers the creation of the corresponding *to* element. Therefore, for every SysML element, we have an equivalent SystemC-AMS element.

The example in figure 4.2 shows a simple rule that will generate a SystemC port (`sc_port`) for every SysML flow port (`sml_port`). This rule will read all attributes from the SysML element, such as name, type and direction, and associate to the equivalent element on the SystemC side.

```
1  rule Ports {
2    from
3      sml_port : SYSML!FlowPort
4    to
5      sc_port : SC!Port (
6        name <- sml_port.base_Port.name,
7        type <- sml_port.base_Port.type.name,
8        direction <- sml_port.direction)
9   ...
10 }
```

FIGURE 4.2 – ATL rule: port from SysML to SystemC

In order to define the M2M transformation both input and output meta-models should be available. SysML's meta-model has been defined by the OMG and it is now an established standard [53] providing a structured definition of the languages elements. Every model written in SysML must conform to its meta-model. SystemC-AMS, on the other hand, has no comparable meta-model reported. Nevertheless, since it is also a language with specific constructions we can define its own meta-model.

SystemC meta-models have been studied in [58] and [9] but were limited to the DE MoC. Based on these previous works, we have added AMS specific constructions and facilities to support multi-formalisms and semantic adaptations. A simplified version with most important elements is shown in figure 4.3.

Differently from [58], we have separated the definition of an atomic module from that of a composed module allowing us to differentiate a user-defined component from a library standard component e.g. integrators from LSF formalism or resistances from ELN. A composed module may contain ports, variables signals and other sub-modules. The

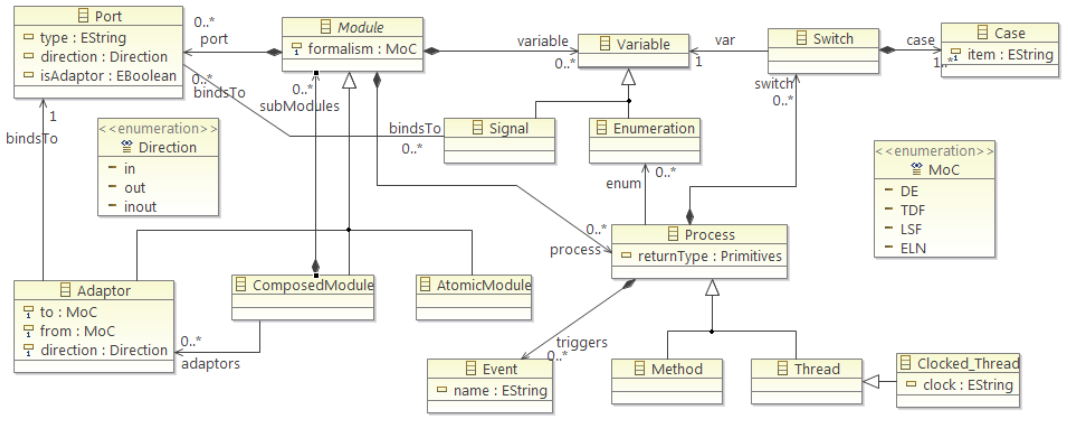

FIGURE 4.3 – SystemC-AMS simplified metamodel

latter model hierarchical compositions. Some other constructions were inherited from basic C/C++ language, such as the case-switch, necessary to implement a state machine in SystemC.

A module may be modeled using one of the four possible formalisms, either using the standard SystemC DE MoC or with any of the specific AMS MoCs, i.e. LSF, ELN or TDF. We take that into account through the use of the *formalism* attribute in the *Module* abstract element. In order to consider the semantic adaptation when crossing the boundaries of different MoCs, we have added an *Adaptor* element that binds to a port.

Adaptors are elements responsible for translating signals from one domain to another. The behavior of an adaptor depends on the combination of input/output MoCs and whether it is a producer or consumer of data. We capture those properties in the meta-model by the three attributes *to*, *from* and *direction*. The generated SystemC-AMS code corresponds to the standard adaptor channels available from the SystemC-AMS 1.0 proof-of-concept released by the Fraunhofer Institute for Integrated Circuits IIS.

Code generation is the second step of our approach. As shown if figure 4.1 T_2 is a model-to-text (M2T) transformation capable of generating C++ code from SystemC-AMS models. To define the M2T transformation, we use ACCELEO [49] which is an implementation of the MOF model-to-text language [52] defined by the OMG. ACCELEO is also a transformation language but the target is text instead of another model. By defining templates for each component of the input meta-model, ACCELEO generates a set of files conforming to the target grammar.

In our approach, T_2 scans the input SystemC-AMS model and generates two files for every block, one header with the module definition (equivalent to the black box), and one source file with the implementation of every process.

```
1  [template public genHeader(m : ComposedModule)]
2  [file (m.name.concat('.hpp'), false, 'UTF-8')]
3  ...
4  SC_MODULE([m.name/])
5  {
6    ...
7    [for (p : Port | m.port)]
8      sc_[p.direction/]<[p.type/]> [p.name/];
9    [/for]
10   ...
11
12 [/file]
13 [/template]
```

FIGURE 4.4 – Header generation with Acceleo

In the example of figure 4.4 we show the creation of each header file when a Composed-Module is found. We define a ComposedModule as a hierarchical element containing

other modules. In this example, we generate SystemC code for the Module's black box, thus we shall declare every port inside a `SC_MODULE` macro. We do that with a **for** loop that iterates over the sequence of ports of the *Module* 'm' and writes equivalent SystemC code. Note that ACCELEO will replace only the code inside the brackets except for internal commands such as **template**, **file**, or **for** loops.

Although code generation is necessary for running simulations, we focus our work on defining concrete semantics to SysML models. We use semantic definitions with the help of SysML constraint blocks. The stereotype "constraintBlock" or simply "constraint" describes constraints on system structures [65]. SysML does not define one language to express constraints. Most will use regular arithmetic expressions to describe relations that can be automatically evaluated by a third party tool. We have chosen to use specific keywords (as we shall demonstrate later in a case study) to indicate directly in the diagram which MoC is used for each SysML Block.

Our approach for filling the semantic gap in SysML is to define concrete semantics of each MoC along the three dimensions of concurrency, communication and time. We also consider the heterogeneity of multi-paradigm systems and the necessary semantic adaptations at the frontier of different domains. These semantic definitions are implemented by our transformations together with the necessary adaptations. In the following section, we introduce the semantics of two MoCs, i.e. CT and FSM so that simulation of SysML diagrams are free from ambiguous definitions. We also describe briefly the simulation engine of SystemC.

4.5.3 A multi-paradigm semantics

4.5.3.1 The simulation engine

The execution model is based on the delta-cycle simulation algorithm defined by SystemC's discrete event engine [48]. At the very heart of its engine, the main algorithm is composed of three steps: Evaluate, Update and Time Advancing (also called delta notification). In the evaluation phase, SystemC will run every process but will not propagate data to corresponding signals or ports until every process is executed. The update phase will then synchronize all processes by updating signal and ports with previously calculated values in the evaluation phase. The update phase may generate instantaneous events. This may trigger the engine to re-evaluate some of the processes without advancing the simulation time. Finally, when the system's state is stable, time advances until the next scheduled event. This ensures that every node is evaluated before data can propagate and guarantees the concurrency of elementary blocks. Concrete semantics is given individually for each MoC.

4.5.3.2 Continuous Time Semantics

Continuous-time models can be expressed using block diagrams. The use of SysML's *internal block diagram* is suitable to represent hierarchical composition of elements of a system. The CT formalism requires the use of pre-defined building blocks, such as subtractors, integrators and gain blocks. These primitive blocks are defined in a separated library, shown in figure 4.5 and are used by the designer to model dedicated transfer functions. The use of the CT formalism is expressed by a SysML constraint block *CT Block* as shown in the example of figure 4.7 (for simplicity, only a subset is shown).

Concurrency is necessary in the continuous-time formalism because the composition of blocks aims at the definition of complex differential equations. Therefore, since blocks connected together belong to the same equation, they should be evaluated concurrently. CT blocks are defined by an equation describing how outputs react to their inputs variations. A CT block shall apply a mathematical function to its input variables every time there is a new sample available on one of its inputs. These mathematical relations can be defined by SysML constraints, as shown for CT building blocks library in figure 4.5

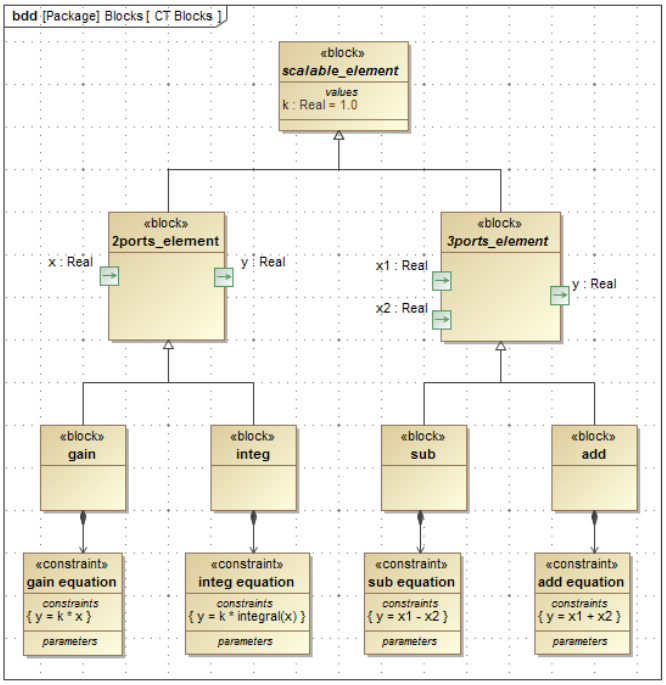

FIGURE 4.5 – Continuous Time Building Blocks

Communication is defined by the interpretation of what connectors do. In the case of a CT block, connectors are interpreted as variables of a differential equation. They act as the system memory, saving the state of that system for every snapshot in time.

Time is the independent variable on which some CT blocks rely to apply their mathematical relations. For instance, the gain block has no state and does not depend on time, but the integrator block requires time variations to apply its transfer function.

4.5.3.3 Finite State Machine Semantics

Finite State Machines have a dedicated diagram in SysML. States are represented by rounded corner rectangles and transitions by arrows. The transition guard is a condition or an event required to change from one state to another. The state invariant represents the control. It takes the form of an equation placed inside the states and produces an output whenever that state is reached. For the purposes of our work, we consider FSMs to be untimed.

Concurrency is defined by regions where independent states run concurrently. The most common kind of construction is the or-state, where no concurrency is defined and the system state is defined by the current state itself. A less regular construction is the and-state set. In this case, the system state is defined by a subset of states of independent regions.

Communication is nonexistent. There is no data flow in a state machine. This kind of diagram is used exclusively to model control.

Time: The notion of time does not exist in an untimed finite state machine. This formalism is driven only by events which do not require a time scale. Semantic adaptation is needed when continuous-time variables are connected to a state machine. In this case, a monitor shall be created for each guard condition to detect threshold crossing and trigger events which are responsible for state changes.

4.5.3.4 Semantic Adaptation

In order to have precise simulations, one has to define not only the semantics of each formalism but also the necessary actions and adaptations if different formalisms are used in the same diagram. This can be achieved by the definition of an adaptor element.

The adaptor is an entity that is bound to a port in order to explicitly adapt data, control and/or time for different formalisms. Our transformation chain chooses appropriate adaptors from the standard SystemC-AMS library depending on the frontier the port is on. For example, using the LSF formalism inside a continuous-time block and the outside environment is of discrete event nature, then a LSF to DE source or

sink (`sca_lsf::sca_de::sca_source` or `sca_lsf::sca_de::sca_sink`) should be chosen, depending on the direction of the port.

Some adaptors require the definition of specific attributes. Input ports from DE to CT require the definition of a sampling time-step to guarantee that analog data will be available periodically. We illustrate the use of multi formalisms and adaptors in the following case study.

4.5.4 Case Study

4.5.4.1 The model

Consider the following example: a vehicle with speed control. This system can be modeled by two blocks: one to model the dynamics of the vehicle and another to model the speed control. We represent the dynamics of the vehicle using an internal block diagram that models the differential equations of the state variables of the system, such as acceleration, speed and distance. The control block, on the other hand can be best modeled using state machines. Those are two different formalisms with different semantics. In figure 4.6 we show the vehicle composed of one part *i_ dynamics* typed by the **Dynamics** block and one part *i_ control* typed by the **Control** block.

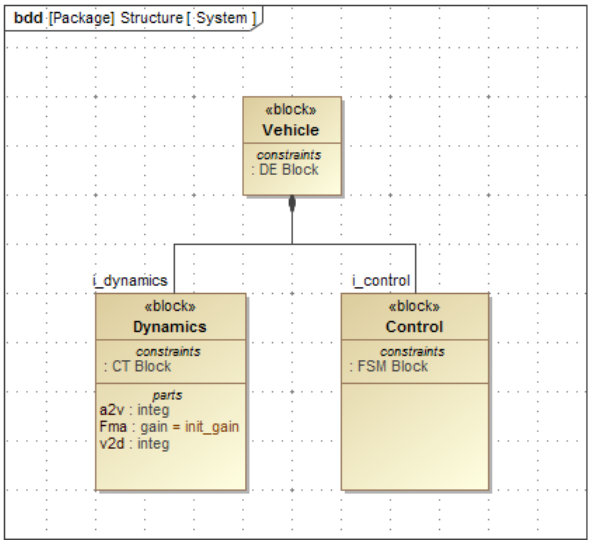

FIGURE 4.6 – Vehicle composition

The dynamics block is composed of two integrators and one gain block. They appear as parts of the dynamics block. Note that some blocks have parts that should be initialized with a proper value. In the diagram of figure 4.6, *init_ gain* is one instance of type *gain* with initialized parameters. Other parts will assume default values as defined by their types.

Figure 4.7 shows the vehicle's dynamics modeled by an internal block diagram with the gain block applying the equation $F = ma$ and two integrators that will compute the speed and distance from the acceleration.

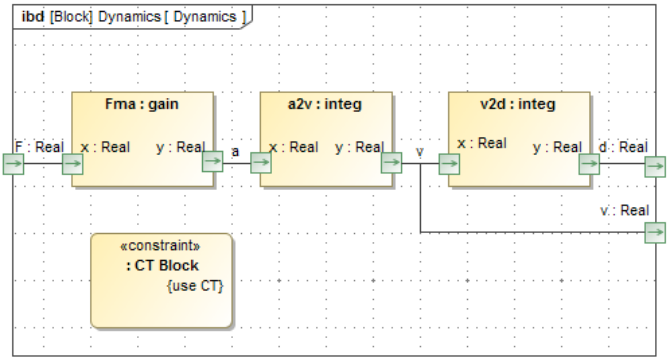

FIGURE 4.7 – Vehicle dynamics

To solve the semantic gap of the internal block diagram, we have added the constraint block *CT Block* with the keyword *useCT*. This implies that semantics defined in section 4.5.3 should be applied to this diagram. Thus the gain block and both integrators shall apply the mathematical relation defined in figure 4.5.

The control block is responsible for applying a certain amount of force to the dynamics block, depending on the state of the vehicle. We have modeled it with a State Machine Diagram as we can see in figure 4.8. The goal is to make the vehicle reach a certain speed, maintain it for a given distance and then stop.

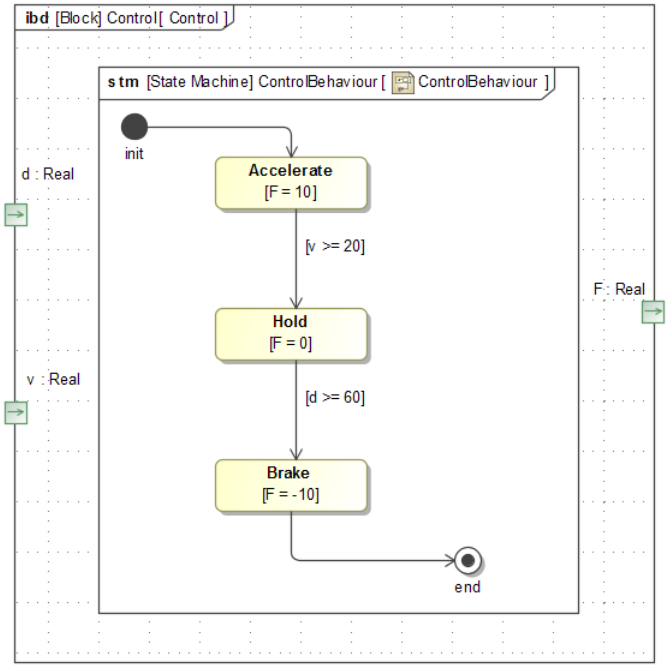

FIGURE 4.8 – Vehicle control

Note that inputs are continuous variables, but inside the control block, we only have events or conditions declared. In this case the semantics of the FSM MoC, as defined in section 4.5.3, is used since we have the constraint *FSM Block* applied to the block Control.

The adaptation is shown in the top-level block, i.e. the internal block diagram of the block Vehicle. In order to define how the state machine interprets analog data and with what precision the inputs are monitored we explicitly annotate in the diagrams, in the form of comments, that ports are bounded to adaptors using the keyword *isAdaptor* as shown in figure 4.9.

The declaration of an adaptor is made in the following form: **isAdaptor**(*adaptor type*), where *adaptor type* is a code for the multi-domain frontier to which the port belongs. In our example, we consider the vehicle to be embedded in the discrete event simulation environment of SystemC. Input port 'F' is in the frontier of a DE-CT environment. It shall then apply the adaptor type *de2ct*.

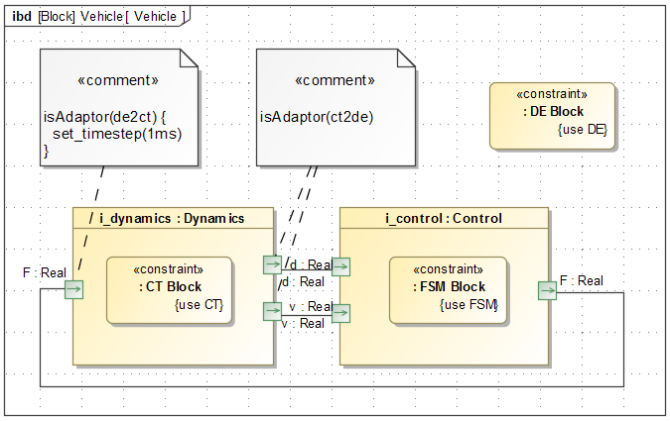

FIGURE 4.9 – Vehicle Composed of the dynamics and control

This special case of adaptor will adjust the time scale for the CT block because in the DE environment, data won't be present at all times. In the example, we chose to use periodic sampling by setting the corresponding attribute *timestep* to 1ms. This will create a sample every 1ms at the input port 'F' required by the CT block to calculate the outputs 'v' and 'd', corresponding to speed and distance respectively.

Outputs 'v' and 'd' apply the inverse adaptor *ct2de*. Contrary to *de2ct* this adaptor will convert data instead of adjusting time. It shall generate an event interpretable by the DE simulator every time a sample is available allowing the FSM to detect with a determined precision (in this case the simulation time step) when events shall trigger its internal guards. The adaptation is a design choice, and the use of adaptors makes it explicit so that different tools can interpret the model in the same way.

4.5.5 Results

From our transformations engine, we obtain plain executable SystemC-AMS code. The CT block was successfully translated to its equivalent LSF model in SystemC-AMS, using base blocks with the same transfer function as defined by the constraints of figure 4.5. The Finite State Machine was automatically mapped into a two process module with variables `current_state` and `next_state` implementing the classical representation of state machines in SystemC.

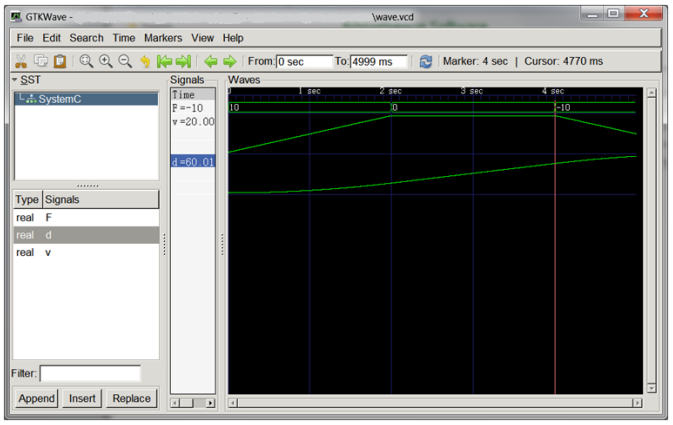

FIGURE 4.10 – Results obtained from an automatic code generation

In figure 4.10, we show the output of the simulation obtained by compiling and running the generated SystemC-AMS code. We can see the force applied to the dynamics model in the first row. It is a signal of discrete nature correctly adapted to work with the continuous time signals 'v' and 'd'. The vehicle accelerates until it reaches a constant speed of 20m/s as specified in the control block of figure 4.8. After that, control will switch to *hold* state keeping the speed constant until the vehicle reaches 60m slightly after 4 seconds. It finally switches to *brake* state until the end of simulation.

The remarkable aspect of this simulation is that using only the SysML diagrams we were able to generate the complete executable SystemC-AMS model. Semantics were defined individually for each MoC. Interactions in multi-domain frontiers were strictly described by the adaptors. This approach could be extended to other languages if the meta-model of the target language is available. Identical simulation results would be obtained since the behavior is strictly determined.

4.5.6 Discussion

This two-step technique is a first approach toward a generic intermediary meta-model from which we could automatically generate code for other languages, e.g. VHDL-AMS. With some minor changes, our approach could be extended to support other MoCs. In

this case, the user only needs to complete the framework with templates of constructions proper to the MoC of interest. If necessary, other languages could be used as well. For example, if the user intends to use Communication Sequential Processes (CSP), HetSC could be a possible candidate. This case results in more changes in the framework such as augmenting the SystemC-AMS meta-model with specific HetSC elements and adding corresponding templates to match HetSC grammar.

The approach has its drawbacks as we try to be as generic as possible. One could claim that since SystemC-AMS provides facilities to model continuous time systems we could benefit from the specific MoCs by defining the use of LSF or ELN directly in SysML diagrams instead of using a generic MoC CT and then translate into equivalent LSF or ELN modules. This could facilitate the approach by having only the code generation step. Again, the choice of the two-step technique allows us to build a generic framework to target the generation of code for other languages.

4.5.7 Partial Conclusions

This work introduced a premature approach for simulating multi-domain systems modeled in SysML. It was published in IEEE's Forum on Design Languages. Here, we validate the behavior through simulation and we target only SystemC-AMS as our execution engine. We address the ambiguity problem of SysML diagrams by assigning concrete semantics (MoCs) to SysML diagrams. In order to solve the semantic adaptation problem, we added the notion of adaptors to SysML based on the existing SystemC-AMS converter channels.

Based on model-driven engineering, our transformation framework is capable of generating executable SystemC-AMS code from multi-paradigm SysML diagrams. The main contribution of this work was to extend SysML to SystemC code generators by adding:

(a) concrete semantics to SysML syntactical elements,

(b) support for the AMS extension and

(c) simulation capabilities to SysML models.

This work can be extended to other formalisms and can be improved with the specification of test-benches and use cases. These would require significant work on the metamodel definition to add concepts of specific formalism. Significant work would have to be done also in both transformations in order to embrace new formalisms.

There are other ways to explore this work beyond of what we have proposed. Semantic verification techniques in our transformation engine is a possible branch. This could provide verification of SysML models not only for the syntax, but also check if the model presents coherent semantics.

Because of the positive feedback we had from the community, we have done a similar work targeting another simulation language: VHDL-AMS. The following section describes it in details. It will show significant improvements regarding this one. We will also exemplify it with another case study, more focused on continuous to discrete interactions. Without further delay, let's jump right into it.

4.6 SysML to VHDL-AMS Transformation

This contribution focuses on the simulation of heterogeneous systems modeled in SysML, in particular, systems that mix different engineering domains such as mechanics, analog and digital circuits. Because of their nature, expressing multi-paradigm behavior in heterogeneous systems is a cumbersome endeavor. SysML does not provide a standard method for defining the operational semantics of individual blocks nor any intrinsic adaptation mechanism when coupling blocks of different domains. We present here a way to address these obstacles. We give well-defined operational semantics to SysML blocks by using profile extensions, together with a language for the description of adaptors. We apply our approach to a test case, using a toolset for SysML to VHDL-AMS transformation, capable of automated generation of VHDL-AMS code for system verification by simulation.

4.6.1 Introduction

In the Electronic Design Automation (EDA) industry, the need for modeling and verification of mixed-signal systems gave rise to several system design languages supporting Analog and Mixed Signal (AMS) extensions. Some examples are VHDL-AMS [17] and SystemC-AMS [28]. These extensions support the use of different models of computation concurrently in a single design thus enabling the modeling of heterogeneous systems. As complexity increases, these textual languages are no longer suitable for proper documentation and communication among different teams. For these use cases, graphical languages are preferable, and they play well with Model Driven Engineering workflows.

SysML, the Systems Modeling Language, is an industry standard for systems specification. It provides a large set of diagrams which can be used to specify system's requirements, model their behavior or even detail the interconnections of structural blocks.

Despite its flexibility, SysML does not provide clear semantics. On the one hand, this can be helpful for engineers wishing to describe systems in an early development phase, especially when some implementation details are not yet entirely defined. In this case, SysML is a helpful communication tool. On the other hand, the lack of clear semantics can be cumbersome if one wants to run simulations from the SysML diagrams.

For the purpose of solving the lack of semantics of SysML diagrams, we have developed a technique to generate executable code from SysML models which is based on two foundations:

(a) Explicitly state the semantics of modeling elements, and

(b) Define the semantic adaptations between heterogeneous models.

The focus of this work is the creation of an adaptor instantiation language for semantic adaptation for specifying interfaces precisely and without ambiguity.

Our previous contribution targeted SystemC-AMS simulation language. This following work is a follow-up that introduces a new adaptor instantiation language and MoC definition mechanisms that are better suited for model driven engineering.

We use a custom SysML profile to extend the semantics of SysML blocks for continuous-time and discrete-event blocks. These two domains are generalized into two stereotypes $\ll$ *analog* $\gg$ and $\ll$ *digital* $\gg$. A third stereotype is dedicated to the description of $\ll$ *adaptor* $\gg$ blocks. Those provide explicit behavior on how to adapt data, time and/or control.

In conjunction to the SysML profile, a mini-DSL was designed to allow the instantiation of off-the-shelf types of adapters. Depending on the target language these could either be present in standard libraries or custom designed. This is also an opportunity to show that our previously developed technique apply to other target languages as well, namely VHDL-AMS.

4.6.2 A case study of a MEMS Accelerometer

Micro Electro Mechanical Systems (MEMS) motion-sensing devices are a good example of heterogeneous systems that mix mechanical, analog and digital components in the same system. They can be used to measure a variety of physical quantities such as acceleration, pressure, force, or chemical concentrations. To make such measurements, MEMS sensors can take advantage of several transduction mechanisms, for example, piezoresistive or capacitive sensing. Here we build a simple model of a capacitive sensing accelerometer to illustrate our proposal.

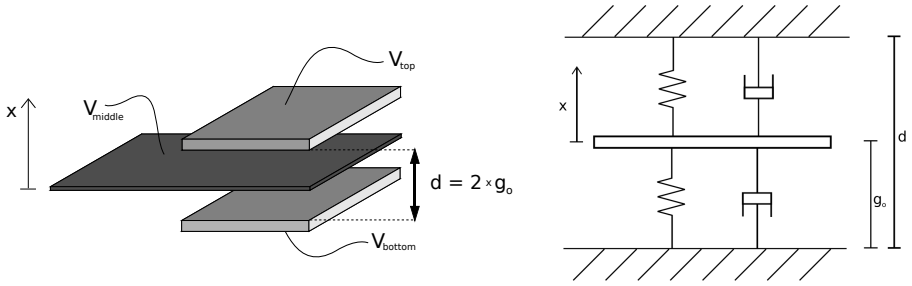

FIGURE 4.11 – Electrical vs Mechanical Model

4.6.2.1 Description of the system

Our case study is a capacitive sensing accelerometer composed of two electrodes and an intermediary membrane free to move only in the vertical axis as illustrated in figure 4.11. This structure forms two capacitors between the middle membrane and both the top and bottom walls. Top and bottom walls are attached to electrodes. The vertical movement of the membrane implies the variation of both capacitances since $C \propto 1/(g_0 \pm x)$, where g_0 is the gap distance at rest and x is the displacement of the membrane from rest. One can either connect the membrane to ground hence fixing the middle voltage V_{middle} to zero or one can leave it disconnected thus fixing the current to zero. In the first case, the change in stored charge caused by the displacement of the membrane leads to a current flow. In the second case, since the middle electrode is disconnected, there is no current flow, and by charge conservation the voltage across the membrane must change with the displacement.

Using the second method, we can obtain a linear relation between the membrane's voltage and its displacement provided that we apply a symmetric voltage on both top and bottom electrodes (i.e. $V_{top} = -V_{bottom} = V_0$) as explained in [16]:

$$Q_{top} + Q_{bottom} = 0$$

$$C_{top}(V_{top} - V_{middle}) + C_{bottom}(V_{bottom} - V_{middle}) = 0$$

Here, Q_{top} and Q_{bottom} are the charges of top and bottom capacitors, respectively. C_{top} and C_{bottom} are the corresponding capacitances. V_{top}, V_{middle} and V_{bottom} are the ground-referenced voltages of the three terminals: top, middle, and bottom. Re-arranging for V_{middle}, we obtain:

$$V_{middle} = \frac{C_{top}V_{top} + C_{bottom}V_{bottom}}{C_{top} + C_{bottom}}$$

One can simplify this relation by choosing to set $V_{top} = -V_{bottom} = V_0$, i.e if we apply a symmetric voltage on both top and bottom electrodes, as explained previously.

$$V_{middle} = V_0 \frac{C_{top} - C_{bottom}}{C_{top} + C_{bottom}} \tag{4.1}$$

We recall the capacitance definition in order to obtain an equation relating the membrane's displacement and voltage : $C = \epsilon S/d$, where S is the plate's surface, ϵ the dielectric constant and d the distance between plates, i.e. $g_0 - x$ for the top capacitor and $g_0 + x$ for the bottom one.

$$C_{top} = \frac{\epsilon S}{g_0 - x} = \frac{\epsilon S}{g_0} \frac{1}{(1 - x/g_0)} = C_0 \frac{1}{(1 - x/g_0)} \tag{4.2}$$

Correspondingly, for the bottom capacitance:

$$C_{bottom} = \frac{\epsilon S}{g_0 + x} = \frac{\epsilon S}{g_0} \frac{1}{(1 + x/g_0)} = C_0 \frac{1}{(1 + x/g_0)} \tag{4.3}$$

Replacing (4.2) and (4.3) in (4.1) gives: Let $C_0 = \epsilon S/g_0$. Replacing those values in (4.1) gives:

$$V_{middle} = V_0 \frac{C_0 \frac{1}{(1-x/g_0)} - C_0 \frac{1}{(1+x/g_0)}}{C_0 \frac{1}{(1-x/g_0)} + C_0 \frac{1}{(1+x/g_0)}} = V_0 \frac{2C_0 \frac{x/g_0}{(1-(x/g_0)^2)}}{2C_0 \frac{1}{(1-(x/g_0)^2)}} = V_0 \frac{x}{g_0} \tag{4.4}$$

The interesting feature here is the linear relation between the output voltage V_{middle} and the membrane's displacement, even though the capacitance is a non-linear function of x. This is only true for the particular case where the voltage on the upper terminal is the opposite of the bottom one, i.e. $V_{top} = -V_{bottom}$. Of course, this linear relation only holds under strict assumptions that are not exactly met in practice.

4.6.2.2 Mechanical model

The membrane's displacement depends on several forces. In our example we consider only inertial, spring and friction ones. These are assumed to act exclusively at the center of the membrane. The spring force is proportional to displacement and the damping (friction) to velocity. We are here interested in studying the behavior of this system when an external force is applied to the membrane, typically gravity, but it could be any external force. Applying Newton's law, we end up with:

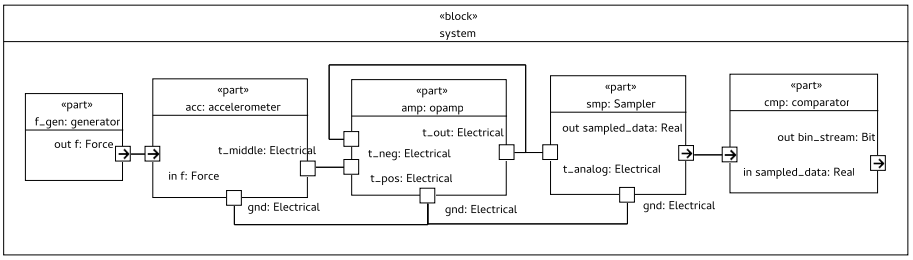

FIGURE 4.12 – SysML Model [IBD]

$$F_{external} = -kx - c\dot{x} + m\ddot{x} \tag{4.5}$$

Several precautions must be taken to accurately extract V_{middle}. Our model uses the most simple read-out circuit, an operational amplifier configured as a buffer. The output of the buffer is fed to a voltage comparator, giving a one-bit output that undergoes further processing in the digital domain. The details of the rest of the system fall outside of the scope of the discussion.

The model of the operational amplifier consists of a single piecewise equation considering the gain and saturation. The latter assures that the output does not exceeds the supply voltages of $V_{DD} = +15V$ and $V_{SS} = -15V$. The piecewise equation is as follows:

$$V_{out}(V_{in}) = \begin{cases} V_{SS} & : V_{in} < V_{SS}/gain \\ V_{DD} & : V_{in} > V_{DD}/gain \\ V_{in} \times gain & : elsewhere \end{cases} \tag{4.6}$$

4.6.2.3 SysML Model

In SysML, we have divided the system into five major blocks as illustrated in figure 4.12:

- the **accelerometer** models the electromechanical dynamics,
- the **opamp** models the operational amplifier,
- the **sampler** adapts analog data to the digital world by periodic sampling,
- the **comparator** checks for a threshold crossing generating a bit stream from the output of the sampler,
- finally a **source** sine wave force generator stimulates the model.

While analog blocks use differential equations defined in continuous time, digital circuitry is best modeled in the discrete domain. These two formalisms handle different types of

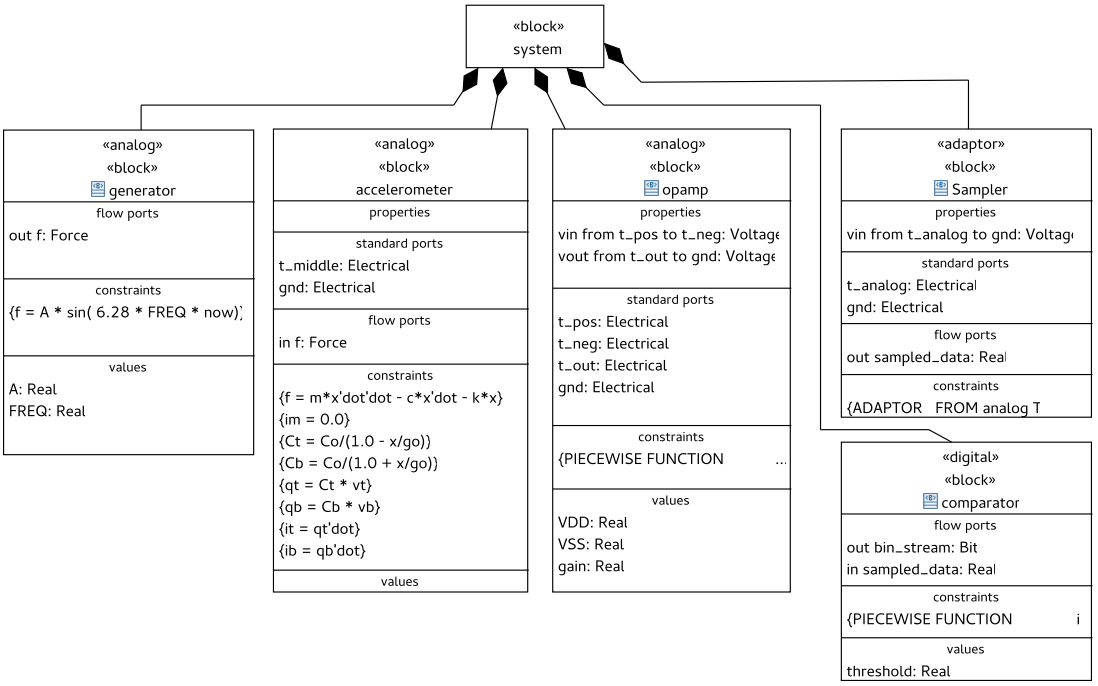

FIGURE 4.13 – SysML Model [BDD]

data and react differently to inputs. If we wish to model and simulate a system with both of them together, we must specify not only the operational semantics (i.e. the Model of Computation) of a particular block but also the semantic adaptation between both domains.

To solve the semantic ambiguity issue, we use custom stereotypes defined in a separate profile. Stereotypes are element modifiers that allow us to give precise meaning to base elements of SysML. In our case, we have chosen to apply specific stereotypes to SysML blocks in order to specify the use of a given model of computation. Since we are dealing with continuous-digital integration, we have added the notion of $\ll$ *analog* $\gg$ and $\ll$ *digital* $\gg$ blocks to SysML as seen in figure 4.13. We have also added one stereotype $\ll$ *adaptor* $\gg$ to specify blocks that are at the frontier of two different MoCs.

For an analog block, we use SysML/UML constraints to describe the physical relations shown previously. The equations defined in SysML constraints are considered to be continuous. The interconnections in an analog block impose other equations that can be inferred from the topology of the system. In the case of electrical circuits, these are the Kirchoff laws. Digital blocks on the other hand are connected by signals that transmit events. Even though in the real world, digital circuits have analog behavior, this formalism abstracts these electrical phenomena making digital circuits design simpler.

One particular case that is worth noting here is the definition of piecewise equations. We have used a particular syntax in SysML constraints to describe these kind of relations. For instance, equation 4.6 describes the simplified behavior of an operational amplifier

and is represented by one SysML constraint preceded by the keywords *PIECEWISE FUNCTION* in our mini-DSL (see figure 4.14-left). In VHDL-AMS, this is translated to *USE* conditions as we see in figure 4.14-right.

```
1 PIECEWISE FUNCTION
2   V_in < VSS/gain:
3      V_out = VSS,
4   V_in > VDD/gain:
5      V_out = VDD,
6   elsewhere:
7      V_out = gain * V_in
```

```
1 IF V_in'ABOVE(VDD/gain) USE
2   V_out == VDD;
3 ELSIF NOT V_in'ABOVE(VSS/gain) USE
4   V_out == VSS;
5 ELSE
6   V_out == gain * V_in;
7 END USE;
```

FIGURE 4.14 – Definition of piecewise equations (SysML vs VHDL-AMS)

We have also defined the quantities that exists between terminals, such as voltage or current using SysML properties (see figure 4.13). These are translated to VHDL-AMS quantities directly.

4.6.2.4 Adaptation Mechanisms

The ≪ *adaptor* ≫ stereotype defines a block whose main purpose is to adapt data, time and/or control from one domain to another. This special block defines the adaptation semantics of heterogeneous interfaces. If they are well defined, then generating executable code from that model should produce the same result regardless of the target language (VHDL-AMS in this case, but it could be any other AMS-capable language, such as SystemC-AMS).

In our example, the block **sampler** is an adaptor from analog to digital domain. It samples data periodically. We do not specify the behavior of the adaptor using our language, rather we instantiate and parameterize a pre-defined adaptor. This is achieved using a SysML constraint starting with the *ADAPTOR* keyword. Figure 4.15 shows our mini-DSL being used to instantiate the sampler.

```
1 ADAPTOR
2   FROM analog TO digital
3   IS sampler
4   PARAMS
5     input     : vin,
6     output    : sampled_data,
7     timestep  : 2us
```

FIGURE 4.15 – Adaptor specification in SysML constraints

Analog data is generated at a dynamic time step in VHDL-AMS simulators. The adaptor specification guarantees that output data will be sampled at a fixed *timestep* of 2 μs. This case of adaptor is interesting because we are not only adapting the time base but also the data format. In the analog domain, ports are considered to be terminals connected

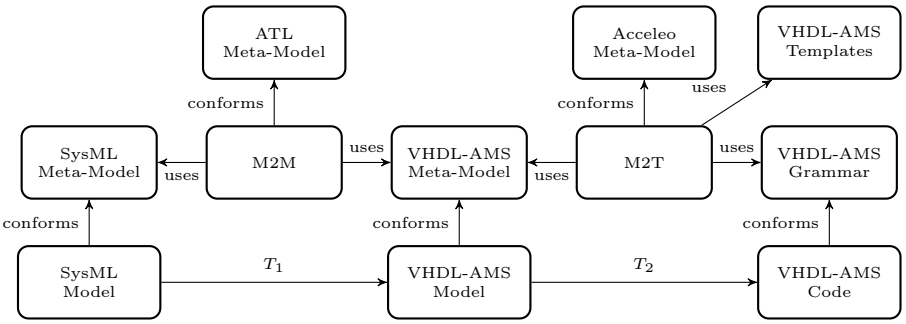

FIGURE 4.16 – Our approach

to nodes of a circuit. Kirchhoff equations can be then deduced from the topology of the circuit. The adaptor must extract the voltage between the input terminal and a reference and propagate it to a discrete event domain as data tokens. The parameters *input* and *output* of figure 4.15 indicate the analog voltage to read and the binary stream to write.

Certain adaptors that we make available to system designers have off-the-shelf counterparts in the target language; others do not. In the former case, our transformation chain chooses adaptors from a standard library; in the latter case it defines new adaptors in the target language.

In this case study, the sampler isn't present in the VHDL-AMS library so we generate the module responsible for this specific adaptation. The generated code can be separated into two different VHDL processes. One for setting the time step and a second one, triggered by the first, to model the adaptor semantics. In this case, the semantics are fairly simple. It consists in copying the analog voltage from input terminal and its reference to the discrete event output at a scheduled moment in time, i.e. every $2\,\mu$s.

The output of the sampler is connected to a comparator which will generate a bit stream from its input. When the input analog voltage crosses a value given by the *threshold* parameter, the digital output switches to a logical value of '1', or '0' otherwise.

4.6.3 Model Transformation

Our approach, illustrated in figure 4.16 consists in two separated phases. Starting from a SysML model, we first perform a model-to-model (M2M) transformation T_1 in order to obtain a VHDL-AMS model. We then generate VHDL-AMS code through a model-to-text (M2T) transformation T_2.

The model-to-model step T_1 translates every SysML element into its equivalent VHDL-AMS element. This step provides the model with semantics on how to interpret SysML elements. For instance, UML ports are converted to terminals while SysML flow ports are

translated to quantity ports. In the same way, a SysML constraint will be transformed into an equation with its variables translated into VHDL-AMS quantities.

For this first step, we have used the Atlas Transformation Language (ATL) [40]. ATL is a language for defining model transformations by a set of rules. Being a model itself, the transformation has its own meta-model. ATL is based on pattern recognition of input elements and conditions which trigger the creation of output elements in the resulting model.

The VHDL-AMS metamodel is an improvement from previous works [2, 30, 64]. It includes the notion of parametrizable adaptors and some slight modifications to the general structure of how libraries are used inside a model. This has proven to be very practical in our implementation but it introduces elements to the metamodel that are not totally part of VHDL-AMS. Instead, a two-step approach separating pure syntax from semantics could also be considered.

Finally, transformation T_2 is responsible for generating the actual code that will be used for running the simulation. For this we use the ACCELEO [49] model-to-text engine from Obeo. In ACCELEO we write templates that specify the code to generate for the various model elements. The adaptors, for instance, were instantiated depending on their type. This is specified in our mini-DSL by the keyword *IS* and is parameterized by the list of parameters listed after *PARAMS*. The generated code is a template with two VHDL processes (as explained in section 4.15).

4.6.4 Simulation Results

Applying both transformations (T_1 and T_2) to the SysML model presented in figure 4.13, we obtain several VHDL-AMS files (one per block) which we use to run simulations. In figure 4.17 we show the output of the Hamster VHDL-AMS simulation tool for a sinusoidal input force.

Note that, despite the non-linear variation of both top and bottom capacitances, the output voltage is linear and follows the input stimulus, which is conform to equation 4.4. The left side of figure 4.17 allows us to conclude that the threshold detection mechanism described by the block **comparator** works correctly as the binary stream output follows the sign of the **opamp**'s output.

A closer look allows us to confirm that digital data is sampled at a fixed time step even tough the analog data is not. The signal *clk* generates events every 2 μs, both on the rising and falling edges. The detail of the right part of figure 4.17 shows that the output voltage was already negative several simulation cycles before the threshold detection.

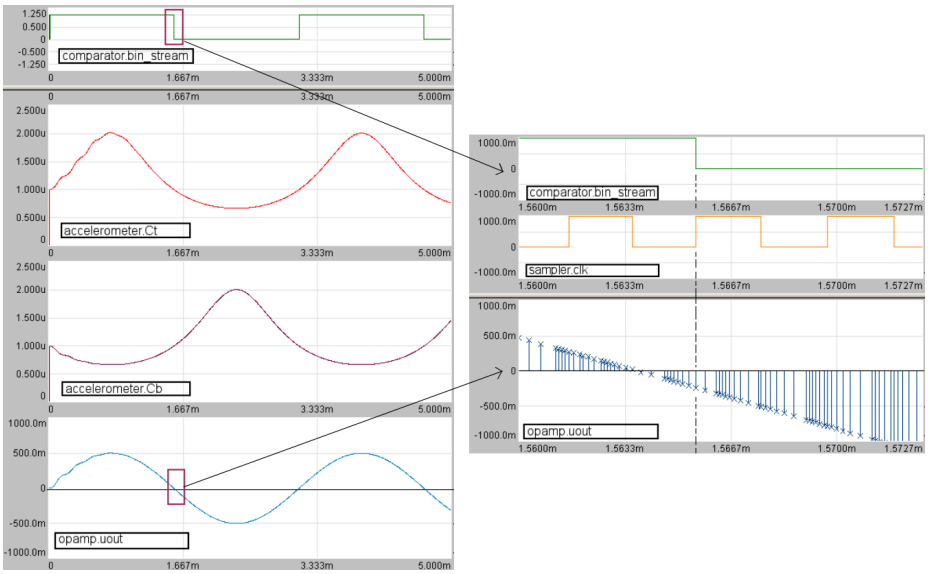

FIGURE 4.17 – Simulation Results

This translates into a delay between the effective crossing of the threshold (bottom left of figure 4.17) and its detection. This is the expected behavior since the specification of the adaptor constraint of figure 4.15 specifies a $2\,\mu$s sampling period thus there can be a delay of up to $2\,\mu$s.

4.6.5 Partial Conclusions

In this section, we have introduced an approach for simulating continuous-digital interaction in SysML models. We validate the behavior through simulation and we generate executable VHDL-AMS code using automatic model transformations. We address the ambiguity of SysML diagrams by assigning them concrete semantics (MoCs) using a simple profile. In order to solve the semantic adaptation problem, we explicitly design adaptation mechanisms using a dedicated language based on SysML constraints. These are translated into specific VHDL-AMS constructs that enforce the specified behavior.

The case study presents a typical case where integration issues occur. We have specified not only the model of computation of individual SysML blocks using stereotypes but also the semantic adaptations between continuous and discrete domains using the notion of adaptor.

The following section is a cumulative follow-up from the previous two contributions. I will try to bridge them together with a more generic approach that could be applied to any textual language. In order to do so, I will first show how to separate the semantic parts of these previous transformations from purely syntactical ones. This will allow us to focus on a more generic approach so as to deal with heterogeneous interactions

independently from the language used to run simulations. I'll then present a generic intermediary metamodel to facilitate transformations to other languages.

4.7 Generalization of the approach

4.7.1 Intermediary Formalism

In the pursuit of a bridge between the two model transformations implemented previously, i.e. from SysML to SystemC-AMS and VHDL-AMS, we have come to the conclusion that an intermediary formalism could be envisaged in order to generalize the work done previously. This intermediary formalism would be the bridge between SysML and textual languages commonly used for heterogeneous systems simulation. It would also include concepts of the semantic adaptation theory, much needed for the semantically correct translation from SysML and textual languages.

We have first tried with DEVS [66] formalism. It looked promising at first glance, but the lack of well-implemented computational tools and a hardly-understandable documentation prevented us from using it as the main intermediary formalism. As a personal note, I believe that DEVS is a very powerful theory and might have been too complex for what we intended to do in this research. After failed tries with DEVS, we have decided to create a custom-tailored language (thus a metamodel) suited only for our specific needs. The main one being to include the concepts of semantic adaptation.

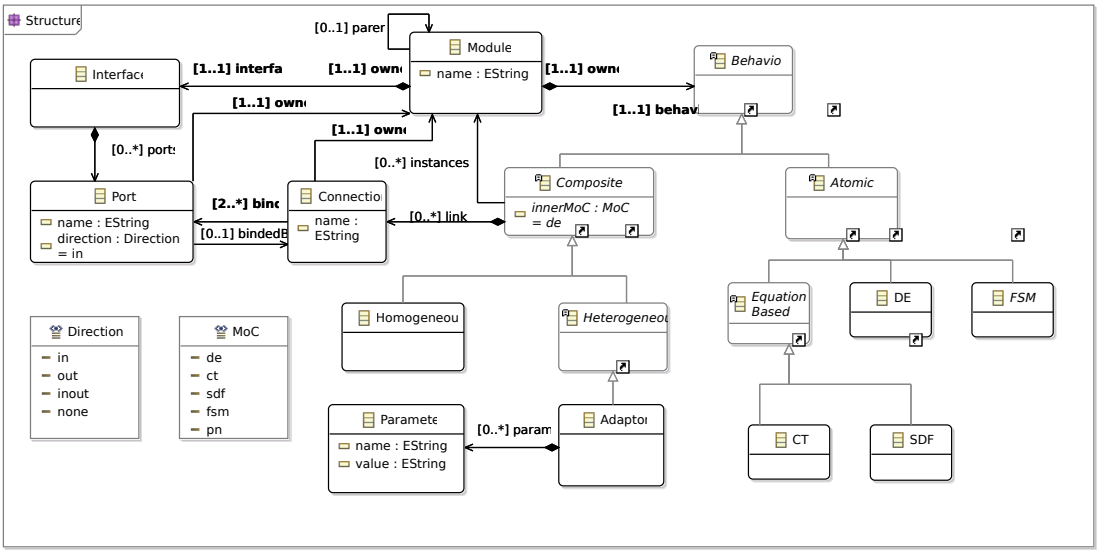

FIGURE 4.18 – Intermediary Metamodel

The important aspects that we have tried to capture in this language are:

- Hierarchical modeling with composition & aggregation bounds
- Separated modeling elements for structure and behavior.
- Dedicated language elements to model semantic adaptation.

The first item comes from the link between a module and its submodules. The atomic module represents an atomic block with no children inside. It can be interpreted as a final hierarchical block. Such modules can be used inside other modules but cannot be further specialized. Compositional modules can be composed of atomic modules or other compositional modules.

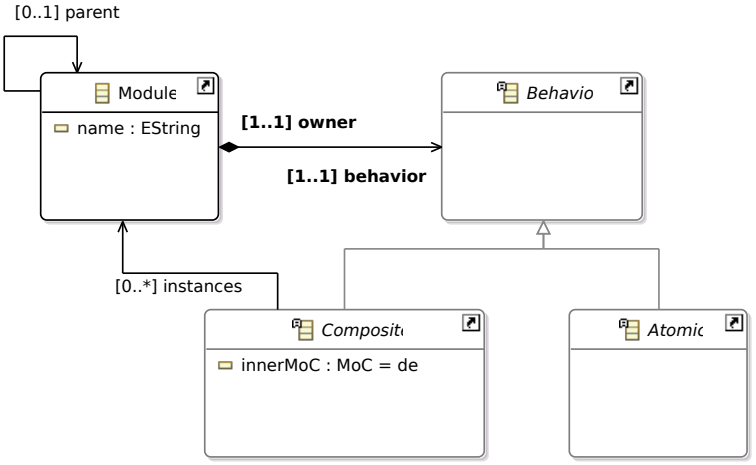

FIGURE 4.19 – Hierarchy in the Intermediary Metamodel

The second major aspect of this language is that it separates the structural elements from behavior definition, just like in VHDL. In VHDL an entity defines only the interface, the type of connectors that should be used and some generic parameters to configure the module. The architecture contains all of the behavioral information of how the module should deal with the incoming data, if any. This was captured in the form of two classes: the **Interface** class and the **Behavior** class.

The Interface class is a simple container for ports. It knows its parent by the owner relation. The Behavior class can be further specialized in any of the available models of computation, i.e. Discrete Event, Finite State Machines, Continuous-Time (equation-based) or even Synchronous Data Flow (also equation-based). These last two are under the class Equation-Based because both of them can be defined only with differential equations and ports bindings. This formalism also takes into account piecewise equations to model possible discontinuities in the function or in its derivative.

4.7.2 Supported Models of Computation

Both continuous time and synchronous data flow models of computation are based on differential equations that model either physical relations or discrete processes. Continuous-Time MoC assumes, as the name suggests, a continuous time scale where the outputs of the system can be calculated for any value $t \in \mathbb{R}$. In a computer simulator though, values will be calculated at discrete points in time because of the discrete nature of the computer. The precision of the output data will depend on the variation rate of the output signal or, as it is the case for many, will depend on the user specification. Some simulators accept only a fixed time-step, others will work with dynamic time-step. In the latter, a few precautions must be taken into account if a Fourier Transform is needed. The discrete Fourier transform, i.e. the one that is actually calculated in a systems simulator, requires periodic data samples. If a dynamic time-step is to be used, an interpolation must take place to correct the position in time of the data samples.

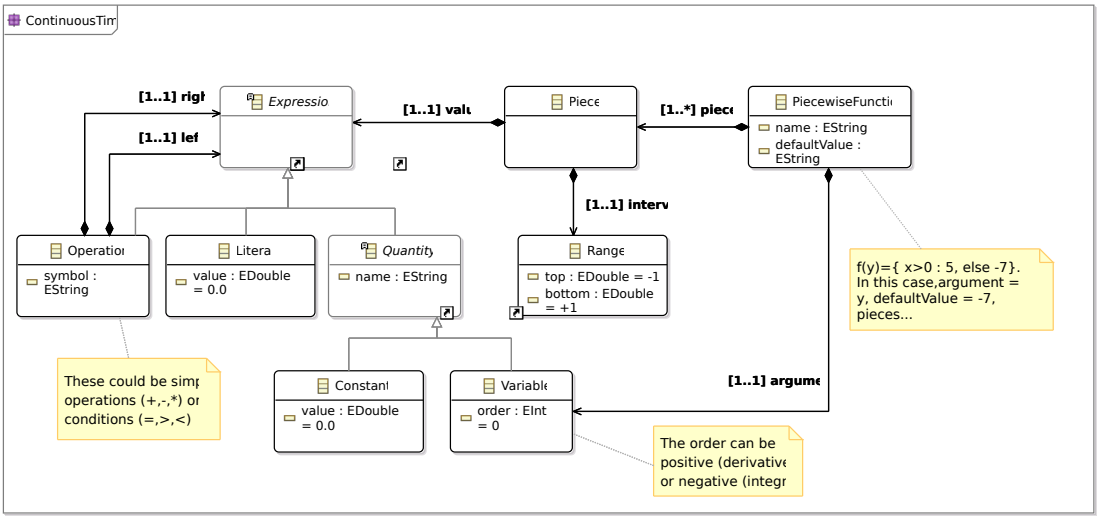

FIGURE 4.20 – Equations subset

In Synchronous Data Flow the time basis is no longer continuous. This formalism is more suited for signal-processing applications and tries to model atomic computing units that can be scheduled in a static or dynamic way. Communication is generalized by FIFOs. Data tokens produced by one block are stored temporarily before being consumed (in the same order as they were produced) by another block. The size of the FIFOs can be calculated previously to the implementation of the system and the abstraction of this kind of communication much simplifies the design of signal-processing applications.

State Machines are probably the most popular model of computation. They were made famous originally by David Harel in his article [33] where he extends conventional state machines to his Statecharts. From that moment onward, State Machines have been

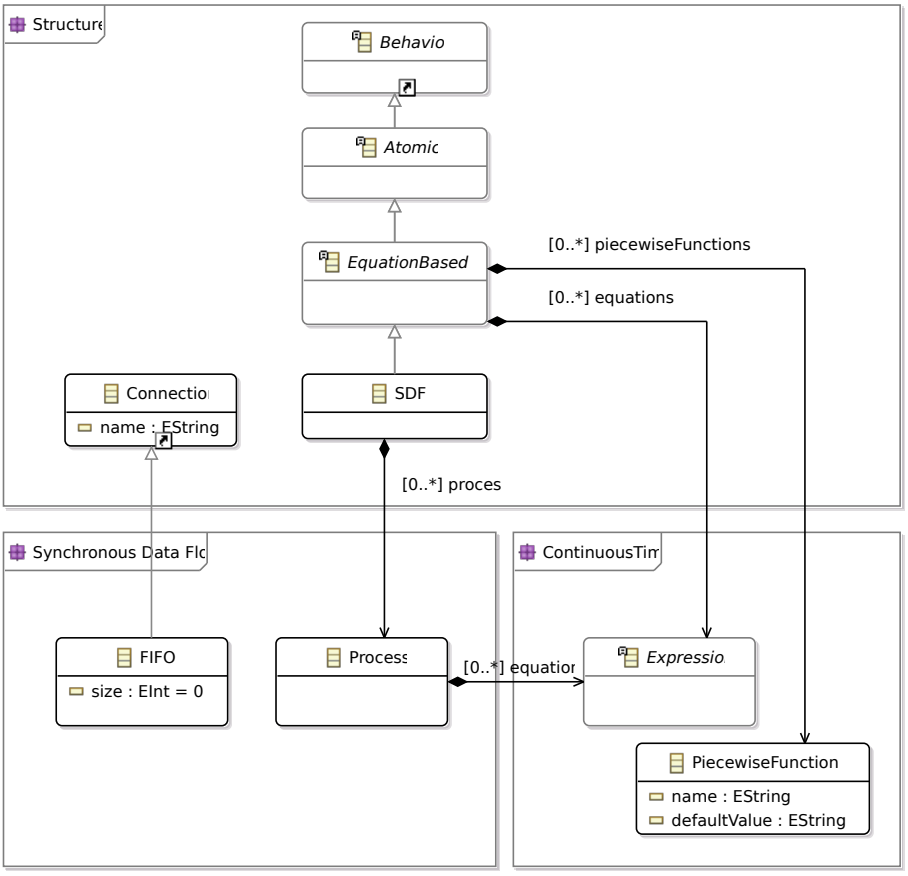

FIGURE 4.21 – Simplified Synchronous Data Flow

heavily used to model different kind of applications. Some of them in the software domain and others in the hardware domain. Finite State Machines are a very natural way to express behavior by a series of state changes. The idea behind this is that a machine can transition from one state to another and perform calculations while transitioning or when in a given state. We will here focus on Finite State Machines with two forms: Synchronous and Asynchronous. Synchronous State Machines are a specialization of State Machines in which transitions can only take place at a given moment in time, dictated by a master signal called clock. These are most used in digital hardware design. Asynchronous State Machines are more appropriated to model systems that are dictated by events that can happen at any time and the system must react almost instantaneously. Asynchronous State Machines can be used to model software for instance.

Finite State Machines in our intermediary metamodel are simplified by the core concepts of a state machine. The FSM behavior is part of the atomic behaviors. One FSM can contain one or many parallel state machines that would run simultaneously. A state machine is only identified by its name. Inside one state machine we can only have states. Transitions are defined exiting a state and have a target state. A transition is composed of a guard and an action. The guard is generally a condition that triggers the transition

to change the state of the machine. The action is generally a value attribution to set an internal variable or an output.

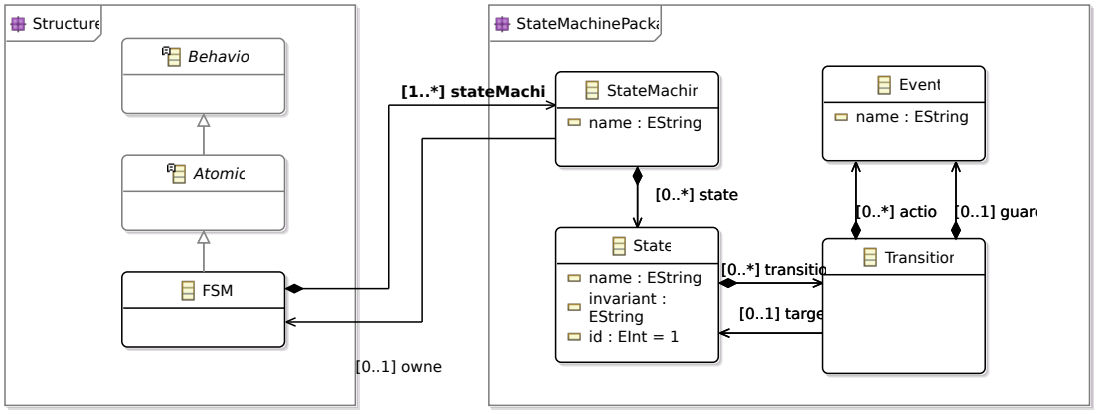

FIGURE 4.22 – Simplified Finite State Machine

Pure discrete event modules do not have yet a model of computation in this intermediary formalism. That is because we consider the simulator where systems will be run to be of a discrete event nature. In fact, the Discrete Event formalism is so generic that most models of computation can be executed on top of it.

4.7.3 Semantic Adatptation

Semantic adaptation is modeled by an heterogeneous composite module called **Adaptor** as seen in figure 4.23. This module can be only defined by a series of parameters containing only a pair name & value. The name of each parameter is also part of the semantics and can only be chosen from a set of pre-defined parameters. These are :

→ Sampling
→ Dynamic Sampling Algorithm
→ Timestep
→ Minimum Timestep
→ Maximum Timestep
→ Data Resolution
→ Interpolation Order
→ Input
→ Output

Sampling refers to the type of data acquisition, whether it is in a fixed time step or in a dynamic time step. The sampling parameter can take therefore two possible values. Either “fixed” or “dynamic”. In case dynamic sampling is needed, another specification is required: the algorithm used for determining the instantaneous time step. If not defined,

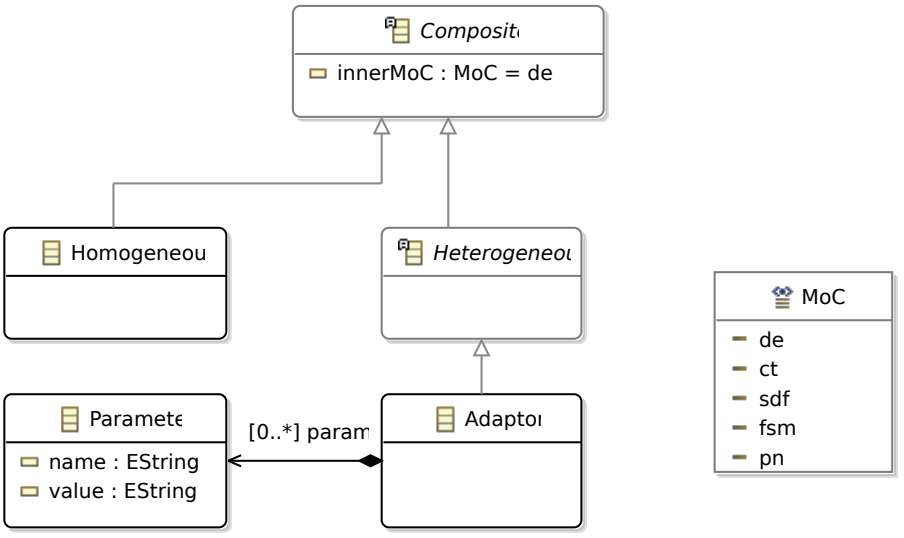

FIGURE 4.23 – Semantic Adaptation

Name	**Symbol**	**Numerical Value**
Picosecond	ps	1 second $\times 10^{-12}$
Nanosecond	ns	1 second $\times 10^{-9}$
Microsecond	us	1 second $\times 10^{-6}$
Milisecond	ms	1 second $\times 10^{-3}$
Second	s, sec	1 second
KiloSecond	ks	1 second $\times 10^{3}$
MegaSecond	Ms, MegaSec	1 second $\times 10^{6}$
GigaSecond	Gs	1 second $\times 10^{9}$
PetaSecond	Ps	1 second $\times 10^{12}$

TABLE 4.1 – Time scales for the timestep parameter

a simple algorithm using the derivative of the output signal should be used. The chosen algorithm should be specified in the **Dynamic Sampling Algorithm** parameter.

Timestep is a time value to determine a fixed time step used by the simulator. Values are composed of a number followed by a timescale. Values such as $15ms$ or $2ns$ are accepted. A full table of possible time scales is given on table 4.1.

If a dynamic sampling mode is selected, then it might be required to set a **Minimum timestep** and/or a **Maximum timestep** in the corresponding attributes.

Data resolution refers to data adaptation. It defines how data will be converted from one domain to the other. If there is need for data interpolation, this should be set in another parameter. In the **data resolution** parameter, we specify the computational representation of a number i.e. the number of bits for the output data or the number of significative digits. Example of accepted entries are: 15 bits integer, 2 digits float. This

means we can either specify to use 15 bits for an integer or a float with 2 significant digits.

Interpolation should take place whenever there is an incompatible time reference. For example, from an analog domain where data is sampled at a dynamic timestep to a digital domain where every process acts according to a global periodic clock. There should be a clear semantics for specifying what happens when the clock ticks and there is no analog data available at the input of the digital block. Should the digital block grab the last available token? Should it consider it to be zero? Should it interpolate with the last 2, 3, ..., n values? The **interpolation order** parameter gives clear semantics for this adaptation. If set to **-1**, it will ignore the absence of data and replace it by *zero* or *false*. If set to **0** it will act like a zero-order hold. It will remember the last input token and will repeat it in its output until another sample is provided. Values above will determine the order of the interpolator used. If set to two, a parabolic interpolation should take place, and so on.

Names of input and output ports should be specified with the **input** & **output** parameters.

4.7.4 Language Example

The example of figure 4.24 shows a fairly simple adaptation from analog domain to digital domain. It was shown in the VHDL-AMS example in order to translate a continuous-time signal to a discrete-event one. In this case, values would be sampled at a fixed time step of 2 micro-seconds with a zero-order hold interpolator which means that an acquired data token should be maintained until the next one is written.

```
1 ADAPTOR
2   FROM analog TO digital
3   IS sampler
4   PARAMS
5     input     : vin,
6     output    : sampled_data,
7     timestep  : 2us
```

FIGURE 4.24 – Semantic Adaptition Specification

The syntax is very straightforward. The adaptor is defined with the keyword ADAPTOR, a label for it is optional and must come after the keyword ADAPTOR. There are no semi-colons to end a line. Tabulation is also part of the language, just like in Python. The adaptor can give hints of the interface it belongs to with the keywords FROM, and TO. These can also be deduced from port bindings of the adaptor. The list of parameter must be defined in pairs of name-value within the PARAMS keyword and separated by commas.

4.7.5 Extended transformation

This new intermediary metamodel is the heart of our new proposed approach. Without touching the code generators, we only have to adapt the previously presented model-to-model transformations and add a first transformation from SysML to this intermediary formalism. The extended transformation would start from a SysML model and depending on the MoC constraint, SysML blocks are mapped to their corresponding MoC domain.

A State Machine block annotated with the *FSM* constraint would be mapped to the State Machine formalism, as described in section 4.7.2. If this block interfaces with a block from another model of computation, an adaptor should be specified in a SysML constraints of an Interface block and thus should be mapped to the adaptor instances of the intermediary formalism. Every parameter would be mapped to instances of *Parameter* of figure 4.23.

4.7.6 Case Study

The case study for this section will be a model of an vehicle's power window. It has simple functions such as an emergency stop when an obstacle is detected. We will model here only the physical model using movement equations and a state machine to control the power window.

The system is composed of a finite state machine control block, a generator to create test inputs and an interface block with SDF atomic blocks defining an equation in the form of a data flow diagram. SysML proposes two different views of the system. A block definition diagram showing only the basic specifications of each block shown in figure 4.26 and an internal block diagram with more details on interconnections and communication patterns. The latter is shown in figure 4.28.

The internal block diagram shows the interconnections between different blocks. The reader should be asking himself now the question: But how does that state machine communicates with the SDF block? As we have done previously in section 4.6.2 the adaptation semantics is defined by a SysML constraint starting with the keyword $ADAPTOR$ shown in figure 4.26. Here is the full adaptor definition :

Now, instead of defining generic domains like *analog* or *digital* we define adaptors between models of computation.

In the SDF block, we model a simple equation with a multiplier, an integrator and a limiter. The multiplier and the integrator are defined with a mathematical constraint

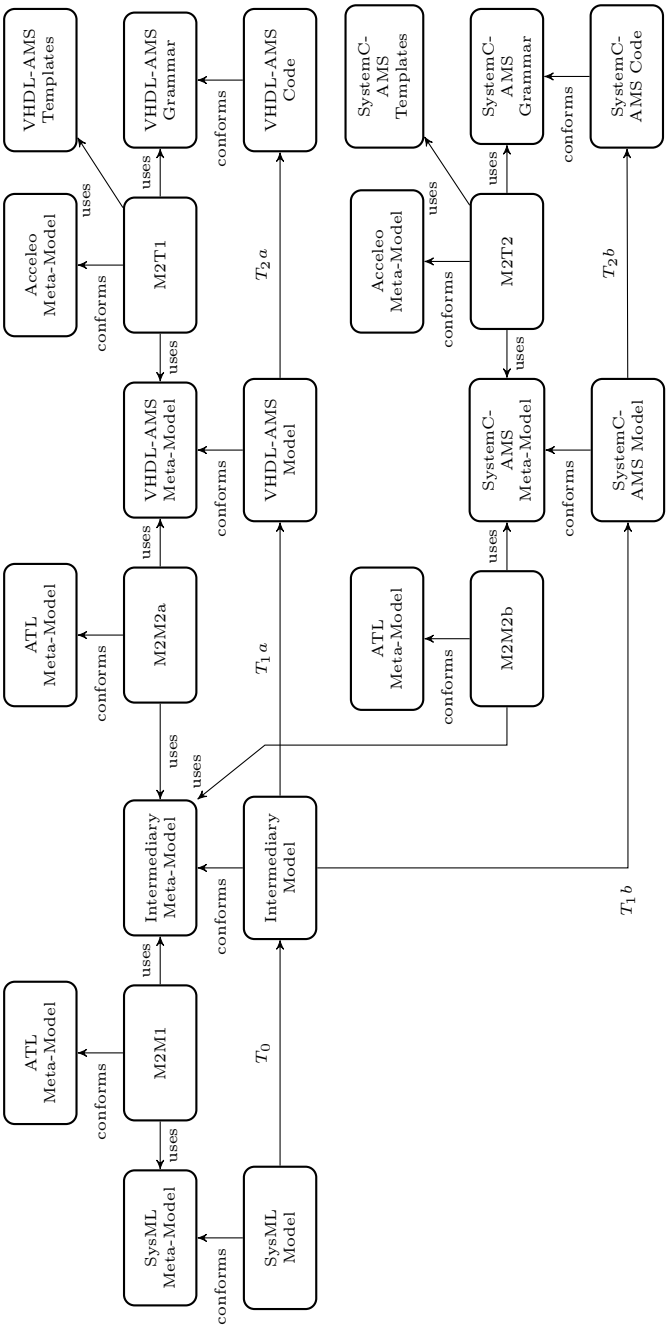

FIGURE 4.25 – The Full Approach

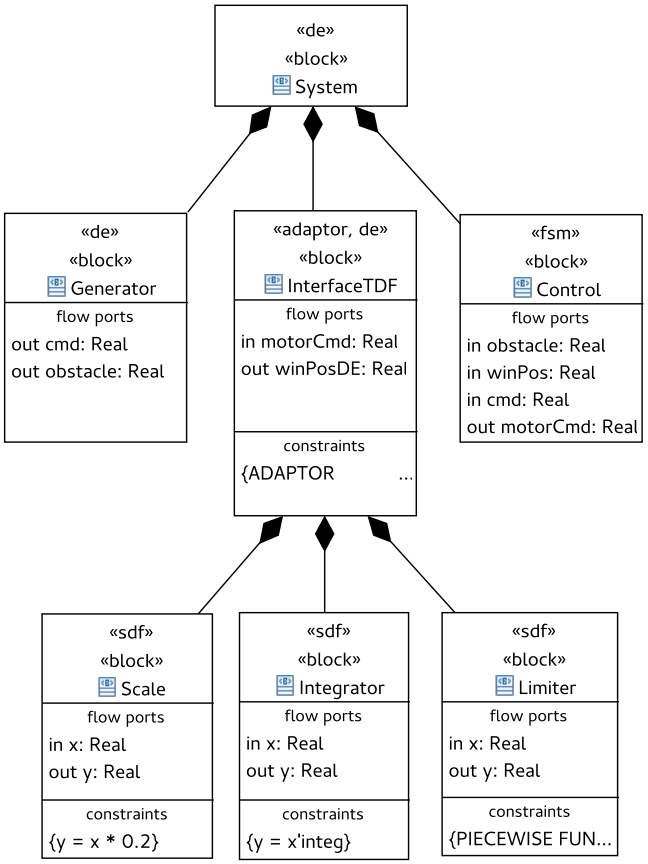

FIGURE 4.26 – Electric Window Block Definition Diagram

```
1 ADAPTOR
2   FROM de TO sdf
3   IS sampler
4   PARAMS
5     timestep : 1 ms
```

FIGURE 4.27 – Semantic Adaptition Specification

shown in figure 4.26. The limiter is defined with a piecewise function and is defined as follows.

The connection of these three modules as shown in figure 4.28 defines the following equation:

$$winPosDE(motorCmd) = \begin{cases} -5 & : motorCmd >= 1 \\ +5 & : motorCmd <= -1 \\ \int (motorCmd \times 0.2) & : elsewhere \end{cases} \quad (4.7)$$

The finite state machine models the control of the power window. If the "up" button is pressed, the control will actuate on the motor to make the window go up. If there

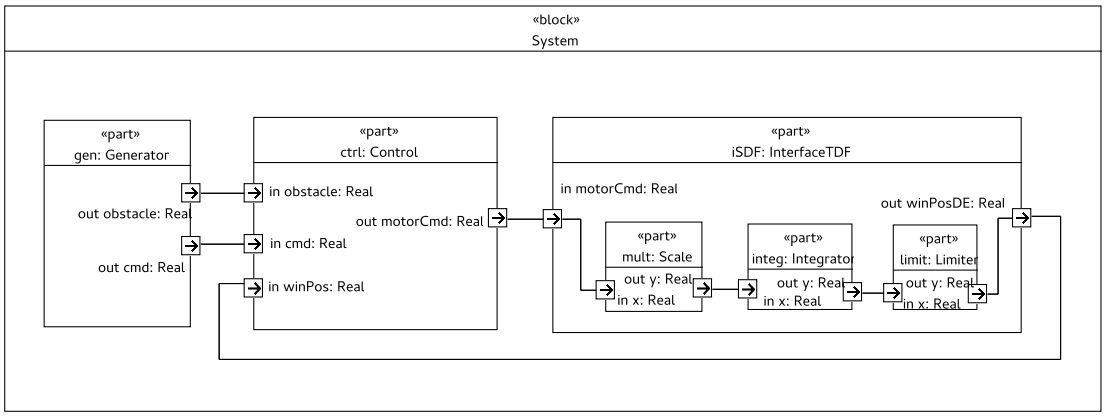

FIGURE 4.28 – Electric Window Internal Block Diagram

```
1 PIECEWISE FUNCTION y@x IS
2   x > 1 : +5,
3   x < −1  : −5,
4   else    : x
```

FIGURE 4.29 – Semantic Adaptition Specification

is an obstacle detected (through the signal "obstacle") while moving up, the control will force the window to open fully even if the button "up" is pressed. We have improved our model transformation to take into account hierarchical state machines. As we can see from figure 4.30 the states "MovingUp", "MovingDown" and "Stopped" have children states with the same properties as their parents. We have used this to declare a state invariant (corresponding to the command sent to the motor) in the parent state only.

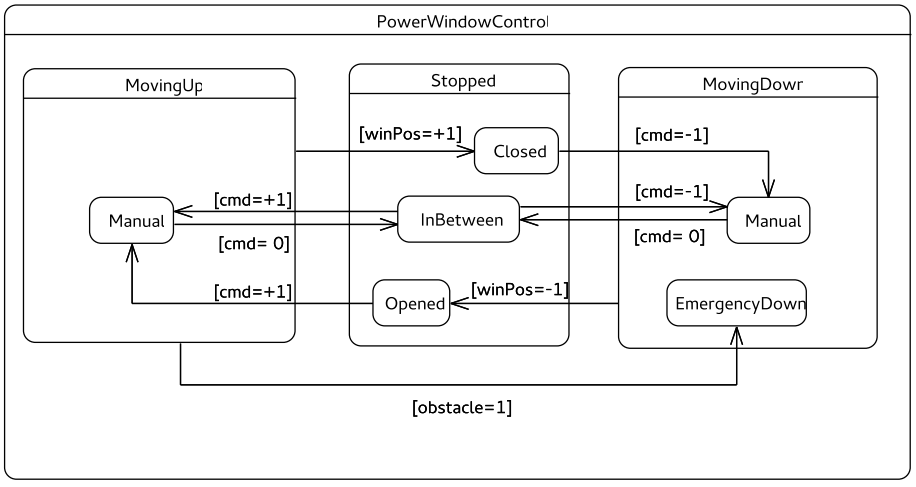

FIGURE 4.30 – Power Window Finite State Machine

4.7.7 Applying the transformation

The result of the first transformation T_0 of our transformation chain is an intermediary representation of the SysML model with the concepts of heterogeneity and semantic adaptation presented previously. MoC constraints are interpreted by our transformation engine and are used to map a SysML block to an intermediary representation with the correct model of behavior.

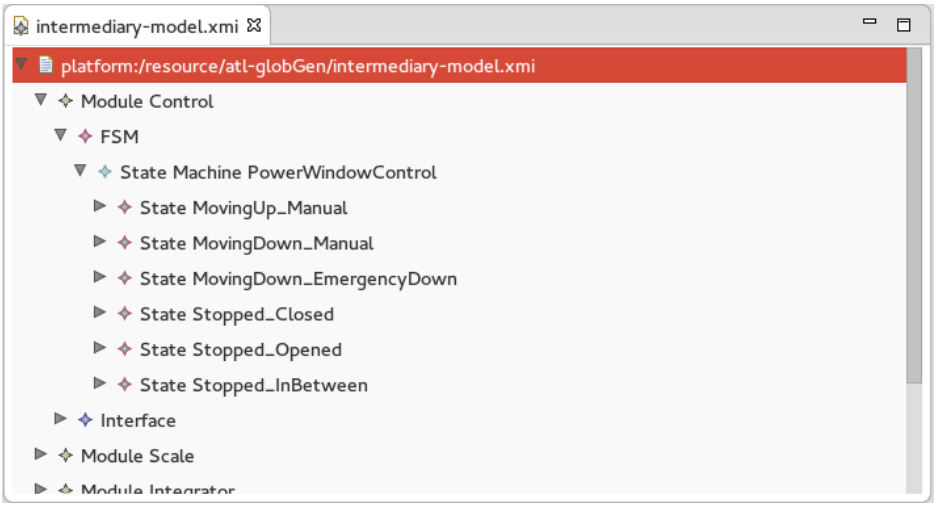

FIGURE 4.31 – Finite State Machine in the Intermediary Model representation

In figure 4.31 we can see how the finite state machine was mapped to a FSM behavior of the metamodel of figures 4.22 and 4.18. The hierarchical state machine was flattened into six states to represent the same behavior described by the hierarchical state machine. Transitions exiting a parent state were replicated to all children states for instance.

FIGURE 4.32 – SDF models in the Intermediary Model representation

Blocks inside the SDF domain were mapped to the SDF behavior of the intermediary metamodel shown in figure 4.21. Equations from different blocks were mapped to separated SDF processes.

The second step of our transformation, as illustrated by T_1a and T_1b in figure 4.25, are a mapping of the intermediary metamodel elements to each of the target languages. Finite state machines in VHDL or in SystemC, for example, are not coded in the form of states and transitions. Instead they are coded in two separated processes: One for the next state logic, and another one to update the current state (thus representing the state register in hardware). The actual coding of state and transitions takes the form of a switch-case construction. If this coding guideline is followed, hardware could be synthesized from both VHDL and SystemC models.

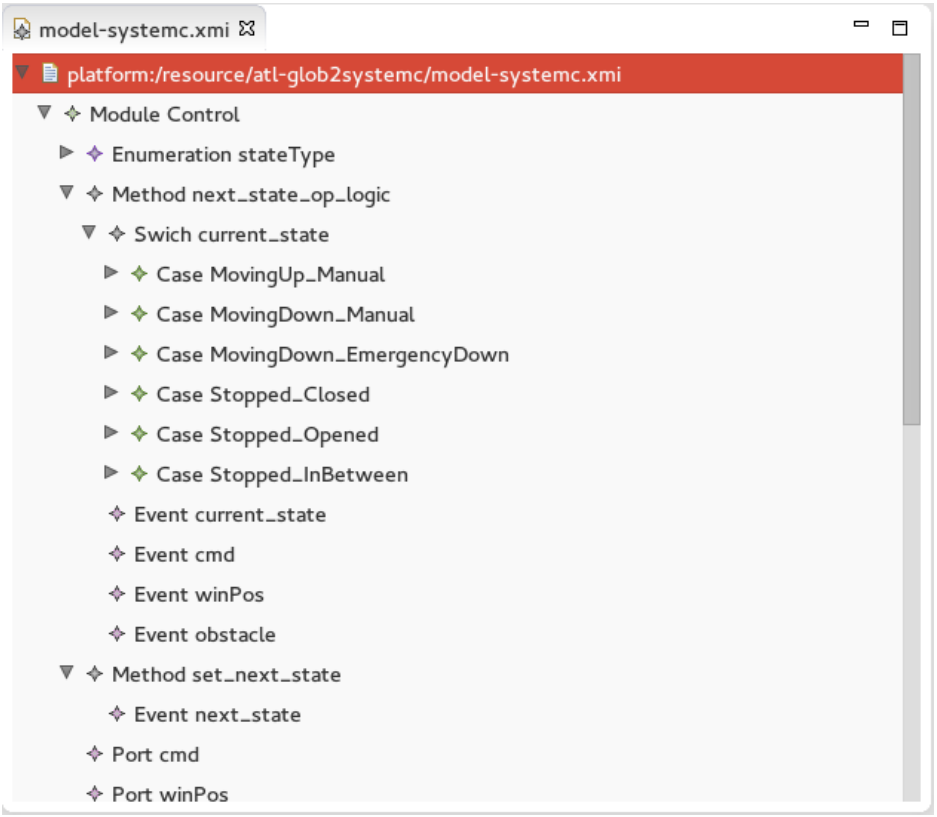

FIGURE 4.33 – Finite State Machine mapping in the SystemC metamodel representation

These transformations (T_1a and T_1b) also interpret the parameters of our adaptors to apply a correct-by-construct semantic adaptation. The TDF interface in SystemC-AMS for example will instantiate specialized channels to make the communication between a discrete event domain to a Timed SDF (TDF) domain.

Finally, the last transformations of the transformation chain, in this case T_2a and T_2b are the simplest of them all. They are nothing but a grammatical mapping from a model to a

textual representation (code). These transformations are composed of textual templates for every model element in the target languages metamodels (i.e. SystemC-AMS and VHDL-AMS) in our case.

4.8 Discussion

From the first to the last implementation of the semantic adaptation theory in the form of model transformations and code generation we can notice a clear evolution and a gradual separation of concerns.

The first attempt described in section 4.5 used SysML constraints and diagram annotations to define concrete semantics to SysML models and semantic adaptation. These provided us the means to generate correct-by-construction SystemC-AMS code. A clear drawback of this approach was the lack of model-driven techniques. The use of diagram annotation is not by any means recommended by the MDE community and in fact, there is a better way to information to SysML models without using annotations: Using profiles.

Profiles are a standard way to extend the semantics of models in a model-driven development. We have used them intensively in our second implementation. The SysML to VHDL-AMS transformation relied on three stereotypes defined by our custom-made profile. These were: $\ll$ *adaptor* $\gg$, $\ll$ *digital* $\gg$ and $\ll$ *analog* $\gg$. These stereotypes have fixed a clear semantic for the stereotyped blocks and thus defined the model of computation that should be used with that block. Not only that, but also the semantic adaptation mechanisms that should be used in order to adapt data, control or time from a domain to another. These stereotypes could be further specified in SysML constraints.

In the third implementation, we have improved the use of these stereotypes to use no longer generic domain names such as digital or analog, but the name of their intended model of computation. Here a choice had to be made. We could either detail the specifications of stereotypes by modeling them in the profile. In this case, the simple application of a stereotype would imply on the specification of a number of parameters such as the data rate for SDF ports or the time step used for a given block. We have chosen not to use this method because it breaks one of our principles that motivated this work. We are here trying to create a way to define semantic adaptation in a explicit way. The "explicit" adverb is very important because we believe that most of integration issues arise from hidden simulator details of how modules should interconnect. We have then chosen to detail stereotypes parameters in SysML constraints, so they could be visible and easily readable with a modern syntax.

Another fact was that gradually, we have separated syntax from semantics definitions in our transformations. The first two implementations mixed a bit of syntax mapping from SysML to each of the target languages and semantics that were coded in the model transformations. In the third implementation, we have delimited pure syntax operations after the intermediary metamodel so that code generation would only deal with syntax mappings. The first model transformation takes a SysML model to an intermediary model with all of our concepts of model of computation and semantic adaptation. This greatly alleviates further model transformations.

Chapter 5

Conclusions & Perspectives

5.1 Conclusion

In this thesis we have shown a way to define concrete semantics to SysML diagrams with stereotypes through the use custom profiles. We have also shown an explicit way to declare the semantic adaptation between blocks of different domains. We have introduced the concept of interface block whose purpose is to be a container to other blocks of a different domain and to constraint semantic adaptation between the outer and the inner model of computation. The theory was presented and developed in the three works presented in chapter 4. We have shown incremental improvements of our technique for each experiment.

The first technique, presented in section 4.5, relied on annotated SysML diagrams using comments to define semantic adaptation. The use of SysML comments is not considered as a standard model driven technique. Gladly, with the feedback of the community, we have improved our technique to use stereotypes provided by custom profiles. Semantics provided by these stereotypes enable correct-by-construction code generation. They provide sound semantics for communication protocols, and thus the model of computation used by each SysML block. A second experiment was shown in section 4.5. It uses stereotypes of a custom profile to map concepts of two different domains, i.e. analog and digital, to VHDL-AMS with the purpose of simulation.

The semantics of each individual MoC is taken into account by our model transformations. In the first step of every transformation chain presented in chapter 4, the meaning of those stereotypes are interpreted by a model-to-model transformation. This allow us to map generic SysML blocks and connectors to their specific MoC concepts, such as processes of SDF blocks, differential equations or state machines. The semantic adaptation is also taken into account providing explicit ways to convert data, time and control

from one domain to another. Constraints are then mapped to several parameters of our intermediary formalism. These parameters will guide the transformation to either instantiate the correct adaptors in the target language (case of study presented in section 4.5), or create them if they do not exist (such as the case presented in section 4.6).

Finally, a proposal of generalization of the approach was presented in section 4.7. An intermediary formalism was presented incorporating concepts of individual models of computation together with semantic adaptation elements. This enables clear and sound semantics not only for homogeneous designs but also and specially for heterogeneous ones. A global transformation chain is proposed in figure 4.25.

Despite de addition of another transformation step, we are convinced that the separation of concerns that this additional step brings with it is mostly beneficent for the extensibility and maintenance of the approach. With this new intermediary formalism, the semantics of each model of computation and the semantic adaptation is coded in the the first transformation, paving the way for following simpler syntax-focused transformations. This was not happening in our first two experiments. The model to model transformation of sections 4.5 and 4.6 were a mix of semantic and syntactical mappings while now they are clearly separated. This helps maintenance of the code and provides better extensibility for other models of computation and also other target languages.

This work was implemented based on Eclipse framework. It depends on Eclipse's modeling tools. Our custom metamodels were based on Ecore's meta-metamodel and our transformations were written in ATL and Acceleo. We have noticed a lack in reutilization of metamodels in the community. Most of the work done here for the languages SystemC, VHDL and their analog extensions (i.e. SystemC-AMS and VHDL-AMS) could be reused by anyone working with code generation or model analysis. The same way, the work done for previous works could be easily reused if the metamodels were freely available in a common repository. Just like for literal languages, like english or french, we only know a part of this language. It composes our limited vocabulary of that language. The same happens with programming and modeling languages. A better modeling of these languages, in the form of metamodels, could be achieved if there were collaborative tools to develop and deploy metamodels for existing languages.

Comparing this work to previous ones [2, 12, 30, 47, 56, 57, 59, 64], we stand out on how to deal with semantic adaptation. None of the previous works that have targeted SysML to SystemC or VHDL code generation has dealt with the semantic inconsistencies that heterogeneity introduces. The semantic adaptation specification language shown in figure 4.24 shows how we deal with inconsistencies. We explicitly model those semantic adaptations in the form of parameters. HetSC [18, 37] has a similar approach, but it only targets SystemC language and not its analog and mixed-signal extensions. UniverCM

is another approach that proposes a solution to the same problem but with a different perspective. Instead of embracing heterogeneity, like Ptolemy, Modhel'X or even this present work, UniverCM tries to keep the design in an homogeneous environment, modeling everything in a generic homogeneous formalism. We defend that by modeling explicitly the heterogeneous interactions, we eliminate misunderstandings of simulations in simulators where the complexity is hidden away from the user and many non explained issues arise simply from integration.

5.2 Perspectives

Because of the generalization of the approach, we can extend this work to other textual target languages. The semantics of the transformation chain are mostly concentrated in the first model-to-model transformation. After the intermediary metamodel, the rest of the transformations are mostly syntactical ones, mapping concepts to each target language.

The inclusion of other models of computation could also be done. This would require the inclusion of the concepts of the specific model of computation in the intermediary metamodel. Although this may sound invasive, the inclusion of another MoC should not provoke any changes in the actual structure of the metamodel. Instead, a simple extension of the behavior class should be done. The clear separation of concerns in the metamodel helps the easy extension to other models of computation.

We can even go further if we use the intermediary metamodel representation for analysis, validation and model checking. This intermediary representation has all of the necessary information for calculating possible SDF schedulings for instance. These verifications could detect flaws on the design in an early phase of development.

Appendix A

Code : SysML to SystemC-AMS

A.1 ATL model to model transformation

```
1  -- @atlcompiler atl2006
2
3  -- @path SC=/ecore.metamodels/systemc.ecore
4  -- @nsURI UML=http://www.eclipse.org/uml2/4.0.0/UML
5  -- @nsURI SYSML=http://www.eclipse.org/papyrus/0.7.0/SysML
6
7  module run;
8
9  create OUT : SC from IN : UML, IN2 : SYSML;
10
11 -- Declare all objects for further manipulation
12 helper def : allModules : Sequence(SC!ComposedModule) = Sequence{};
13 helper def : signals    : Sequence(SC!Signal) = Sequence{};
14 helper def : ports      : Sequence(SC!Port) = Sequence{};
15
16 -- Usefull functions
17 -- aModule : This function takes a named object in its input and provide
18 --   the corresponding module from the Sequence allModules that maches
19 --   the object name.
20 helper def : aModule(o : OclAny) : SC!Module =
21   if (o.oclIsUndefined()) then
22     OclUndefined else
23   thisModule.allModules->
24     select(e | e.name = o.name).first() endif;
25
26 -- Create all base Modules
27 rule lsfModules {
28   from
29     sml_block : SYSML!Block(sml_block.base_Class.name.startsWith('sca_lsf'))
30   to
31     sc_module : SC!lsfModule (name <- sml_block.base_Class.name)
32   do {
33     -- Save this module in a Sequence :
```

```
34      thisModule.allModules <- Sequence{thisModule.allModules, sc_module}->
        flatten();
35    }
36  }
37  rule tdfModules {
38    from
39      sml_block : SYSML!Block(sml_block.base_Class.name.startsWith('sca_tdf'))
40    to
41      sc_module : SC!tdfModule (name <- sml_block.base_Class.name)
42    do {
43      -- Save this module in a Sequence :
44      thisModule.allModules <- Sequence{thisModule.allModules, sc_module}->
        flatten();
45    }
46  }
47
48
49  -- Create all the Composed Modules (those composed of the base modules)
50  rule ComposedModules {
51    from --            (not sml_block.base_Class.isFinalSpecialization)
52      sml_block : SYSML!Block(not sml_block.base_Class.name.startsWith('sc'))
53    to
54        -- Check if is primitive :
55        --     Either by checking isFinalSpecialisation,
56      --     or name.startsWith('sc')
57      sc_module : SC!ComposedModule(name <- sml_block.base_Class.name)
58    do {
59      if (not sml_block.base_Class.ownedRule.isEmpty()) {
60        if (sml_block.base_Class.ownedRule.first().specification.value.toString
        ().endsWith('LSF'))
61          sc_module.formalism <- 'LSF';
62      }
63
64      -- Save this module in a Sequence :
65      thisModule.allModules <-
66        Sequence{thisModule.allModules, sc_module}->flatten();
67      }
68  }
69
70  -- Create all ports for the Composed Modules
71  rule Ports {
72    from
73      sml_port : SYSML!FlowPort
74    to
75      sc_port : SC!Port (
76        name <- sml_port.base_Port.name,
77        type <- sml_port.base_Port.type.name,
78        direction <- sml_port.direction
79      )
80    do {
81      -- First, set the specified formalism, if not specified, default value is
        'DE'
```

```
82      -- TODO
83      -- sc_port.formalism <-
84      for (m in thisModule.allModules) {
85        --sml_port.base_Port.owner.name.debug();
86        if(not thisModule.allModules->
87            select(e | e.name = sml_port.base_Port.owner.name).first().
88              oclIsUndefined() )
89        {
90          thisModule.allModules->
91            select(e | e.name = sml_port.base_Port.owner.name).first().
92              port <- sc_port;
93        }
94      }
95      for (q in sml_port.base_Port.qualifier) {
96        if (q.name.startsWith('isAdaptor'))
97        {
98          thisModule.createAdaptor(thisModule.aModule(sml_port.base_Port.owner)
      ,q,sc_port);
99        }
100     }
101   }
102 }
103
104 rule createAdaptor(m : SC!ComposedModule, q : UML!Property, bindedTo : SC!
      Port) {
105   to
106     sc_adaptor : SC!Adaptor (
107       bindsTo <- bindedTo,
108       direction <- bindedTo.direction,
109       name <- q.name.replaceAll('(', '@').split('@').last().replaceAll(')', '
      @').split('@').first().concat('_' + bindedTo.name),
110       from <- q.name.replaceAll('(', '@').split('@').last().replaceAll(')', '
      @').split('@').first().split('2').first().toUpper(),
111       to   <- q.name.replaceAll('(', '@').split('@').last().replaceAll(')', '
      @').split('@').first().split('2').last().toUpper()
112     )
113   do {
114     m.subModules <- sc_adaptor;
115     --m.adaptors <- sc_adaptor;
116     bindedTo.hasAdaptor <- sc_adaptor;
117   }
118 }
119
120 rule Adaptors {
121   from
122     uml_adapt : SYSML!ConstraintProperty(uml_adapt.base_Property.name.
      startsWith('isAdaptor'))
123 --  to
124 --    sc_adaptor : SC!Adaptor (
125 --      name <- uml_adapt.base_Property.owner.name,
126 --      direction <- uml_adapt.base_Property.owner.direction
127 --      )
```

```
128   do {
129     -- If Ports direction is 'in', use the first formalism, if 'out' use the
      second formalim defined
130     -- in the adaptor
131     if (thisModule.aModule(uml_adapt.base_Property.owner.owner).port->
132       select(e | e.name = uml_adapt.base_Property.owner.name).first().
      direction.toString() = 'in')
133     {
134       thisModule.aModule(uml_adapt.base_Property.owner.owner).port->
135         select(e | e.name = uml_adapt.base_Property.owner.name).first().
      formalism
136           <- uml_adapt.base_Property.name.replaceAll('(', '@').split('@').
      last().replaceAll(')', '@').split('@').first().split('2').first().toUpper
      ();
137     }
138     else
139     {
140       thisModule.aModule(uml_adapt.base_Property.owner.owner).port->
141         select(e | e.name = uml_adapt.base_Property.owner.name).first().
      formalism
142           <- uml_adapt.base_Property.name.replaceAll('(', '@').split('@').
      last().replaceAll(')', '@').split('@').first().split('2').last().toUpper
      ();
143     }
144   }
145 }
146
147 -- Every Composed module can have parts. If any, create subModules
148 rule Parts {
149   from
150     sml_part : UML!Property(
151       sml_part.isComposite and
152       sml_part.type.getAppliedStereotypes()->
153         collect(e | e.name).includes('Block')
154       )
155
156   to
157     sc_module : SC!Module (name <- sml_part.name)
158   do {
159       sc_module.instanceOf <- thisModule.allModules->
160         select(e | e.name = sml_part.type.name).first();
161
162       thisModule.allModules <- Sequence{thisModule.allModules, sc_module}->
      flatten();
163
164       for (p in thisModule.aModule(sml_part.type).port) {
165         thisModule.copyPort(sc_module,p);
166       }
167
168       thisModule.allModules->
169         select(e | e.name = sml_part.owner.name).first().
170           subModules <- sc_module;
```

```
171   }
172 }
173
174 rule copyPort(m : SC!Component, p : SC!Port) {
175   to
176     sc_port : SC!Port (
177       name <- p.name,
178       type <- p.type,
179       --bindsTo <- p.bindsTo,
180       direction <- p.direction,
181       formalism <- p.formalism
182     )
183   do {
184     m.port <- sc_port;
185   }
186 }
187
188 rule Signals {
189   from
190     uml_connector : UML!Connector
191   to
192     sc_signal : SC!Signal (
193       name <- uml_connector.name)
194       --type <- uml_connector.end.last().role.type.name)
195   do {
196     -- For every end of the connector (there should be only two of them)
197     --   bind the signal to the corresponding port.
198     -- If the property partWithPort is Undefined, that
199     --   means that this port belongs to the owner of the connector
200     if (thisModule.aModule(uml_connector.owner).formalism.toString().toLower
      () = 'de') {
201       sc_signal.type <- uml_connector.end.last().role.type.name;
202     } else if(thisModule.aModule(uml_connector.owner).formalism.toString().
     toLower() = 'lsf') {
203       sc_signal.type <- 'lsf_signal';
204     }
205     for (end in uml_connector.end) {
206       if (end.partWithPort.oclIsUndefined()) {
207         sc_signal.bindsTo <- thisModule.aModule(uml_connector.owner).
208           port->select(e | e.name = end.role.name);
209 --        thisModule.aModule(uml_connector.owner).
210 --          port->select(e | e.name = end.role.name).first().bindsTo <-
     sc_signal;
211       } else {
212         sc_signal.bindsTo <- thisModule.aModule(end.partWithPort).
213           port->select(e | e.name = end.role.name);
214 --        thisModule.aModule(end.partWithPort.type).debug().
215 --          port->select(e | e.name = end.role.name).debug().first().bindsTo
     <- sc_signal;
216       }
217     }
218     -- Register signal in the appropriate module.
```

```
219     thisModule.aModule(uml_connector.owner).variable <- sc_signal;
220
221     --thisModule.cropDuplicatedSignals();
222     -- TODO:Find duplicates and merge them
223 --    for (sig_a in thisModule.aModule(uml_connector.owner).variable.debug('
    first')) {
224 --      for (sig_b in thisModule.aModule(uml_connector.owner).variable.
    excluding(sig_a).debug()) {
225 --        if (sig_a.name = sig_b.name) {
226 --          sig_b.bindsTo <- sig_a.bindsTo->excluding(sig_b.bindsTo).debug('
    inside');
227 --
228 --        }
229 --      }
230 --    }
231   }
232 }
233
234 rule cropDuplicatedSignals() {
235   to
236
237   do {
238     for (m in thisModule.allModules) {
239       for (v in m.variable) {
240         if (v.oclIsTypeOf(SC!Signal)) {
241           thisModule.signals <- Sequence{thisModule.signals, v}->flatten();
242         }
243       }
244       for (sig in thisModule.signals) {
245         if (thisModule.signals.debug()->select(e | e.name=sig.name).debug()->
    size() > 1) {
246           for (s in thisModule.signals.debug()->select(e | e.name=sig.name).
247               excluding(thisModule.signals.debug()->select(e | e.name=sig.
    name).first() ) )
248           {
249             thisModule.signals->select(e | e.name=sig.name).first().bindsTo
250              <- s.bindsTo;
251             --s.destroy();
252           }
253           thisModule.signals->select(e | e.name=sig.name).first().bindsTo->
    select(e | e.name<>sig.name);
254         }
255       }
256       thisModule.signals <- Sequence{};
257     }
258   }
259 }
260
261 rule StateMachine {
262   from
263     uml_stm : UML!StateMachine
264   using {
```

```
265     --m : SC!Component = thisModule.aModule(uml_stm.owner);
266     --m : SC!Component = thisModule.allModules.debug()->
267     --          select(e | e.name = uml_stm.owner.debug().name).first();
268   }
269   to
270     -- Create the objects where we'll save the states
271     sc_enum : SC!Enumeration (
272       name <- 'StateType'),
273     -- Instances of Enumeration
274     sc_nst  : SC!Signal(
275       name <- 'next_state',
276       type <- 'StateType'),
277     sc_cst  : SC!Signal(
278       name <- 'current_state',
279       type <- 'StateType'),
280     -- A switch element
281     sc_swch : SC!Swich(
282       var <- sc_cst),
283     sc_ev_cst : SC!Event(name <- 'current_state'),
284     sc_ev_nst : SC!Event(name <- 'next_state'),
285     sc_proc : SC!Method(
286       name  <- 'next_state_op_logic',
287       triggers <- sc_ev_cst,
288       swich <- sc_swch),
289     code_frag : SC!CodeFragment(line <- 'current_state = next_state;'),
290     sc_proc_set_next : SC!Method(
291       name  <- 'set_next_state',
292       triggers <- sc_ev_nst,
293       code  <- code_frag
294       )
295
296   do {
297     -- TODO: Get rid of the "thisModule.aModule(uml_stm.owner)",
298     --   this should be a simple variable "m"
299     -- Should create a trigger for every input of the state machine.
300     for (p in thisModule.aModule(uml_stm.owner).port) {
301       if (p.direction.toString() = 'in') {
302         thisModule.createTrigger(sc_proc,p);
303       }
304     }
305     thisModule.aModule(uml_stm.owner).variable <- sc_enum;
306     thisModule.aModule(uml_stm.owner).variable <- sc_cst;
307     thisModule.aModule(uml_stm.owner).variable <- sc_nst;
308     thisModule.aModule(uml_stm.owner).process  <- sc_proc;
309     thisModule.aModule(uml_stm.owner).process  <- sc_proc_set_next;
310     for (s in uml_stm.region.first().subvertex) {
311       thisModule.createState(s, sc_swch, sc_enum);
312     }
313   }
314 }
315
316 rule createTrigger(process : SC!Process, p : SC!Port) {
```

```
317   to
318     sc_ev : SC!Event (name <- p.name)
319   do {
320     process.triggers <- sc_ev;
321   }
322 }
323
324 -- For every State, we have to add a Case to the Swhitch and a literal to the
        enumeration
325 rule createState(uml_state : UML!State, swch : SC!Swich, enum : SC!
      Enumeration) {
326   to
327     sc_code : SC!CodeFragment(
328       line <- uml_state.ownedRule->first().specification.name),
329     sc_case : SC!Case (
330       item <- uml_state.name,
331       code <- sc_code),
332     sc_literal : SC!NamedElement (name <- uml_state.name)
333   do {
334     for (t in uml_state.outgoing) {
335       thisModule.createIf(sc_case,t);
336     }
337     swch.case <- sc_case;
338     enum.literals <- sc_literal;
339   }
340 }
341
342 -- Create the if clause corresponding to the transition
343 rule createIf(c : SC!Case, t : UML!Transition) {
344   to
345     code_fragment : SC!CodeFragment (
346       line <- 'next_state = ' + t.target.name),
347     sc_if : SC!IfClause (
348       condition <- t.guard.specification.name,
349       code <- code_fragment)
350   do {
351     c.ifClause <- sc_if;
352   }
353 }
```

A.2 ACCELEO Code Generation

```
1  [comment encoding = UTF-8 /]
2  [module run('/metamodels/systemc.ecore')]
3
4  [template public run(m : ComposedModule)]
5  [comment @main /]
6  [if (m.name.startsWith('sc'))]
7    do nothing..
8  [else]
9    [genHeader(m)/]
10   [genBody(m)/]
11 [comment] [genTest(m)/][/comment]
12   [genMain(m)/]
13 [/if]
14 [/template]
15
16 [template public genHeader(m : ComposedModule)]
17 [comment Header /]
18 [file (m.name.concat('.hpp'), false, 'UTF-8')]
19 #ifndef [m.name.toUpper()/]_HPP
20 #define [m.name.toUpper()/]_HPP
21
22 #include "systemc.h"
23 #include "systemc-ams.h"
24 [for (sm : Component | m.subModules)]
25 [includeSubModules(sm)/][/for]
26
27 SC_MODULE([m.name/])
28 {
29 [if (m.port->size() <> 0)]
30
31   // Ports
32 [/if]
33 [for (p : Port | m.port)]
34   sc_[p.direction/]<[p.type/]> [p.name/];
35 [/for]
36 [comment                                                    comment/]
37 [if (m.subModules->selectByType(Adaptor)->size() <> 0)]
38
39   // Adaptors
40 [declareAdaptors(m)/]
41 [/if]
42 [comment                                                    comment/]
43 [if (m.variable->size() <> 0)]
44
45   // Variables
46 [/if]
47 [for (v : Variable | m.variable)]
48   [declareVar(v)/]
49 [/for]
50 [comment                                                    comment/]
```

```
51 [if (m.process->size() <> 0)]
52
53   // Process declaration
54 [/if]
55 [for (p : Process | m.process) ]
56   [p.returnType/] [p.name/]();
57 [/for]
58 [comment                                                    comment/]
59 [if (m.subModules->size() <> 0)]
60
61   // SubModules
62 [/if]
63 [for (sm : Module | m.subModules->selectByType(Module))]
64   [declareSubModule(sm)/]
65 [/for]
66
67   SC_CTOR([m.name/]) [for (sm : Component | m.subModules)
68       before (': ') separator(', ') ][initModules(sm)/][/for]
69   {
70 [if (m.variable->size() <> 0)]
71
72     // Variables Initialization
73 [/if]
74 [for (v : Variable | m.variable)]
75 [if (v.type <> 'lsf_signal')]
76     [initVar(v)/]
77 [/if]
78 [/for]
79 [comment                                                    comment/]
80 [if (m.subModules->selectByType(Adaptor)->size() <> 0)]
81
82     // Input Adaptors
83 [/if]
84 [for (a : Adaptor | m.subModules->selectByType(Adaptor))]
85 [if (a.direction.toString() = 'in')]
86     [instantiateAdaptor(a)/]
87 [/if][/for]
88 [comment                                                    comment/]
89 [if (m.subModules->size() <> 0)]
90
91     // SubModules Instantiation
92 [/if]
93 [for (sm : Module | m.subModules->selectByType(Module))]
94     [instantiateModule(sm)/]
95 [/for]
96 [comment                                                    comment/]
97 [if (m.subModules->selectByType(Adaptor)->size() <> 0)]
98
99     // Output Adaptors
100 [/if]
101 [for (a : Adaptor | m.subModules->selectByType(Adaptor))]
102 [if (a.direction.toString() = 'out')]
```

```
103     [instantiateAdaptor(a)/]
104 [/if][/for]
105 [comment                                                    comment/]
106 [if (m.process->size() <> 0)]
107
108     // Process Registration
109 [/if]
110 [for (p : Process | m.process)]
111   [if (p.oclIsTypeOf(Method))]
112     SC_METHOD([p.name/]);
113   [elseif (p.oclIsTypeOf(Thread))]
114     SC_THREAD([p.name/]);
115   [/if]
116     [for (t : Event | p.triggers)
117        before ('    sensitive << ')
118        separator (' << ')
119        after (';')][t.name/][/for]
120
121 [/for]
122   }
123 };
124 #endif
125 [/file]
126 [/template]
127
128 [template public instantiateAdaptor(a : Adaptor)]
129 [a.name/].[if
130              (a.direction.toString()='in' )][if
131                  (a.from.toString().toLower() = 'de' )]inp([elseif
132                  (a.from.toString().toLower() = 'lsf')]x([/if][a.bindsTo.name
    /]);
133 [elseif
134              (a.direction.toString()='out')][if
135                  (a.from.toString().toLower() = 'de' )]inp([elseif
136                  (a.from.toString().toLower() = 'lsf')]x([/if][a.bindsTo.
    bindsTo.name/]);
137 [/if]
138 [a.name/].[if
139              (a.direction.toString()='in' )][if
140                  (a.to.toString().toLower() = 'de' )]outp([elseif
141                  (a.to.toString().toLower() = 'lsf')]y([/if][a.bindsTo.
    bindsTo.name/]);
142 [elseif
143              (a.direction.toString()='out')][if
144                  (a.to.toString().toLower() = 'de' )]outp([elseif
145                  (a.to.toString().toLower() = 'lsf')]y([/if][a.bindsTo.name
    /]);
146 [/if]
147 [/template]
148
149 [template public declareAdaptors(m : ComposedModule)]
150 [for (a : Adaptor | m.subModules->selectByType(Adaptor))]
```

```
151   sca_[if
152         (a.direction.toString()= 'in')][a.to.  toString().toLower()/][elseif
153         (a.direction.toString()='out')][a.from.toString().toLower()/][/if]::
      sca_[if
154     (a.direction.toString()= 'in')][a.from.toString().toLower()/]::
      sca_source [elseif
155         (a.direction.toString()='out')][a.to.  toString().toLower()/]::
      sca_sink [/if][a.name/];
156 [/for]
157 [/template]
158
159 [template public includeSubModules(sm : Component)/]
160 [template public includeSubModules(sm : Module)]
161 [if (sm.instanceOf.name.startsWith('sc_' ) or
162      sm.instanceOf.name.startsWith('sca_')     )][else]
163 #include "[sm.instanceOf.name/].hpp"
164 [/if]
165 [/template]
166
167 [template public instantiateModule(sm: Component)/]
168 [template public instantiateModule(sm: Module)]
169 [for (port : Port | sm.port)]
170 [for (signal : Signal | port.bindsTo)]
171 [sm.name/].[port.name/]([signal.name/]);
172 [/for]
173 [/for]
174 [/template]
175
176 [template public initModules(m : Component)/]
177 [template public initModules(sm : Module)]
178 [sm.name/]("[sm.name/]")
179 [/template]
180 [template public initModules(sm : lsfModule)]
181 [sm.name/]("[sm.name/]",[sm.rate/])
182 [/template]
183 [template public initModules(sm : tdfModule)]
184 [sm.name/]("[sm.name/]",[sm.rate/])
185 [/template]
186 [template public initModules(sm : Adaptor)]
187 [sm.name/]("[sm.name/]")
188 [/template]
189
190 [template public declareSubModule(sm : Component)/]
191 [template public declareSubModule(sm : Module)]
192 [sm.instanceOf.name/] [sm.name/];
193 [/template]
194
195 [template public genBody(m : ComposedModule)]
196 [file (m.name.concat('.cpp'), false, 'UTF-8')]
197 #ifndef [m.name.toUpper()/]_CPP
198 #define [m.name.toUpper()/]_CPP
199
```

```
#include "[m.name/].hpp"

[for (p : Process | m.process)]
[p.returnType/] [m.name/]::[p.name/]()
{
[for (s : Swich | p.swich)]
[genSwitch(s)/]
[/for]
[for (cl : CodeFragment | p.code)]
  [cl.line/]
[/for]
}
[/for]

#endif
[/file]
[/template]

[template public initVar(v : Variable)]
[v.name/]=[if
  (v.type='int' or v.type='double')]0;[elseif
  (v.type='bool')]false;[elseif
  (v.type='StateType')][
       v.ancestors(ComposedModule)->
       first().variable->filter(Enumeration)->
       first().literals.name->first()/];[/if]
[/template]

[template public genMain(m : ComposedModule)]
[file ('main.cpp', false, 'Cp1252')]
[comment]#include "[m.name.concat('.hpp')/]"
#include "[m.name.concat('_tb.cpp')/]"

int sc_main(int argc, char* argv['[]'/] )
{
  // Elaboration
  // Instantiate the top-level SystemC modules here along
  // with the port bindings etc
  [for (p : Port | m.port) ?(p.direction<>'out')]
    sc_signal<[p.type/]> [p.name/];
  [/for]

  // Instantiate Module
  [m.name/] i_[m.name/]("[m.name/]");
    [for (p : Port | m.port)]
      i_[m.name/].[p.name/]([p.name/]);
    [/for]

  // Instance of the testbench
  [m.name/]_tb tb_[m.name/]("[m.name/]_tb");
    [for (p : Port | m.port) ? (direction<>'out')]
      tb_[m.name/].[p.name/]([p.name/]);
```

```
252     [/for]
253
254   // Trace
255   sc_trace_file *tf = sc_create_vcd_trace_file("wave");
256   [for (p : Port | m.port) ?(p.direction<>'out')]
257     sc_trace(tf,[p.name/],"[p.name/]");
258   [/for]
259
260   // Start the main simulation thread
261   sc_core::sc_start(100,SC_NS);
262
263   // Post processing -- parsing the log files etc
264   sc_close_vcd_trace_file(tf);
265   return 0;
266 }[/comment]
267 #include "System.hpp"
268
269 int sc_main(int argc, char* argv['[]'/] )
270 {
271   // Elaboration
272   // Instantiate the top-level SystemC modules here along
273   // with the port bindings etc
274
275   // Instantiate Module
276   System i_system("system");
277
278   // Trace
279   sc_trace_file *tf = sc_create_vcd_trace_file("wave");
280     sc_trace(tf,i_system.connector1,"v");
281     sc_trace(tf,i_system.connector2,"d");
282     sc_trace(tf,i_system.connector3,"F");
283
284   // Start the main simulation thread
285   sc_core::sc_start(5,SC_SEC);
286
287   // Post processing -- parsing the log files etc
288   sc_close_vcd_trace_file(tf);
289   return 0;
290 }
291 [/file]
292 [/template]
293
294 [template public genTest(m : ComposedModule)
295 {
296   ports_in : OrderedSet(Port) = m.port->select(direction<>'out');
297   ModuleName : String = m.name.concat('_tb');
298 }]
299 [file (ModuleName.concat('.cpp'), false, 'UTF-8')]
300 #ifndef [m.name.toUpper()/]CPP
301 #define [ModuleName.toUpper()/]_CPP
302
303 #include "systemc.h"
```

```
304 #include "math.h"
305 SC_MODULE([ModuleName/])
306 {
307   // Ports
308 [for (p : Port | ports_in)]
309   sc_out<[p.type/]> [p.name/];
310 [/for]
311
312   SC_CTOR([ModuleName/])
313   {
314     SC_THREAD(run);
315   }
316
317   void run() {
318     while(true) {
319     [for (p : Port | ports_in)]
320       [p.name/].write(rand() % 10);
321     [/for]
322       wait(5,SC_NS);
323     }
324   }
325
326 };
327
328 #endif
329 [/file]
330 [/template]
331
332 [template public declareVar(v : Variable)]
333 [v.type/] [v.name/];
334 [/template]
335
336 [template public declareVar(v : Signal)]
337 [if (v.type = 'lsf_signal')]
338 sca_lsf::sca_signal [v.name/];[else]
339 sc_signal<[v.type/]> [v.name/];[/if]
340 [/template]
341
342 [template public declareVar(v : Enumeration)]
343 enum [v.name/] {
344 [for (item : NamedElement| v.literals) separator (',')]
345   [item.name/][/for] };
346
347 [/template]
348
349 [template public genSwitch(s : Swich)]
350 switch ([s.var.name/]) {
351 [for (c : Case| s.case)]
352   case [c.item/] :
353   [for (ic : IfClause | c.ifClause)]
354     if ([ic.condition/]) {
355     [for (code : CodeFragment | ic.code)]
```

```
356       [code.line/];
357     [/for]
358     }
359   [/for]
360   [for (code : CodeFragment | c.code)]
361     [code.line/];
362   [/for]
363     break;
364
365 [/for]
366   default :
367     break;
368 };
369 [/template]
370
371 [template public initVar(v : Enumeration)/]
```

Appendix B

Code : SysML to VHDL-AMS

B.1 ATL model to model transformation

```
1  -- @atlcompiler atl2006
2
3  -- @path VHDL=/metamodel/vhdl.ecore
4  -- @nsURI SYSML=http://www.eclipse.org/papyrus/0.7.0/SysML
5  -- @nsURI UML=http://www.eclipse.org/uml2/4.0.0/UML
6
7  module m2m;
8
9  create OUT : VHDL from IN : SYSML, IN2 : UML;
10
11 -- Declare all objects for further manipulation
12 helper def : allModels : Sequence(VHDL!Model) = Sequence{};
13
14 -- Usefull functions
15 -- aModule : This function takes a named object in its input and provide
16 --   the corresponding module from the Sequence allModules that maches
17 --   the object name.
18 helper def : aModel(o : OclAny) : VHDL!Model =
19   if (o.oclIsUndefined()) then OclUndefined else
20     thisModule.allModels->
21       select(e | e.entity.name = o.name).first() endif;
22
23 rule block {
24   from
25     sml_block : SYSML!Block(not sml_block.base_Class.isAbstract)
26   to
27     vha_model  : VHDL!Model,
28     vha_entity : VHDL!Entity (name <- sml_block.base_Class.name), --.
       replaceAll(' ', '_')
29     vha_archi  : VHDL!Architecture (name <- 'behavior')
30   do {
31     -- Save this model in a sequence
```

```
32    thisModule.allModels <- Sequence{thisModule.allModels, vha_model}->
      flatten();
33
34    vha_model.entity       <- vha_entity;
35    vha_model.architecture <- vha_archi ;
36    vha_archi.entity       <- vha_entity;
37
38    -- Get all attributes (including inherited ones)
39    for (e in  sml_block.base_Class.member) {
40      -- Constraints are equations
41      if (e.oclIsTypeOf(SYSML!Constraint)) {
42        -- If using a piecewise function, create a conditioned equation.
43        if (e.specification.value.toString()->startsWith('PIECEWISE FUNCTION'
      ))
44          thisModule.createConditionedEquation(e.specification.value,
      vha_archi);
45        -- If using the adaptor language, create a 'use' equation
46        if (e.specification.value.toString()->startsWith('ADAPTOR'))
47          thisModule.createAdaptor(e.specification.value,vha_archi);
48        -- If not, create a simple equation
49        else
50          thisModule.createEquation(e.specification.value,vha_archi);
51      }
52      if (e.oclIsTypeOf(SYSML!Property)) {
53        -- If it has a default value, create a generic
54        if (not e.defaultValue.oclIsUndefined())
55          thisModule.createGeneric(e,vha_entity);
56        -- If not, create a quantity
57        else
58          thisModule.createQuantity(e,vha_archi);
59      }
60      --
61      if (e.oclIsTypeOf(SYSML!Port) and e.getAppliedStereotypes().first().
      oclIsUndefined())
62        -- Workaround (papyrus bug 23566) : Heritaded ports do not appear on
      internal block diagram.
63        --   If there is no terminal with the same name, create the terminal.
64        if (vha_model.entity.ports->select(terminal | terminal.name = e.name)
      .isEmpty())
65          -- Actually create the terminal.
66          thisModule.createTerminal(e,vha_model);
67
68    }
69  } -- end "do"
70} -- end "rule"
71
72rule createQuantityPort {
73  from
74    sml_port : SYSML!FlowPort (sml_port.base_Port.type.name = 'Force'
      or
75                  sml_port.base_Port.type.name = 'Displacement' or
76                  sml_port.base_Port.type.name = 'Velocity'     or
```

```
77                    sml_port.base_Port.type.name = 'Acceleration'    )
78  to
79    vha_port : VHDL!QuantityPort ( name <- sml_port.base_Port.name,
80                     type <- sml_port.base_Port.type.name,
81                     direction <- sml_port.direction.name)
82  do {
83    thisModule.aModel(sml_port.base_Port.owner).entity.ports <- vha_port;
84    --thisModule.addLibraryOnce(thisModule.aModel(sml_port.base_Port.owner),
      'ieee','mechanical_systems'); -- Simplorer
85    thisModule.addLibraryOnce(thisModule.aModel(sml_port.base_Port.owner), '
      disciplines','kinematic_system'); -- Hamster
86
87  }
88 }
89
90 rule createTerminal (sml_port : SYSML!Port, model : VHDL!Model ) {
91  to
92    vha_port : VHDL!Terminal ( name <- sml_port.name,
93                    type <- sml_port.type.name)
94  do {
95    model.entity.ports <- vha_port;
96    if (sml_port.type.name = 'Electrical') {
97      --thisModule.addLibraryOnce(model, 'ieee','electrical_systems');  --
      Simplorer
98      thisModule.addLibraryOnce(model, 'disciplines','electromagnetic_system'
      ); -- Hamster
99    }
100 }
101 }
102
103 rule createSubComponent {
104  from
105    sml_composite : UML!Association
106    -- Test if both ends of the connector are << blocks >>
107    (
108      sml_composite.memberEnd.first().type.getAppliedStereotypes()->
109        collect(e | e.name).includes('Block')
110    and
111      sml_composite.memberEnd.last().type.getAppliedStereotypes()->
112        collect(e | e.name).includes('Block')
113    )
114  do {
115    -- Add the composite one to the "submodules" property of the other
116    if (sml_composite.memberEnd.first().isComposite) {
117      thisModule.aModel(sml_composite.memberEnd.last().type).architecture.
      first().submodules <-
118        thisModule.aModel(sml_composite.memberEnd.first().type);
119      for (lib in thisModule.aModel(sml_composite.memberEnd.first().type).lib
      ) {
120        for (package in lib.usePackage) {
121          thisModule.addLibraryOnce(thisModule.aModel(sml_composite.memberEnd
      .last().type), lib.name,package.name);
```

```
122         }
123       }
124     }
125     else if (sml_composite.memberEnd.last().isComposite) {
126       thisModule.aModel(sml_composite.memberEnd.first().type).architecture.
    first().submodules <-
127         thisModule.aModel(sml_composite.memberEnd.last().type);
128     }
129   }
130 }
131
132 rule createDigitalPort {
133   from
134     sml_port : SYSML!FlowPort (sml_port.base_Port.type.name = 'Signal' )
135   to
136     vha_port : VHDL!DigitalPort ( name <- sml_port.base_Port.name,
137                         type <- sml_port.base_Port.type.name,
138                         direction <- sml_port.direction.name)
139   do {
140     thisModule.aModel(sml_port.base_Port.owner).entity.ports <- vha_port;
141   }
142 }
143
144 rule connectors {
145     from
146         uml_connector : UML!Connector
147     do {
148       -- If the connector does not yet exists,
149       if (thisModule.aModel(uml_connector.owner).architecture.first().
    connectors
150           ->select(connector | connector.name = uml_connector.name).isEmpty()
    )
151       {
152       -- Create a connector inside the given architecture
153         thisModule.createConnector(
154           uml_connector,
155           thisModule.aModel(uml_connector.owner).architecture.first()
156       );
157     }
158       else -- If there is already a connector with the same name, then extend
     the existing one.
159       {
160         for (uml_port in uml_connector.end) {
161           thisModule.aModel(uml_connector.owner).entity;
162           if (thisModule.aModel(uml_connector.owner).architecture.first().
    connectors -- from existing connectors
163           ->select(connector | connector.name = uml_connector.name).first()
        -- select the current one (with the same name as the uml_connector)
164           .binds->select( port | port.name = uml_port.role.name).isEmpty() )
        -- and verify that there is no port with the same name as the
    uml_connector ends
165           {
```

```
166             thisModule.aModel(uml_connector.owner).architecture.first().
    connectors
167             ->select(connector | connector.name = uml_connector.name).first()
    .binds
168               <- thisModule.aModel(uml_port.role.owner).entity.ports
169                   ->select(vha_port | vha_port.name = uml_port.role.name);
170         }
171         }
172       }
173     }
174 }
175
176 rule createConnector(uml_connector : UML!Connector, vha_arch : VHDL!
    Architecture) {
177   to
178         vha_connector : VHDL!Connector
179   do {
180     -- Set the name of the connector
181         vha_connector.name <- uml_connector.name;
182       vha_connector.arch <- vha_arch;
183     -- Bind the two ends to corresponding ports
184       for (uml_port in uml_connector.end) {
185         vha_connector.binds <- thisModule.aModel(uml_port.role.owner).entity.
    ports
186                       ->select(vha_port | vha_port.name = uml_port.role.name)
    ;
187
188         -- And set the type accordingly
189         vha_connector.type <- uml_port.role.type.name;
190         if (uml_port.role.type.name = 'Electrical')
191           vha_connector.nature <- 'Terminal';
192         else if (uml_port.role.type.name = 'Signal')
193           vha_connector.nature <- 'Signal';
194         else -- Voltage, Current, Force, etc ..
195           vha_connector.nature <- 'Quantity';
196       }
197
198       -- Add the connector to the corresponding model's body (architecture)
199       vha_arch.connectors <- vha_connector;
200   }
201 }
202
203 rule createEquation(eq : String, archi : VHDL!Architecture) {
204   to
205     vha_equation : VHDL!Equation
206   do {
207     vha_equation.lhs <- eq.split('=').first().regexReplaceAll('\\^', '**');
208     vha_equation.rhs <- eq.split('=').last().regexReplaceAll('\\^', '**');
209     archi.equations  <- vha_equation; }
210 }
211
212 rule createConditionedEquation(eq : String, archi : VHDL!Architecture) {
```

```
213   to
214     vha_equation : VHDL!Equation
215   do {
216     -- Piecewise Function
217     --    (interval a) : (equation a),
218     --    (interval b) : (equation b),
219     --        ...      :     ...
220     --    elsewhere    : (equation n),
221     --
222     vha_equation.lhs <- eq.split('elsewhere').last().split(':').last().split(
      '=').first().trim().regexReplaceAll('\\^', '**');
223     vha_equation.rhs <- eq.split('elsewhere').last().split(':').last().split(
      '=').last( ).trim().regexReplaceAll('\\^', '**');
224     for(piecewise in eq.split('PIECEWISE FUNCTION').last().split(',')) {
225       if (not piecewise.trim().startsWith('elsewhere'))
226         thisModule.addPiecewiseFunction(vha_equation,piecewise.split(':').
      first(),piecewise.split(':').last());
227     }
228     archi.equations  <- vha_equation;
229   }
230 }
231
232 rule createAdaptor(eq : String, archi : VHDL!Architecture) {
233   to
234     vha_adaptor : VHDL!Adaptor(name <- 'adaptor')
235   do {
236     -- ADAPTOR
237     --    FROM moc1 TO moc2
238     --    IS adaptorType
239     --    PARAMS
240     --      p1 : value1,
241     --      .  :   .
242     --      pN : valueN
243     --
244     vha_adaptor.MoC_to   <- eq.split('FROM').last().split('IS').first().split
      ('TO').last().trim();
245     vha_adaptor.MoC_from <- eq.split('FROM').last().split('IS').first().split
      ('TO').first().trim();
246     vha_adaptor.type <- eq.split('IS').last().split('PARAMS').first().trim();
247     for(param in eq.split('PARAMS').last().split(',')) {
248       thisModule.addParam(vha_adaptor,param.split(':').first().trim(),param.
      split(':').last().trim());
249     }
250     archi.adaptors  <- vha_adaptor;
251   }
252 }
253
254
255 rule addParam(adaptor : VHDL!Adaptor, name : String, value : String) {
256   to
257     vha_param : VHDL!Parameter
258   do {
```

```
259     vha_param.name  <- name;
260     vha_param.value <- value;
261     adaptor.params  <- vha_param;
262   }
263 }
264
265 rule addPiecewiseFunction(vha_equation : VHDL!Equation, condition : String,
        equation : String) {
266   to
267     vha_cond_eq : VHDL!ConditionedEquations (condition <- condition.trim(),
268                           lhs       <- equation.split('=').first().trim() ,
269                           rhs       <- equation.split('=').last( ).trim() )
270   do {vha_equation.piecewise <- vha_cond_eq;}
271 }
272
273 rule createQuantity(quantity : SYSML!Property, archi : VHDL!Architecture) {
274   do {
275     if (quantity.name.split('from').size() > 1) {
276       -- If type is Voltage
277       if (quantity.type.name = 'Voltage') {
278         self.addQuantityBranch('across',
279           quantity.name.split('from').first().trim(),
280           quantity.name.split(' from ').last().split(' to ').first().trim(),
281           quantity.name.split(' from ').last().split(' to ').last().trim(),
282           archi);
283       }
284       -- If type is Current
285       if (quantity.type.name = 'Current' ) {
286         self.addQuantityBranch('through',
287           quantity.name.split('from').first().trim(),
288           quantity.name.split('from').last().split('to').first().trim(),
289           quantity.name.split('from').last().split('to').last().trim(),
290           archi);
291       }
292     }
293     -- If it is not a branch quantity, it is a free quantity
294     else if (quantity.type.name = 'Real'         or
295          quantity.type.name = 'Capacitance'  or
296          quantity.type.name = 'Force'        or
297          quantity.type.name = 'Displacement' or
298          quantity.type.name = 'Velocity'     or
299          quantity.type.name = 'Acceleration'    )
300     {
301       self.addQuantity(
302         quantity.name,
303         quantity.type.name,
304         archi);
305     }
306     else if (quantity.type.name = 'Electrical') {
307       self.addNode(
308         quantity.name,
309         quantity.type.name,
```

```
310         archi);
311     }
312   }
313 }
314
315 rule createGeneric(gen : SYSML!Property, entity : VHDL!Entity) {
316   to
317     vha_generic : VHDL!Generic (name <- gen.name,
318                       type <- gen.type.name)
319   do {
320
321     if (gen.defaultValue.name.toString().endsWith('p'))
322       vha_generic.initialValue <- gen.defaultValue.name.toString().substring
      (1,gen.defaultValue.name.toString()->size()-1).concat('.0E-12');
323     else if (gen.defaultValue.name.toString().endsWith('n'))
324       vha_generic.initialValue <- gen.defaultValue.name.toString().substring
      (1,gen.defaultValue.name.toString()->size()-1).concat('.0E-9');
325     else if (gen.defaultValue.name.toString().endsWith('u'))
326       vha_generic.initialValue <- gen.defaultValue.name.toString().substring
      (1,gen.defaultValue.name.toString()->size()-1).concat('.0E-6');
327     else if (gen.defaultValue.name.toString().endsWith('m'))
328       vha_generic.initialValue <- gen.defaultValue.name.toString().substring
      (1,gen.defaultValue.name.toString()->size()-1).concat('.0E-3');
329     else if (gen.defaultValue.name.toString().endsWith('k'))
330       vha_generic.initialValue <- gen.defaultValue.name.toString().substring
      (1,gen.defaultValue.name.toString()->size()-1).concat('.0E03');
331     else if (gen.defaultValue.name.toString().endsWith('M'))
332       vha_generic.initialValue <- gen.defaultValue.name.toString().substring
      (1,gen.defaultValue.name.toString()->size()-1).concat('.0E06');
333     else if (gen.defaultValue.name.toString().endsWith('G'))
334       vha_generic.initialValue <- gen.defaultValue.name.toString().substring
      (1,gen.defaultValue.name.toString()->size()-1).concat('.0E09');
335     else if (gen.defaultValue.name.toString().endsWith('T'))
336       vha_generic.initialValue <- gen.defaultValue.name.toString().substring
      (1,gen.defaultValue.name.toString()->size()-1).concat('.0E12');
337     else
338       vha_generic.initialValue <- gen.defaultValue.name.toString().substring
      (1,gen.defaultValue.name.toString()->size()).concat('.0');
339
340     entity.generics <- vha_generic;
341   }
342 }
343
344 rule addQuantityBranch(branchType : String, name : String, plus_terminal :
      String, minus_terminal : String, archi : VHDL!Architecture) {
345   to
346     vha_quantity : VHDL!QuantityBranch (
347       name <- name,
348       branchType <- branchType,
349       plusTerminal <- plus_terminal )
350
351   do {
```

```
352     if (not (plus_terminal = minus_terminal))
353       vha_quantity.minusTerminal <- minus_terminal;
354
355     archi.quantities <- vha_quantity;
356   }
357 }
358
359 rule addQuantity(name : String, type : String, archi : VHDL!Architecture) {
360   to
361     vha_quantity : VHDL!Quantity (name <- name, type <- type )
362
363   do {
364     archi.quantities <- vha_quantity;
365   }
366 }
367
368 rule addNode(name : String, type : String, archi : VHDL!Architecture) {
369   to
370     vha_node : VHDL!Terminal (name <- name, type <- type )
371
372   do {
373     archi.nodes <- vha_node;
374   }
375 }
376
377 rule addLibraryOnce(model : VHDL!Model, alibrary : String, aPackage : String)
       {
378   do {
379     if (model.lib->select(lib | lib.name = alibrary).isEmpty()) {
380       thisModule.addLibrary(model, alibrary);
381       thisModule.addPackage(model, alibrary, aPackage);
382     }
383     else if (model.lib->select(lib | lib.name = alibrary).first().usePackage
      ->select(package | package.name = aPackage).isEmpty()) {
384       thisModule.addPackage(model, alibrary, aPackage);
385     }
386   }
387 }
388
389 rule addLibrary(model : VHDL!Model, alibrary : String) {
390   to
391     vha_lib : VHDL!Library (name <- alibrary)
392   do {
393     model.lib <- vha_lib; }
394 }
395
396 rule addPackage(model : VHDL!Model, alibrary : String, aPackage : String) {
397   to
398     vha_pack: VHDL!Package (name <- aPackage)
399   do {
400     model.lib->select(lib | lib.name = alibrary).first().usePackage <-
      vha_pack;
```

```
401     }
402 }
```

B.2 ACCELEO Code Generation

```
1  [comment encoding = UTF-8 /]
2  [module model('/metamodel/vhdl.ecore')]
3
4  [template public aModel(aModel : Model)]
5  [comment @main/]
6  [file (self.entity.name.replaceAll(' ', '_').concat('.vhd'), false, 'UTF-8')]
7  [comment
8
9  This template generates one file for every model found.
10 It can be separated in three parts :
11   - Library declaration
12   - Entity
13   - Architecture
14
15 /]
16 [for (lib : Library | self.lib)]
17 library [lib.name/];
18 [for (pack : Package | lib.usePackage)]use [lib.name/].[pack.name/].all;
19 [/for][/for]
20
21 [comment if has entity /]
22 entity [self.entity.name.replaceAll(' ', '_')/] is
23   [for (g : Generic | self.entity.generics) before ('\tgeneric (\n')
       separator(';\n') after ('\n\t);\n')]
24     [g.name/] : [g.type/] := [g.initialValue/][comment
25 /][/for]
26   [for (p : Port | self.entity.ports) before ('\tport (\n') separator (';\n')
       after ('\n\t);\n')]
27   [declarePort(p)/][comment
28 /][/for]
29 end entity;
30 [comment end if has entity /]
31
32 [comment if has architecture /]
33 [for (a : Architecture | self.architecture)]
34 architecture [a.name/] of [a.entity.name.replaceAll(' ', '_')/] is
35   [comment :
36       Declare quantities
37   /]
38   [for (node : Terminal | a.nodes) before ('\n')]
39   terminal [node.name/] : [node.type/];
40   [/for]
41   [comment :
42       Declare quantities
43   /]
44   [for (q : Quantity | a.quantities) before ('\n')]
45   [declareQuantity(q)/]
46   [/for]
47   [comment :
48       Components declaration
```

```
49   /]
50   [for (e : Model | a.submodules) separator('\n')]
51   component [e.entity.name.replaceAll(' ', '_')/] is
52     [for (g : Generic | e.entity.generics) before ('\tgeneric (\n') separator
       (';\n') after ('\n\t);\n')]
53     [g.name/] : [g.type/] := [g.initialValue/][comment
54   /][/for]
55     [for (p : Port | e.entity.ports) before ('\tport (\n') separator (';\n')
      after ('\n\t);\n')]
56   [declarePort(p)/][comment
57   /][/for]
58     end component;
59   [/for]
60   [comment :
61       Connectors go here
62   /]
63   [for (connector : Connector | a.connectors)]
64   [connector.nature/] [connector.name/] : [connector.type/];
65   [/for]
66
67 begin
68 [comment :
69     Component Instantiation
70 /]
71 [for (instance : Model | a.submodules) before ('\n') separator ('\n')]
72   i_[instance.entity.name.replaceAll(' ', '_')/] : component [instance.entity
       .name.replaceAll(' ', '_')/]
73   [for (g : Generic | instance.entity.generics) before ('\tgeneric map(\n')
      separator(',\n') after ('\n\t)\n')]
74     [g.name/] => [g.initialValue/][comment
75   /][/for]
76     [for (p : Port | instance.entity.ports) before ('\tport map(\n')
     separator (',\n') after ('\n\t);\n')]
77         [comment : TODO Verify the context we are /]
78     [p.name/] => [p.con->select(c : Connector | c.arch.entity.name = aModel.
      entity.name).name/][comment
79   /][/for]
80 [/for]
81 [comment :
82   Declare adaptors
83 /]
84 [for (adapt : Adaptor | a.adaptors)]
85 [if (adapt.type = 'Threshold_Detector')]
86
87 sampling_clock : process
88   begin
89     clk <= '1';
90     wait for [adapt.params->select(name = 'timestep').value/];
91     clk <= '0';
92     wait for [adapt.params->select(name = 'timestep').value/];
93   end process;
94
```

```
95    output <= output_signal;
96
97    compare : process (clk) is
98    begin
99      if vin > threshold then
100       output_signal <= '1';
101     else
102       output_signal <= '0';
103     end if;
104 [/if]
105 [/for]
106 [comment :
107   Declare equations
108 /]
109 [for (eq : Equation | a.equations) before ('\n')]
110   [declareEquations(eq)/]
111 [/for]
112
113 end architecture [a.name/];
114 [/for]
115 [comment : End "if" has architecture /]
116 [/file]
117 [/template]
118
119 [comment ———————————————————————————————— /]
120
121 [template public declarePort(p : Port)]
122   Should never happen
123 [/template]
124 [template public declarePort(p : QuantityPort)]
125   quantity [p.name/] : [p.direction/] [p.type/]
126 [/template]
127 [template public declarePort(p : DigitalPort)]
128   [p.name/] : [p.direction/] [p.type/]
129 [/template]
130 [template public declarePort(p : Terminal)]
131   terminal [p.name/] : [p.type/]
132 [/template]
133
134 [comment ———————————————————————————————— /]
135
136 [template public declareQuantity(q : Quantity)]
137 quantity [q.name/] : [q.type/];
138 [/template]
139 [template public declareQuantity(q : QuantityBranch)]
140 quantity [q.name/] [q.branchType/] [q.plusTerminal/][if (not q.minusTerminal.
      oclIsUndefined())] to [q.minusTerminal/][/if];
141 [/template]
142
143 [comment ———————————————————————————————— /]
144
145 [template public declareEquations(eq : Equation)]
```

```
146 [if (eq.piecewise->isEmpty())]
147 [eq.lhs/] == [eq.rhs/];
148 [else]
149 [for (cond_eq : ConditionedEquations | eq.piecewise) before ('if') separator
        ('else if')]
150 [cond_eq.condition/] use
151   [cond_eq.lhs/] == [cond_eq.rhs/];
152 [/for]
153 else
154   [eq.lhs/] == [eq.rhs/];
155 end use;
156 [/if]
157 [/template]
```

Appendix C

Code : SysML to All

C.1 ATL : SysML to Intermediary representation

```
1  -- @nsURI SysML=http://www.eclipse.org/papyrus/0.7.0/SysML
2  -- @path GLOB=/Metamodels/glob.ecore
3  -- @nsURI UML=http://www.eclipse.org/uml2/4.0.0/UML
4  -- @path MoC=/SysML-Profile-MoC/model.profile.uml
5
6  module globber;
7  create OUT : GLOB from IN : SysML, IN1 : UML;
8
9  -- allModules : A sequence of objects containing all created Modules
10 -- getModule  : This function takes the name of a Module and provides
11 --   the Module from the sequence 'allModules'.
12 helper def : allModules : Sequence(GLOB!State) = Sequence{};
13 helper def : getModule(name : String) : GLOB!Module =
14   thisModule.allModules->select(e | e.name = name).first();
15
16 -- allStates : A sequence of objects containing all created States
17 -- getState : This function takes a named object in its input and provides
18 --   the corresponding state from the Sequence allStates that matches
19 --   the object name.
20 helper def : allStates : Sequence(GLOB!State) = Sequence{};
21 helper def : getState(name : String) : GLOB!State =
22   thisModule.allStates->select(e | e.name = name).first();
23
24 --  An atomic module should have the option isLeaf to indicate
25 --     that it is an end device. We could also look for this information
26 --     in two other ways. We could look for compositions or other
27 --     parts attributes or we could verify the use of the stereotype
28 --     << adaptor >>. The attribute isLeaf seamed to be the best option
29 --     because it provides clear semantics without ambiguity neither the
30 --     hassle of creating complicated transformation rules.
31 --
32 --  Here, we distinguish one rule for each MoC
33 --
```

```
34 ------------------------------------------------------------------------

35 -- Creation of Finite State Machines (FSM) behavior :
36 --    * When a state machine is detected it triggers the creation
37 --      of a FSM behavior with state machine inside it.
38 --    * This rule will convert a hierarchical state machine into
39 --      a flat one, as long as hierarchy is limited to two levels.
40 ------------------------------------------------------------------------

41
42 rule AtomicFSMModule {
43
44   from
45     sml_block : SysML!Block (sml_block.base_Class.isLeaf and
46                  sml_block.base_Class.getAppliedStereotypes()
47                            ->select(s | s.name = 'fsm').size() = 1)
48   to
49     glob_module      : GLOB!Module (name <- sml_block.base_Class.name),
50     glob_interface   : GLOB!Interface,
51     glob_fsmBehavior : GLOB!FSM
52   do {
53     glob_module.interface  <- glob_interface;
54     glob_module.behavior   <- glob_fsmBehavior;
55     glob_fsmBehavior.owner <- glob_module;
56     -- Save this module to the list of all Modules
57     thisModule.allModules <- thisModule.allModules.append(glob_module);
58   }
59 }
60
61
62 rule StateMachine {
63   from
64     sml_stmachine : UML!StateMachine
65   to
66     --glob_fsmBehavior : GLOB!FSM,
67     glob_stmachine   : GLOB!StateMachine (name <- sml_stmachine.name)
68   do {
69     for (region in sml_stmachine.region) {
70       -- Get all States _o/
71       for (state in region.subvertex) {
72         -- Atomic states
73         if (state.region.size()=0)
74           thisModule.createState(glob_stmachine,state);
75         -- Hierarchical States
76         else {
77           for (subRegion in state.region) {
78             for (subState in subRegion.subvertex) {
79               thisModule.createState(glob_stmachine,subState);
80             }
81           }
82         }
83       }
```

```
84      }
85      thisModule.getModule(sml_stmachine.owner.name).behavior.stateMachines <-
        glob_stmachine;
86      glob_stmachine.owner <- thisModule.getModule(sml_stmachine.owner.name).
        behavior;
87    }
88  }
89
90  rule createState(stMachine : GLOB!StateMachine, uml_states : UML!State) {
91
92    to
93      -- The name is composed of the state and it's owner.
94      glob_state : GLOB!State (name <- uml_states.container.owner.name.toString
        ().concat('_').concat(uml_states.name.toString()) )
95    do {
96
97      -- If there is an invariant, propagate it to the intermediary model
98      if (not uml_states.stateInvariant.oclIsUndefined())
99        glob_state.invariant <- uml_states.stateInvariant.name;
100     -- The invariant of the parent state must also be propagated
101     if (uml_states.owner.owner.oclIsKindOf(UML!State))
102       if (not uml_states.owner.owner.stateInvariant.oclIsUndefined())
103         glob_state.invariant <- uml_states.owner.owner.stateInvariant.name;
104
105     -- Add this state to the state machine ..
106     stMachine.states <- glob_state;
107     -- ... and to the list of all States
108     thisModule.allStates <- thisModule.allStates.append(glob_state);
109   }
110 }
111
112 rule SingleTransitions {
113   from
114     uml_transition : UML!Transition (uml_transition.source.region.size() = 0)
115   to
116     glob_guard : GLOB!Event(name <- uml_transition.guard.specification.value)
      ,
117     glob_transition : GLOB!Transition (
118       target <- thisModule.getState(uml_transition.target.container.owner.
        name.concat('_').concat(uml_transition.target.name))
119 --      guard <- uml_transition.guard.specification.value
120       )
121   do {
122     glob_transition.guard <- glob_guard;
123     thisModule.getState(uml_transition.source.container.owner.name.concat('_'
      ).concat(uml_transition.source.name)).transition <- glob_transition;
124   }
125 }
126
127 rule MultiTransitions {
128   from
```

```
129     uml_transition : UML!Transition (uml_transition.source.region.size() <>
        0)
130   do {
131     for (region in uml_transition.source.region) {
132       for (state in region.subvertex) {
133         thisModule.createSingleTransition(
134           thisModule.getState(region.owner.name.concat('_').concat(state.name
        )),
135           thisModule.getState(uml_transition.target.container.owner.name.
        concat('_').concat(uml_transition.target.name)),
136           uml_transition.guard.specification.value);
137       }
138     }
139   }
140 }
141
142 rule createSingleTransition(stateFrom : GLOB!State, stateTo : GLOB!State,
        guard : String ) {
143   to
144     glob_guard : GLOB!Event (name <- guard),
145     glob_transition : GLOB!Transition (
146       target <- stateTo)
147       --,guard <- guard)
148   do {
149     glob_transition.guard <- glob_guard;
150     stateFrom.transition <- glob_transition;
151   }
152 }
153
154 ------------------------------------------------------------------------------
155 -- Creation of Continuous Time (CT) behavior :
156 --     * A CT module is composed of equations and piecewise functions
157 --     * Equations continuous functions of time
158 --     * Piecewise functions are defined in ranges of an argument.
159 ------------------------------------------------------------------------------
160
161 rule AtomicCTModule {
162
163   from
164     sml_block : SysML!Block (sml_block.base_Class.isLeaf and
165                   sml_block.base_Class.getAppliedStereotypes()
166                               ->select(s | s.name = 'ct').size() = 1)
167   to
168     glob_module     : GLOB!Module (name <- sml_block.base_Class.name),
169     glob_interface  : GLOB!Interface,
170     glob_ctBehavior : GLOB!CT
171   do {
172     -- Link interface and behavior to the module.
173     glob_module.interface <- glob_interface;
174     glob_module.behavior  <- glob_ctBehavior;
```

```
175     -- Save this module to the list of all Modules
176     thisModule.allModules <- thisModule.allModules.append(glob_module);
177
178     for (constraint in sml_block.base_Class.ownedRule) {
179       if (constraint.specification.value.startsWith('PIECEWISE'))
180         thisModule.createPiecewiseFunction(glob_module, constraint);
181       else
182         thisModule.createSingleEquation(glob_module,constraint);
183     }
184   }
185 }
186
187 rule createSingleEquation(glob_block : GLOB!Module, uml_constraint : UML!
      Constraint) {
188   to
189     glob_var   : GLOB!Variable(name <- uml_constraint.specification.value)
190   do {
191     if (glob_block.behavior.oclIsTypeOf(GLOB!SDF))
192       glob_block.behavior.equations <- glob_var;
193     if (glob_block.behavior.oclIsTypeOf(GLOB!CT))
194       glob_block.behavior.equations <- glob_var;
195   }
196 }
197
198 rule createPiecewiseFunction(glob_block : GLOB!Module, uml_constraint : UML!
      Constraint) {
199   to
200     argument : GLOB!Variable (name <- uml_constraint.specification.value
201                   .split('@').last()
202                   .split('IS').first()
203                   .trim() ),
204     glob_piecewiseFunction : GLOB!PiecewiseFunction (name <- uml_constraint.
      specification.value
205                   .split('FUNCTION').last()
206                   .split('@').first()
207                   .trim() )
208   do {
209     glob_piecewiseFunction.argument <- argument;
210     glob_piecewiseFunction.defaultValue <- uml_constraint.specification.value
      .split('else').last().split(':').last().trim();
211     glob_block.behavior.piecewiseFunctions <- glob_piecewiseFunction;
212
213     for (expression in uml_constraint.specification.value.toString().split('
      IS').last().split(',')) {
214       if (not expression->contains('else'))
215         thisModule.createPiece(glob_piecewiseFunction,
216                         uml_constraint.specification.value
217                         .split('@').last()
218                         .split('IS').first()
219                           .trim(),
220                       expression.split(':').first().trim(),
221                       expression.split(':').last().trim());
```

```
222     }
223   }
224 }
225
226 rule createPiece(container : GLOB!PiecewiseFunction,
227          var : String,
228          range : String,
229          value : String) {
230   to
231     glob_piece : GLOB!Piece,
232     glob_range : GLOB!Range,
233     glob_value : GLOB!Variable (name <- value)
234
235   do {
236
237     -- The case : a > x > b (2x greater than)
238     if (range.split('>').size() = 3)
239     {
240       glob_range.top    <- range.split('>').first();
241       glob_range.bottom <- range.split('>'). last();
242     }
243
244     -- The case : a < x < b (2x lesser than)
245     if (range.split('>').size() = 3)
246     {
247       glob_range.top    <- range.split('<'). last();
248       glob_range.bottom <- range.split('<').first();
249     }
250
251     -- The case : a < x
252     if (range.split('<').size() = 2 and
253       range.endsWith(var))
254     {
255       glob_range.top    <- 65565; -- (+inf)
256       glob_range.bottom <- range.split('<').first().toReal();
257     }
258
259     -- The case : a > x
260     if (range.split('>').size() = 2 and
261       range.endsWith(var))
262     {
263       glob_range.top    <- range.split('>').first().toReal();
264       glob_range.bottom <- -65565; -- (-inf)
265     }
266
267     -- The case : x < a
268     if (range.split('<').size() = 2 and
269       range.startsWith(var))
270     {
271       glob_range.top    <- range.split('<').last().toReal();
272       glob_range.bottom <- -65565; -- (-inf)
273     }
```

```
274
275     -- The case : x > a
276     if (range.split('>').size() = 2 and
277       range.startsWith(var))
278     {
279       glob_range.top    <- 65565; -- (+inf);
280       glob_range.bottom <- range.split('>').last().toReal();
281     }
282
283     glob_piece.interval <- glob_range;
284     glob_piece.value    <- glob_value;
285     container.pieces    <- glob_piece;
286   }
287 }
288
289 ------------------------------------------------------------------------

290 -- Creation of Synchronous Data Flow (SDF) behavior :
291 --    * Data Flows are defined by a process (calculus) and a rate
292 --      it should respond to. The rate defines the number of tokens that
293 --      the model should have at it's input to trigger the process.
294 --      An example would be a simple Fourier transform where 2^N samples
    should
295 --      be available for the process to run.
296 ------------------------------------------------------------------------

297
298 rule AtomicSDFModule {
299
300   from
301     sml_block : SysML!Block (sml_block.base_Class.isLeaf and
302                   sml_block.base_Class.getAppliedStereotypes()
303                             ->select(s | s.name = 'sdf').size() = 1)
304   to
305     glob_module      : GLOB!Module (name <- sml_block.base_Class.name),
306     glob_interface   : GLOB!Interface,
307     glob_sdfBehavior : GLOB!SDF
308     --glob_sdfProcess  : GLOB!Process
309   do {
310
311     -- First create all the metamodel links.
312     --glob_sdfBehavior.process <- glob_sdfProcess;
313     glob_module.interface <- glob_interface;
314     glob_module.behavior  <- glob_sdfBehavior;
315
316     -- Then, save this module to the list of all Modules
317     thisModule.allModules <- thisModule.allModules.append(glob_module);
318
319     -- For the moment, we allow only the use of single equations in SDF
    modules
320     -- Piecewise equations should be carefully studied before implementing
    this function.
```

```
321     for (constraint in sml_block.base_Class.ownedRule) {
322       if (constraint.specification.value.startsWith('PIECEWISE'))
323         thisModule.createPiecewiseFunction(glob_module, constraint);
324       else
325         thisModule.createSingleEquation(glob_module,constraint);
326     }
327   }
328 }
329
330 ------------------------------------------------------------------------
331 -- Creation of Discrete Event (DE) behavior :
332 --     * Discrete Event are fairly simple. Modules will react
333 --       exclusevely on events presented at their inputs.
334 ------------------------------------------------------------------------
335
336 rule AtomicDEModule {
337
338   from
339     sml_block : SysML!Block (sml_block.base_Class.isLeaf and
340                   sml_block.base_Class.getAppliedStereotypes()
341                             ->select(s | s.name = 'de').size() = 1)
342   to
343     glob_module     : GLOB!Module (name <- sml_block.base_Class.name),
344     glob_interface  : GLOB!Interface,
345     glob_deBehavior : GLOB!DE
346   do {
347
348     glob_module.interface <- glob_interface;
349     glob_module.behavior  <- glob_deBehavior;
350
351     -- Save this module to the list of all Modules
352     thisModule.allModules <- thisModule.allModules.append(glob_module);
353
354     -- I have absolutely no ideia of what else to treat here..
355   }
356 }
357
358 ------------------------------------------------------------------------
359 -- Composite Homogeneous Modules
360 --     * Communication is defined by the innerMoC
361 --     * Only modules of the same MoC are allowed inside an homogeneous
    behavior
362 --     * There is no need to adapt data, control nor time
363 --     * Internal parts are considered to be instances of atomic modules or
    other
364 --       composite modules
365 ------------------------------------------------------------------------
366
```

```
rule CompositeHomogeneousModules {
  from
    sml_block : SysML!Block (not sml_block.base_Class.isLeaf and
                    sml_block.base_Class.getAppliedStereotypes()
                            ->select(s | s.name = 'adaptor').size() = 0)
  to
    glob_module    : GLOB!Module (name <- sml_block.base_Class.name),
    glob_interface : GLOB!Interface(),
    glob_behavior  : GLOB!Homogeneous
  do {

    if (sml_block.base_Class.getAppliedStereotypes()->select(s | s.name = 'de
    ').size() = 1)
      glob_behavior.innerMoC <- 'de';
    if (sml_block.base_Class.getAppliedStereotypes()->select(s | s.name = '
    fsm').size() = 1)
      glob_behavior.innerMoC <- 'fsm';
    if (sml_block.base_Class.getAppliedStereotypes()->select(s | s.name = '
    sdf').size() = 1)
      glob_behavior.innerMoC <- 'sdf';
    if (sml_block.base_Class.getAppliedStereotypes()->select(s | s.name = 'ct
    ').size() = 1)
      glob_behavior.innerMoC <- 'ct';

    glob_module.interface <- glob_interface;
    glob_module.behavior  <- glob_behavior;
    glob_behavior.owner   <- glob_module;

    -- Save this module to the list of all Modules
    thisModule.allModules <- thisModule.allModules.append(glob_module);
  }
}

-------------------------------------------------------------------------------

-- Composite Heterogeneous Modules
--    * Communication is defined by the innerMoC
--    * Semantic adaptation happens in the interface :
--       - Data should be adapted here
--       - Control may change (e.g. SDF modules may wait for N tokens to
    arrive to be activated)
--       - The notion of time may change. (Does this makes any sense?)
--    * This is an special type of behavior namely Adaptor. The semantic of
    this module
--      is defined by the set of parameters specified such as the sampling
    timestep.
-------------------------------------------------------------------------------


rule CompositeHeterogeneousModules {
  from
    sml_block : SysML!Block (not sml_block.base_Class.isLeaf and
```

```
410                     sml_block.base_Class.getAppliedStereotypes()
411                             ->select(s | s.name = 'adaptor').size() = 1)
412   to
413     glob_module    : GLOB!Module (name <- sml_block.base_Class.name),
414     glob_interface : GLOB!Interface(),
415     glob_adaptor   : GLOB!Adaptor
416   do {
417     if (sml_block.base_Class.ownedRule.size() <> 0) {
418       if (sml_block.base_Class.ownedRule->first().specification.value.
      startsWith('ADAPTOR')) {
419         glob_adaptor.innerMoC <- sml_block.base_Class.ownedRule->first().
      specification.value
420                                 .split('FROM').last().split('TO').first().
      trim();
421         for (param in sml_block.base_Class.ownedRule->first().specification.
      value.split('PARAMS').last().split(',')) {
422           thisModule.createAdaptorParameter(glob_adaptor, param);
423         }
424       }
425     }
426
427     if (sml_block.base_Class.getAppliedStereotypes()->select(s | s.name = 'de
      ').size() = 1)
428       thisModule.createAdaptorParameter(glob_adaptor, 'adaptFrom : de');
429     if (sml_block.base_Class.getAppliedStereotypes()->select(s | s.name = '
      sdf').size() = 1)
430       thisModule.createAdaptorParameter(glob_adaptor, 'adaptFrom : sdf');
431     if (sml_block.base_Class.getAppliedStereotypes()->select(s | s.name = 'ct
      ').size() = 1)
432       thisModule.createAdaptorParameter(glob_adaptor, 'adaptFrom : ct');
433     if (sml_block.base_Class.getAppliedStereotypes()->select(s | s.name = '
      fsm').size() = 1)
434       thisModule.createAdaptorParameter(glob_adaptor, 'adaptFrom : fsm');
435
436     thisModule.createAdaptorParameter(glob_adaptor, 'adaptTo : ' +
437       sml_block.base_Class.getValue(sml_block.base_Class.
      getAppliedStereotypes()
438       ->select(s | s.name = 'adaptor').first(), 'adaptTo')
439     );
440
441     glob_adaptor.innerMoC <- sml_block.base_Class.getValue(sml_block.
      base_Class.getAppliedStereotypes()
442       ->select(s | s.name = 'adaptor').first(), 'adaptTo');
443
444     glob_module.interface <- glob_interface;
445     glob_module.behavior  <- glob_adaptor;
446     glob_adaptor.owner    <- glob_module;
447
448     -- Save this module to the list of all Modules
449     thisModule.allModules <- thisModule.allModules.append(glob_module);
450   }
451 }
```

```
452
453 rule createAdaptorParameter(adaptor : GLOB!Adaptor, param : String) {
454   to
455     glob_param : GLOB!Parameter (name  <- param.split(':').first().trim(),
456                   value <- param.split(':').last().trim())
457   do {
458     adaptor.params <- glob_param;
459   }
460 }
461
462 ------------------------------------------------------------------------
463 -- Interfaces definition
464 --    * Ports
465 --    * Connectors
466 --    * Instances?
467 ------------------------------------------------------------------------
468
469 rule Ports {
470   from
471     sml_port  : SysML!FlowPort
472   to
473     glob_port : GLOB!Port ( name      <- sml_port.base_Port.name,
474                 owner     <- thisModule.getModule(sml_port.base_Port.owner.
    name),
475                 direction <- sml_port.direction.toString())
476   do {
477     thisModule.getModule(sml_port.base_Port.owner.name).interface.ports <-
    glob_port;
478   }
479 }
480
481 --rule connector {
482 --   from
483 --     sml_connector : UML!Connector
484 --   to
485 --     glob_connection : GLOB!Connection(name  <- sml_connector.name,
486 --                       owner <- thisModule.getModule(sml_connector.owner.
    name))
487 --   do{
488 --     for (component in sml_connector.end) {
489 --     glob_connection.binds <-
490 --       thisModule.getModule(component.role.owner.name).interface.ports
491 --         ->select(p | p.name = component.role.name).first();
492 --     }
493 --     thisModule.getModule(sml_connector.owner.name).behavior.link <-
    glob_connection;
494 --   }
495 --}
496
497
```

```
498 rule connectors {
499     from
500         uml_connector : UML!Connector
501     do {
502         -- If the connector does not yet exists,
503         if (thisModule.getModule(uml_connector.owner.name).behavior.link
504             ->select(connector | connector.name = uml_connector.name).isEmpty
    ())
505         {
506       -- Create a connector inside the given architecture
507       thisModule.createConnector(uml_connector,thisModule.getModule(
    uml_connector.owner.name).behavior);
508     }
509         else -- If there is already a connector with the same name, then
    extend the existing one.
510         {
511       for (uml_port in uml_connector.end) {
512         thisModule.getModule(uml_connector.owner.name).interface;
513         if (thisModule.getModule(uml_connector.owner.name).behavior.link
            -- from existing connectors
514           ->select(connector | connector.name = uml_connector.name).first()
          -- select the current one (with the same name as the uml_connector)
515           .binds->select( port | port.name = uml_port.role.name).isEmpty() )
          -- and verify that there is no port with the same name as the
    uml_connector ends
516         {
517           thisModule.getModule(uml_connector.owner.name).behavior.link
518             ->select(connector | connector.name = uml_connector.name).first()
    .binds
519               <- thisModule.getModule(uml_port.role.owner.name).interface.
    ports
520                 ->select(vha_port | vha_port.name = uml_port.role.name);
521         }
522       }
523         }
524     }
525 }
526
527 rule createConnector(uml_connector : UML!Connector, glob_behavior : GLOB!
    Behavior) {
528   to
529     glob_connector : GLOB!Connection
530   do {
531     -- Set the name of the connector
532     glob_connector.name  <- uml_connector.name;
533     glob_connector.owner <- thisModule.getModule(uml_connector.owner.name);
534     -- Bind the two ends to corresponding ports
535     for (uml_port in uml_connector.end) {
536       glob_connector.binds <- thisModule.getModule(uml_port.role.owner.name).
    interface.ports
537                         ->select(port | port.name = uml_port.role.name);
538
```

```
539       -- And set the type accordingly
540       --glob_connector.type <- uml_port.role.type.name;
541       --if (uml_port.role.type.name = 'Electrical')
542       --  vha_connector.nature <- 'Terminal';
543       --else if (uml_port.role.type.name = 'Signal')
544       --  vha_connector.nature <- 'Signal';
545       --else -- Voltage, Current, Force, etc ..
546       --  vha_connector.nature <- 'Quantity';
547
548     }
549
550     -- Add the connector to the corresponding model's body (architecture)
551         glob_behavior.link <- glob_connector;
552     for (p in glob_connector.binds ) {
553       p.bindedBy <- glob_connector;
554     }
555   }
556
557 }
558
559 rule Instances {
560   from
561     uml_composition : UML!Association
562 --  using {
563 --    --TODO: Solve this bug : these variables are beeing created before the
      list allModules
564 --    owner : GLOB!Module = thisModule.getModule(uml_composition.endType->
      last().name);
565 --    owned : GLOB!Module = thisModule.getModule(uml_composition.endType->
      first().name);
566 --  }
567   do {
568     -- Attach the child module to the parent by the relation "instances"
569     thisModule.getModule(uml_composition.endType->last().name).behavior.
      instances <-
570       thisModule.getModule(uml_composition.endType->first().name);
571     -- The other way around : Make the child know its parent through the
      relation "parent"
572     thisModule.getModule(uml_composition.endType->first().name).parent <-
573       thisModule.getModule(uml_composition.endType->last().name);
574 --    owner.behavior.instances <- owned;
575   }
576 }
```

C.2 ATL : From Intermediary Representation to SystemC-AMS

```
1  -- @path GLOB=/Metamodels/glob.ecore
2  -- @path SC=/Metamodels/systemc.ecore
3
4  module glob2systemc;
5  create OUT : SC from IN : GLOB;
6
7  -- Declare all objects for further manipulation
8  helper def : allModules : Sequence(SC!Module) = Sequence{};
9  helper def : signals    : Sequence(SC!Signal) = Sequence{};
10 helper def : ports      : Sequence(SC!Port)   = Sequence{};
11
12 -- Usefull functions
13 -- aModule : This function takes a named object in its input and provide
14 --   the corresponding module from the Sequence allModules that maches
15 --   the object name.
16 helper def : aModule(o : OclAny) : SC!Module =
17         if (o.oclIsUndefined()) then
18                 OclUndefined else
19         thisModule.allModules->
20                 select(e | e.name = o.name).first() endif;
21
22 -- SystemC is a C++ library for the simulation of systems. It comes with a
23 -- discrete event simulation kernel. It works much similar to a VHDL
       simulator
24 -- SystemC-AMS, the Analog and Mixed-Signal extension, adds the notion of
25 -- three different models of computation on top of SystemC. These are :
26 -- (a) Timed Data Flow (TDF), much similar to a Synchronous Data Flow (SDF)
       but
27 -- tokens are tagged with time values.
28 -- (b) Linear Signal Flow LSF, very close to what is done in simulink. This
       is
29 -- continuous time model of computation with a periodic or dynamic timestep
       solver.
30 -- (c) Electrical Linear Networks, witch is inspired in SPICE netlists. It
       uses
31 -- the topology of the system (circuit) to deduce the differential equations
       from it.
32 -- The ELN model of computation uses the same numerical solver as LSF.
33
34 -- Here I'll detail each one separetly.
35
36 ---------------------------------- TDF ------------------------------------
37 rule abstractTdfModules {
38   from
39     glob_module : GLOB!Module(glob_module.behavior.oclIsTypeOf(GLOB!SDF))
40   do {
41     if (glob_module.behavior.equations.size() <> 0)
42     {
```

```
43      if (glob_module.behavior.equations.first().name.split('=').last().trim
    () = 'x\'integ')
44        thisModule.tdfAtomicIntegral(glob_module);
45      else if (glob_module.behavior.equations.first().name.split('=').last().
    trim() = 'x\'dot')
46        thisModule.tdfAtomicDerivative(glob_module);
47      else
48        thisModule.tdfModules(glob_module);
49    }
50    else
51    {
52      thisModule.tdfModules(glob_module);
53    }
54  }
55 }
56
57 rule tdfModules(glob_module : GLOB!Module)  {
58  to
59    sc_module : SC!TimedDataFlowModule (name <- glob_module.name),
60    proc_attributes : SC!Process(name <- 'set_attributes',returnType <- 'void
    '),
61    --proc_initialize : SC!Process(name <- 'initialize', returnType <- 'void'
    ),
62    proc_processing : SC!Process(name <- 'processing', returnType <- 'void')
63
64  do {
65
66    -- Save this module in a Sequence :
67    thisModule.allModules <- Sequence{thisModule.allModules, sc_module}->
    flatten();
68
69    --glob_module.debug();
70    --glob_module.parent.debug();
71    -- This complicated code is supposed to propagate the adaptor information
    , such as the type of adaptor
72    -- and timestep to the inner modules inside an adaptor interface.
73    if (glob_module.parent.behavior.oclIsTypeOf(GLOB!Adaptor))
74      if(glob_module.parent.behavior.params->select(p | p.name = 'adaptFrom')
    .first().value = 'de') {
75        for (port in glob_module.parent.interface.ports) {
76          if (port.bindedBy.binds->select(p | p <> port).first().owner =
    glob_module)
77            thisModule.createPortInnerTDFOuterDE(sc_module, port.bindedBy.
    binds->select(p | p <> port).first());
78        }
79        sc_module.ports.first().timestepValue <-
80          glob_module.parent.behavior.params->select(p | p.name = 'timestep')
    .first().value.split(' ').first().toReal();
81        sc_module.ports.first().timestepScale <-
82          glob_module.parent.behavior.params->select(p | p.name = 'timestep')
    .first().value.split(' ').last ().toUpper().trim();
83
```

```
84          proc_attributes.codeLines <- sc_module.ports.first().name + '.
    set_timestep(' +
85            sc_module.ports.first().timestepValue.toString() + ',SC_' +
86            sc_module.ports.first().timestepScale +
87            ');';
88        }
89
90      for (port in glob_module.interface.ports->select(p | p.name <> sc_module.
    ports.first().name)) {
91        thisModule.createTDFPorts(sc_module,port,glob_module.behavior);
92      }
93
94      for (equation in glob_module.behavior.equations) {
95        proc_processing.codeLines <- equation.name;
96      }
97
98      for (equation in glob_module.behavior.piecewiseFunctions) {
99        thisModule.createPiecewiseEquation(proc_processing,equation);
100     }
101
102     sc_module.process <- proc_attributes;
103     --sc_module.process <- proc_initialize;
104     sc_module.process <- proc_processing;
105
106   }
107 }
108
109 rule createPiecewiseEquation(container : SC!Process, equation : GLOB!
    PiecewiseFunction) {
110   to
111     inputVariable  : SC!Variable (name <- 'inputVar' , type <- 'double',
    initialValue <- equation.argument.name  + '.read(0)'),
112     outputVariable : SC!Variable (name <- 'outputVar', type <- 'double'),
113     sc_if : SC!If(codeLines <- outputVariable.name + ' = ' + equation.pieces.
    first().value.name),
114     sc_else : SC!Else(codeLines <- outputVariable.name + ' = ' +
    inputVariable.name)
115
116   do {
117     container.variables <-  inputVariable;
118     container.variables <- outputVariable;
119     --container.codeLines <-  inputVariable.name +  '=' + equation.argument.
    name  + '.read(0)';
120     if (equation.pieces.first().interval.top >= 65565.0)
121       sc_if.condition <- inputVariable.name + ' > ' + equation.pieces.first()
    .interval.bottom;
122     else if (equation.pieces.first().interval.bottom <= -65565.0)
123       sc_if.condition <- inputVariable.name + ' < ' + equation.pieces.first()
    .interval.top;
124     else
125       sc_if.condition <- inputVariable.name + ' > ' + equation.pieces.first()
    .interval.bottom + '&&'
```

```
126                         + inputVariable.name + ' < ' + equation.pieces.first()
    .interval.top;
127     -- Add the If clause to the process
128     container.codePiece <- sc_if;
129     -- And generate the other cases in the form of 'else if' clauses
130     for (piece in equation.pieces.excluding(equation.pieces.first())) {
131       thisModule.createPiece(sc_if,piece);
132     }
133
134     -- And add the final 'else' for the default value.
135     sc_if.elseClause <- sc_else;
136     container.codeLines <- equation.name + '= ' + outputVariable.name;
137
138   }
139 }
140
141 rule createPiece(container : SC!If, piece : GLOB!Piece) {
142   to
143     sc_elseIf : SC!ElseIf (codeLines <- 'outputVar = ' + piece.value.name)
144   do {
145     if (piece.interval.top >= 65565.0)
146       sc_elseIf.condition <- 'inputVar > ' + piece.interval.bottom;
147     else if (piece.interval.bottom <= -65565.0)
148       sc_elseIf.condition <- 'inputVar < ' + piece.interval.top;
149
150     container.elseIfClause <- sc_elseIf;
151   }
152 }
153
154 -- The integrator block in the TDF formalism is actually only an interface
155 -- to the continuous-time domain LSF.
156 rule tdfAtomicIntegral(glob_module : GLOB!Module) {
157   to
158     -- TDF interface
159     TDFinterface      : SC!Module  (name <- glob_module.name),
160     TDFportIn       : SC!TDF_Port(name <- glob_module.interface.ports->select
    (p | p.direction.toString() = 'in').first().name,
161                     direction <- 'in', rate <- 1,type <- 'double',owner <-
    TDFinterface),
162     TDFportOut        : SC!TDF_Port(name <- glob_module.interface.ports->
    select(p | p.direction.toString() = 'out').first().name,
163                     direction <- 'out',rate <- 1,type <- 'double',owner <-
    TDFinterface),
164     -- Input adaptor
165     LSFAdaptorIn      : SC!Source  (name <- 'tdf2lsf'),
166     LSFAdaptInPortIn  : SC!TDF_Port(name <- 'inp' ,owner <- LSFAdaptorIn),
167     LSFAdaptInPortOut : SC!LSF_Port(name <- 'y'   ,owner <- LSFAdaptorIn),
168     -- Output adaptor
169     LSFAdaptorOut     : SC!Sink    (name <- 'lsf2tdf'),
170     LSFAdaptOutPortIn : SC!LSF_Port(name <- 'x'   ,owner <- LSFAdaptorOut),
171     LSFAdaptOutPortOut: SC!TDF_Port(name <- 'outp',owner <- LSFAdaptorOut),
172     -- The heart of it : the integrator
```

```
173     LSFIntegrator     : SC!Integ   (name <- 'LSF_integrator'),
174     LSFPortIn         : SC!LSF_Port(name <- 'x',owner <- LSFIntegrator),
175     LSFPortOut        : SC!LSF_Port(name <- 'y',owner <- LSFIntegrator),
176     -- Internal connections
177     SignalIn        : SC!TDFHierarchicalSignal(name <- 'tdf2lsfSignal',type
        <- 'double'),
178     SignalOut         : SC!TDFHierarchicalSignal(name <- 'lsf2tdfSignal',type
         <- 'double'),
179     LSFSignalIn       : SC!LSF_Signal(name <- 'internal_x'),
180     LSFSignalOut      : SC!LSF_Signal(name <- 'internal_y')
181
182   do {
183
184     -- Save this module in a Sequence :
185     thisModule.allModules <- Sequence{thisModule.allModules, TDFinterface}->
        flatten();
186
187     --for (port in glob_module.interface.ports) {
188     --  thisModule.createTDFPorts(TDFinterface,port,glob_module.behavior);
189     --}
190
191     -- TDF interface bindings
192     TDFinterface.ports <- TDFportIn;
193     TDFinterface.ports <- TDFportOut;
194
195     -- Adaptors
196     LSFAdaptorIn.ports  <- LSFAdaptInPortIn;
197     LSFAdaptorIn.ports  <- LSFAdaptInPortOut;
198
199     LSFAdaptorOut.ports <- LSFAdaptOutPortIn;
200     LSFAdaptorOut.ports <- LSFAdaptOutPortOut;
201
202     -- LSF Integrator bindings
203     LSFIntegrator.ports <- LSFPortIn;
204     LSFIntegrator.ports <- LSFPortOut;
205
206     -- Internal bindings : instances
207     TDFinterface.instances <- LSFIntegrator;
208     TDFinterface.instances <- LSFAdaptorIn ;
209     TDFinterface.instances <- LSFAdaptorOut;
210
211     -- Internal bindings : connections
212     -- Input port bindings
213     -- TODO : Check if there are multiple input ports, and scream if it is
        the case
214     SignalIn .binds <- TDFportIn;
215     SignalIn .binds <- LSFAdaptInPortIn;
216     -- Main LSF block bindings
217     LSFSignalIn.binds <- LSFAdaptInPortOut;
218     LSFSignalIn.binds <- LSFPortIn;
219     LSFSignalOut.binds <- LSFPortOut;
220     LSFSignalOut.binds <- LSFAdaptOutPortIn;
```

```
221     -- Output port bindings
222     -- TODO : Check if there are multiple output ports, and scream if it is
        the case
223     SignalOut.binds <- TDFportOut;
224     SignalOut.binds <- LSFAdaptOutPortOut;
225
226     -- All signals belong to the TDFinterface
227     TDFinterface.signals <- SignalIn;
228     TDFinterface.signals <- SignalOut;
229     TDFinterface.signals <- LSFSignalIn;
230     TDFinterface.signals <- LSFSignalOut;
231
232   }
233 }
234
235 -- The integrator block in the TDF formalism is actually only an interface
236 -- to the continuous-time domain LSF.
237 rule tdfAtomicDerivative(glob_module : GLOB!Module) {
238   to
239     TDFinterface  : SC!TimedDataFlowModule (name <- 'tdfInterface' +
        glob_module.name),
240     LSFDerivator  : SC!Dot(name <- glob_module.name),
241     LSFPortIn     : SC!LSF_Port(name <- 'x'),
242     LSFPortOut    : SC!LSF_Port(name <- 'y'),
243     LSFSignalIn   : SC!LSF_Signal(name <- 'internal_x'),
244     LSFSignalOut  : SC!LSF_Signal(name <- 'internal_y'),
245     LSFAdaptorIn  : SC!Source(name <- 'tdf2lsf' + glob_module.name + 'Adaptor
        '),
246     LSFAdaptorOut : SC!Sink  (name <- 'lsf2tdf' + glob_module.name + 'Adaptor
        '),
247     SignalIn    : SC!Signal(name <- 'tdf2lsfSignal'),
248     SignalOut     : SC!Signal(name <- 'lsf2tdfSignal')
249
250   do {
251
252     -- Save this module in a Sequence :
253     thisModule.allModules <- Sequence{thisModule.allModules, TDFinterface}->
        flatten();
254
255     LSFDerivator.ports <- LSFPortIn;
256     LSFDerivator.ports <- LSFPortOut;
257
258
259     TDFinterface.instances <- LSFDerivator;
260     TDFinterface.instances <- LSFAdaptorIn ;
261     TDFinterface.instances <- LSFAdaptorOut;
262
263     -- Internal connections
264     SignalIn .binds <- LSFAdaptorIn.ports->select(p | p.name = 'y');
265     SignalIn .binds <- LSFDerivator.ports->select(p | p.name = 'x');
266     SignalOut.binds <- LSFDerivator.ports->select(p | p.name = 'y');
267     SignalOut.binds <- LSFAdaptorOut.ports->select(p | p.name = 'x');
```

```
268     TDFinterface.signals <- SignalIn;
269     TDFinterface.signals <- SignalOut;
270
271   }
272 }
273
274
275 ---------------------------------- LSF ----------------------------------
276 rule abstractRuleLSFatomic {
277     from
278     glob_module : GLOB!Module(glob_module.behavior.oclIsTypeOf(GLOB!CT))
279   do {
280     if (glob_module.behavior.equations.size() <> 0) {
281       if (glob_module.behavior.equations.first().name.split('=').last().trim
      () =
282           'x\'integ') thisModule.lsfAtomicIntegrator(glob_module);
283       else if (glob_module.behavior.equations.first().name.split('=').last().
      trim() =
284           'x\'dot') thisModule.lsfAtomicDerivator(glob_module);
285       else
286         thisModule.lsfCustomModules(glob_module);
287     }
288     else
289     {
290       thisModule.lsfCustomModules(glob_module);
291     }
292   }
293 }
294
295 rule lsfCustomModules(glob_block : GLOB!Module) {
296   --from
297   --   glob_block : GLOB!Module(glob_block.behavior.oclIsTypeOf(GLOB!CT))
298   to
299     sc_module : SC!LinearSignalFlowModule (name <- glob_block.name)
300   do {
301     -- Save this module idn a Sequence :
302     thisModule.allModules <- Sequence{thisModule.allModules, sc_module}->
      flatten();
303
304     for (port in glob_block.interface.ports) {
305       thisModule.createPorts(sc_module,port,glob_block.behavior);
306     }
307
308     for (equations in glob_block.behavior.equations) {
309       'LSF equations - Yet to do'.debug();
310     }
311   }
312 }
313
314
315 rule lsfAtomicIntegrator(glob_module : GLOB!Module) {
316   to
```

```
317     sc_lsf_integrator : SC!Integ (name <- glob_module.name)
318   do {
319     -- Save this module in a Sequence :
320     thisModule.allModules <- Sequence{thisModule.allModules,
        sc_lsf_integrator}->flatten();
321
322     for (port in glob_module.interface.ports) {
323       thisModule.createPorts(sc_lsf_integrator,port,glob_module.behavior);
324     }
325   }
326 }
327
328 rule lsfAtomicDerivator(glob_module : GLOB!Module) {
329   to
330     sc_lsf_derivator : SC!Dot (name <- glob_module.name)
331   do {
332     -- Save this module in a Sequence :
333     thisModule.allModules <- Sequence{thisModule.allModules, sc_lsf_derivator
        }->flatten();
334
335     for (port in glob_module.interface.ports) {
336       thisModule.createPorts(sc_lsf_derivator,port,glob_module.behavior);
337     }
338   }
339 }
340
341 rule fsmModules {
342     from
343     glob_block : GLOB!Module(glob_block.behavior.oclIsTypeOf(GLOB!FSM))
344     to
345     sc_module : SC!Module (name <- glob_block.name)
346     do {
347     -- Save this module in a Sequence :
348     thisModule.allModules <- Sequence{thisModule.allModules, sc_module}->
        flatten();
349
350     for (port in glob_block.interface.ports) {
351       thisModule.createDEPorts(sc_module,port,glob_block.behavior);
352     }
353     }
354 }
355
356 ------------------------------ FSM ------------------------------
357 rule StateMachine {
358   from
359     glob_stateMachine : GLOB!StateMachine
360   to
361     -- Create the objects where we'll save the states
362     sc_enum : SC!Enumeration (
363       name <- 'stateType'),
364     -- Instances of Enumeration
365     sc_nst  : SC!Signal(
```

```
366       name <- 'next_state',
367       type <- 'stateType'),
368     sc_cst  : SC!Signal(
369       name <- 'current_state',
370       type <- 'stateType'),
371     -- A switch element
372     sc_swch : SC!Swich(
373       var <- sc_cst.name),
374     sc_ev_cst : SC!Event(name <- 'current_state'),
375     sc_ev_nst : SC!Event(name <- 'next_state'),
376     -- Process : Next State logic
377     sc_proc : SC!Method(
378       name  <- 'next_state_op_logic',
379       triggers <- sc_ev_cst,
380       codePiece <- sc_swch),
381     -- Process : Update state
382     --code_frag : SC!CodeFragment(codeLines <- 'current_state = next_state;')
    ,
383     sc_proc_set_next : SC!Method(
384       name  <- 'set_next_state',
385       triggers <- sc_ev_nst,
386       codeLines <- 'current_state = next_state')
387   do {
388     -- TODO: Get rid of the "thisModule.aModule(uml_stm.owner)",
389     --   this should be a simple variable "m"
390     -- Should create a trigger for every input of the state machine.
391     for (p in thisModule.aModule(glob_stateMachine.owner.owner).ports) {
392       if (p.direction.toString() = 'in') {
393         thisModule.createTrigger(sc_proc,p);
394       }
395     }
396     thisModule.aModule(glob_stateMachine.owner.owner).codePiece <- sc_enum;
397     thisModule.aModule(glob_stateMachine.owner.owner).signals   <- sc_cst;
398     thisModule.aModule(glob_stateMachine.owner.owner).signals   <- sc_nst;
399     thisModule.aModule(glob_stateMachine.owner.owner).process   <- sc_proc;
400     thisModule.aModule(glob_stateMachine.owner.owner).process   <-
    sc_proc_set_next;
401     for (s in glob_stateMachine.states) {
402       thisModule.createState(s, sc_swch, sc_enum);
403     }
404   }
405 }
406
407 rule createTrigger(process : SC!Process, p : SC!Port) {
408   to
409     sc_ev : SC!Event (name <- p.name)
410   do {
411     process.triggers <- sc_ev;
412   }
413 }
414
```

```
415 -- For every State, we have to add a Case to the Swhitch and a literal to the
        enumeration
416 rule createState(glob_state : GLOB!State, swch : SC!Swich, enum : SC!
        Enumeration) {
417   to
418     --sc_code : SC!CodeFragment(line <- glob_state.invariant),
419     sc_case : SC!Case (
420       item <- glob_state.name,
421     --  code <- sc_code),
422       codeLines <- glob_state.invariant),
423     sc_label : SC!Label (name <- glob_state.name)
424   do {
425
426     for (t in glob_state.transition) {
427       thisModule.createIf(sc_case,t);
428     }
429     swch.cases <- sc_case;
430     enum.labels <- sc_label;
431   }
432 }
433
434 -- Create the if clause corresponding to the transition
435 rule createIf(c : SC!Case, t : GLOB!Transition) {
436   to
437 --    code_fragment : SC!CodeFragment (line <- 'next_state = ' + t.target.
        name),
438     sc_if : SC!If (
439       condition <- t.guard.name.regexReplaceAll('=', '=='),
440 --      code <- code_fragment)
441       codeLines <- 'next_state = ' + t.target.name)
442   do {
443     c.codePiece <- sc_if;
444   }
445 }
446
447 ------------------------------- DE --------------------------------
448 rule deModules {
449     from
450     glob_block : GLOB!Module(glob_block.behavior.oclIsTypeOf(GLOB!DE))
451     to
452     sc_module : SC!Module (name <- glob_block.name)
453     do {
454     -- Save this module in a Sequence :
455     thisModule.allModules <- Sequence{thisModule.allModules, sc_module}->
        flatten();
456
457     for (port in glob_block.interface.ports) {
458       thisModule.createDEPorts(sc_module,port,glob_block.behavior);
459     }
460     }
461 }
462
```

```
463
464 ------------------------------ Composites ------------------------------
465 -- Heterogeneous Modules are interface modules that allow different models of
466 -- computation to comunicate. Usually, an heterogeneous module is contained
467 -- within a MoC and defines a different MoC at it's interior.
468 rule HeterogeneousModules {
469     from
470     glob_block : GLOB!Module(glob_block.behavior.oclIsKindOf(GLOB!
        Heterogeneous))-- or
471     to
472     sc_module : SC!Module(name <- glob_block.name)
473     do {
474
475       -- Save this module in a Sequence :
476     thisModule.allModules <- Sequence{thisModule.allModules, sc_module}->
        flatten();
477
478     if(glob_block.behavior.params->select(p | p.name = 'adaptFrom').first().
        value = 'de') {
479         for (port in glob_block.interface.ports) {
480         thisModule.createDEPorts(sc_module, port, glob_block.behavior);
481         }
482     }
483     if(glob_block.behavior.params->select(p | p.name = 'adaptFrom').first().
        value = 'sdf') {
484         for (port in glob_block.interface.ports) {
485         thisModule.createTDFPorts(sc_module, port, glob_block.behavior);
486         }
487     }
488
489       for (instance in glob_block.behavior.instances) {
490       sc_module.instances <- thisModule.aModule(instance);
491     }
492     }
493 }
494
495 -- Depending on the MoC, we should create apropriate ports.
496 -- DE Modules use simple DE ports, witch differs from TDF and LSF ports
497 rule HomogeneousModules {
498     from
499     glob_block : GLOB!Module(glob_block.behavior.oclIsKindOf(GLOB!Homogeneous
        ))-- or
500     to
501     sc_module : SC!Module(name <- glob_block.name)
502     do {
503     -- Save this module in a Sequence :
504     thisModule.allModules <- Sequence{thisModule.allModules, sc_module}->
        flatten();
505
506     if(glob_block.behavior.innerMoC = 'de') {
507       for (port in glob_block.interface.ports) {
508         thisModule.createDEPorts(sc_module,port,glob_block.behavior);
```

```
509       }
510     }
511     if(glob_block.behavior.innerMoC = 'tdf') {
512       for (port in glob_block.interface.ports) {
513         thisModule.createDEPorts(sc_module,port,glob_block.behavior);
514       }
515     }
516
517     for (instance in glob_block.behavior.instances) {
518       sc_module.instances <- thisModule.aModule(instance);
519     }
520
521     -- TODO
522     for (signal in glob_block.behavior.link) {
523       signal.binds;
524     }
525   }
526 }
527
528 -- Create all ports for the Composed Modules
529 rule createDEPorts(sc_module : SC!Module, port : GLOB!Port, behavior : GLOB!
      Behavior) {
530   to
531     sc_port : SC!Port (
532       name <- port.name,
533       type <- 'double',--glob_port.base_Port.type.name,
534       direction <- port.direction,
535       owner <- sc_module
536     )
537   do {
538
539     sc_module.ports <- sc_port;
540   }
541 }
542
543 rule createTDFPorts(sc_module : SC!Module, port : GLOB!Port, behavior : GLOB!
      Behavior) {
544   to
545     sc_port : SC!TDF_Port (name <- port.name,type <- 'double',direction <-
      port.direction,owner <- sc_module)
546   do {
547     sc_module.ports <- sc_port;
548   }
549 }
550
551
552 -- Signals :
553 --   By construction, signals only exist in composite modules
554 --   so we must check the inner model of computation to instantiate
555 --   the proper signal. TDF modules use TDF signals, LSF modules use
556 --   LSF signals, etc ..
557 rule DESignals {
```

```
558   from
559     glob_connection : GLOB!Connection(glob_connection.owner.behavior.innerMoC
      .toString() = 'de')
560   to
561     sc_signal : SC!Signal (name <- glob_connection.name,type <- 'double')
562   do {
563
564     -- signal owner
565     thisModule.aModule(glob_connection.owner).signals <- sc_signal;
566
567     -- Binding new connections
568     --'--------------------'.debug();
569     --glob_connection.owner.debug();
570     --glob_connection.binds.debug();
571     for (port in glob_connection.binds) {
572       --if (port.owner = glob_connection.owner)
573     --  ('port owner : ' + port.owner.name).debug();
574     --  ('connection owner : ' + glob_connection.owner.name).debug();
575       sc_signal.binds <- thisModule.aModule(port.owner).ports->select(p | p.
      name = port.name);
576     }
577   }
578 }
579
580 rule SDFSignals {
581   from
582     glob_connection : GLOB!Connection(glob_connection.owner.behavior.innerMoC
      .toString() = 'sdf')
583   do {
584
585     -- Binding new connections
586     if (glob_connection.binds->select(p | p.owner = glob_connection.owner).
      size() <> 0)
587       thisModule.createHierarchicalSignalTDF(glob_connection);
588     else
589       thisModule.createPureSignalTDF(glob_connection);
590
591   }
592 }
593
594 rule createPureSignalTDF(glob_connection : GLOB!Connection) {
595   to
596     sc_signal : SC!TDF_Signal (name <- glob_connection.name,type <- 'double')
597   do {
598     thisModule.aModule(glob_connection.owner).signals <- sc_signal;
599     for (port in glob_connection.binds) {
600       sc_signal.binds <- thisModule.aModule(port.owner).ports->select(p | p.
      name = port.name);
601     }
602   }
603 }
604
```

```
605 rule createHierarchicalSignalTDF(glob_connection : GLOB!Connection) {
606   to
607     sc_signal : SC!TDFHierarchicalSignal (name <- glob_connection.name,type
      <- 'double')
608   do {
609     thisModule.aModule(glob_connection.owner).signals <- sc_signal;
610     for (port in glob_connection.binds) {
611       sc_signal.binds <- thisModule.aModule(port.owner).ports->select(p | p.
      name = port.name);
612     }
613   }
614 }
615
616 rule CTSignals {
617   from
618     glob_connection : GLOB!Connection(glob_connection.owner.behavior.innerMoC
      .toString() = 'ct')
619   to
620     sc_signal : SC!LSF_Signal (name <- glob_connection.name, type <- 'double'
      )
621   do {
622
623     -- signal owner
624     thisModule.aModule(glob_connection.owner).signals <- sc_signal;
625
626     -- Binding new connections
627     for (port in glob_connection.binds) {
628       sc_signal.binds <- thisModule.aModule(port.owner).ports->select(p | p.
      name = port.name);
629     }
630   }
631 }
632
633 ---------------------------------- ELN ----------------------------------
634 -- Yet to do
635
636 ---------------------------------- Port Adaptors  ----------------------------------
637 rule createPortInnerTDFOuterDE(sc_module :SC!Module, port : GLOB!Port) {
638   to
639     sc_port : SC!TDF_DE_Adaptor(name <- port.name,
640                   direction <- port.direction,
641                   type <- 'double',
642                   owner <- sc_module)
643                  --bindsTo <- sc_port)
644   do {
645 --    sc_port.hasAdaptor <- sc_adaptor;
646 --    sc_module.adaptors <- sc_adaptor;
647     sc_module.ports <- sc_port;
648   }
649 }
```

C.3 ATL : From Intermediary Representation to VHDL-AMS

```
1  -- @atlcompiler atl2006
2
3  -- @path VHDL=/Metamodels/vhdl-ams.ecore
4  -- @path GLOB=/Metamodels/glob.ecore
5
6  module glob2vhdl;
7
8  create OUT : VHDL from IN : GLOB;
9
10 -- Declare all objects for further manipulation
11 helper def : allEntities : Sequence(VHDL!Entity) = Sequence{};
12
13 -- Usefull functions
14 -- aModule : This function takes a named object in its input and provide
15 --   the corresponding module from the Sequence allModules that maches
16 --   the object name.
17 helper def : getEntity(o : OclAny) : VHDL!Entity =
18   if (o.oclIsUndefined()) then OclUndefined else
19     thisModule.allEntities->select(e | e.name = o.name).first() endif;
20
21
22 rule Entity {
23   from
24     glob_interface : GLOB!Interface
25   to
26     vhdl_entity : VHDL!Entity (name <- glob_interface.owner.name)
27   do {
28     thisModule.allEntities <- Sequence{thisModule.allEntities, vhdl_entity}->
       flatten();
29     for (port in glob_interface.ports) {
30       thisModule.createPort(port,vhdl_entity);
31     }
32   }
33
34 }
35
36
37 rule ArchitectureHierarichical {
38   from
39     glob_behavior : GLOB!Composite
40   to
41     vhdl_archi    : VHDL!Architecture ()
42   do {
43     vhdl_archi.ofEntity       <- thisModule.getEntity(glob_behavior.owner);
44     thisModule.getEntity(glob_behavior.owner).architectures <- vhdl_archi ;
45     for (instance in glob_behavior.instances) {
46       self.createComponent(vhdl_archi,instance);
47     }
48
```

```
49    }
50
51  }
52
53  rule ArchitectureContinuousTime {
54    from
55      glob_behavior : GLOB!CT
56    to
57      vhdl_archi   : VHDL!Architecture ()
58    do {
59      vhdl_archi.ofEntity       <- thisModule.getEntity(glob_behavior.owner);
60      thisModule.getEntity(glob_behavior.owner).architectures <- vhdl_archi ;
61      for (equation in  glob_behavior.equations) {
62        --if (equation.pieces->size() > 1)
63        --  thisModule.createConditionedEquation(equation,vha_archi);
64        --else
65        thisModule.createEquation(equation,vhdl_archi);
66        -- If using the adaptor language, create a 'use' equation
67  --      if (e.specification.value.toString()->startsWith('ADAPTOR'))
68  --        thisModule.createAdaptor(e.specification.value,vha_archi);
69      }
70
71      for (equation in glob_behavior.piecewiseFunctions) {
72        thisModule.createConditionedEquation(equation, vhdl_archi);
73      }
74  --    if (e.oclIsTypeOf(SYSML!Property)) {
75  --      -- If it has a default value, create a generic
76  --      if (not e.defaultValue.oclIsUndefined())
77  --        thisModule.createGeneric(e,vha_entity);
78  --      -- If not, create a quantity
79  --      else
80  --        thisModule.createQuantity(e,vha_archi);
81  --    }
82
83    }
84
85  }
86
87  rule ArchitectureSynchronousDataFlow {
88    from
89      glob_behavior : GLOB!SDF
90    to
91      vhdl_archi   : VHDL!Architecture ()
92    do {
93      vhdl_archi.ofEntity       <- thisModule.getEntity(glob_behavior.owner);
94      thisModule.getEntity(glob_behavior.owner).architectures <- vhdl_archi ;
95    }
96  }
97
98  rule ArchitectureStateMachines {
99    from
100     glob_behavior : GLOB!FSM
```

```
101   to
102     vhdl_archi   : VHDL!Architecture ()
103   do {
104     vhdl_archi.ofEntity       <- thisModule.getEntity(glob_behavior.owner);
105     thisModule.getEntity(glob_behavior.owner).architectures <- vhdl_archi ;
106   }
107 }
108
109
110 --rule block {
111 --   from
112 --     glob_module : GLOB!Module(not glob_module.behavior.oclIsKindOf(GLOB!
      Composite))
113 --   to
114 --     --vha_model  : VHDL!Model,
115 --     vha_entity : VHDL!Entity (name <- glob_module.name), --.replaceAll(' ',
       '_')
116 --     vha_archi  : VHDL!Architecture (name <- 'behavior')
117 --   do {
118 --     -- Save this model in a sequence
119 --     --thisModule.allEntities <- Sequence{thisModule.allEntities, vha_entity
      }->flatten();
120 --
121 --     --vha_model.entity       <- vha_entity;
122 --     --vha_model.architecture <- vha_archi ;
123 --     --vha_archi.ofEntity       <- vha_entity;
124 --     --vha_entity.architectures <- vha_archi ;
125 --
126 --     if (glob_module.behavior.oclIsTypeOf(GLOB!CT) or glob_module.behavior.
      oclIsTypeOf(GLOB!SDF) ) {
127 --       for (equation in  glob_module.behavior.equations) {
128 --         --if (equation.pieces->size() > 1)
129 --         --  thisModule.createConditionedEquation(equation,vha_archi);
130 --         --else
131 --         thisModule.createEquation(equation,vha_archi);
132 --         -- If using the adaptor language, create a 'use' equation
133 --         if (e.specification.value.toString()->startsWith('ADAPTOR'))
134 --           thisModule.createAdaptor(e.specification.value,vha_archi);
135 --
136 --       }
137 --       for (equation in glob_module.behavior.piecewiseFunctions) {
138 --         thisModule.createConditionedEquation(equation, vha_archi);
139 --       }
140 --       if (e.oclIsTypeOf(SYSML!Property)) {
141 --         -- If it has a default value, create a generic
142 --         if (not e.defaultValue.oclIsUndefined())
143 --           thisModule.createGeneric(e,vha_entity);
144 --         -- If not, create a quantity
145 --         else
146 --           thisModule.createQuantity(e,vha_archi);
147 --       }
148 --
```

```
149 --      for (port in glob_module.interface.ports) {
150 --        if (port.direction = 'inout')
151 --          thisModule.createTerminal(port,vha_entity);
152 --        if (glob_module.behavior.oclIsTypeOf(GLOB!SDF))
153 --          thisModule.createPort(port,vha_entity);
154 --      }
155 --
156 --    }
157 --
158 --  } -- end "do"
159 --} -- end "rule"
160
161 rule createPort(port : GLOB!Port, vha_entity : VHDL!Entity) {
162   to
163     vha_port : VHDL!QuantityPort (
164       name <- port.name,
165       direction <- port.direction
166     )
167   do {
168     vha_entity.ports <- vha_port;
169   }
170 }
171
172 --rule CompositeBlocks {
173 --  from
174 --    glob_module : GLOB!Module(glob_module.behavior.oclIsKindOf(GLOB!
      Composite))
175 --  to
176 --    vha_entity : VHDL!Entity (name <- glob_module.name), --.replaceAll(' ',
       '_')
177 --    vha_archi  : VHDL!Architecture (name <- 'behavior')
178 --  do {
179 --    -- Save this model in a sequence
180 --    --thisModule.allEntities <- Sequence{thisModule.allEntities, vha_entity
      }->flatten();
181 --
182 --    -- Save the proper references.
183 --    --vha_archi.ofEntity       <- vha_entity;
184 --    --vha_entity.architectures <- vha_archi ;
185 --
186 --    for (instance in glob_module.behavior.instances) {
187 --      self.createComponent(vha_archi,instance);
188 --    }
189 --  }
190 --}
191
192 rule createComponent(architecture : VHDL!Architecture ,instance : GLOB!Module
      ) {
193   to
194     vha_comp : VHDL!Component (type <- thisModule.getEntity(instance))
195   do {
196     vha_comp.portMap;
```

```
197     architecture.components <- vha_comp;
198   }
199 }
200
201 rule createEquation(eq : GLOB!Expression, archi : VHDL!Architecture) {
202   to
203     vha_equation : VHDL!Equation
204   do {
205     --vha_equation.lhs <- eq.pieces.first().value.name.split('=').first().
      regexReplaceAll('\\^', '**');
206     --vha_equation.rhs <- eq.pieces.first().value.name.split('=').last( ).
      regexReplaceAll('\\^', '**');
207     vha_equation.lhs <- eq.name.split('=').first();
208     vha_equation.rhs <- eq.name.split('=').last();
209     archi.equations  <- vha_equation; }
210 }
211
212 rule createConditionedEquation(eq : GLOB!PiecewiseFunction, archi : VHDL!
    Architecture) {
213   to
214     vha_equation : VHDL!IfUse( condition <- eq.pieces.first().interval.bottom
      .toString() + '<' +
215                         eq.argument.name + '<' +
216                         eq.pieces.first().interval.top.toString(),
217                   codeLines <- eq.name + '==' + eq.pieces.first().value.name
      ),
218     vha_default  : VHDL!Else(codeLines <- eq.name + '==' + eq.defaultValue)
219   do {
220     -- Target object should look like this :
221     --
222     --   if ug > usatp use
223     --     uout == usatp;
224     --   elsif ug < usatm use
225     --     uout == usatm;
226     --   else
227     --     uout == ug - ROUT*iout;
228     --   end use;
229
230     for(piece in eq.pieces.excluding(eq.pieces.first())) {
231       thisModule.addPiecewiseFunction(
232         vha_equation,
233         piece.interval.bottom.toString() + '<' + eq.argument.name + '<' +
      piece.interval.top.toString(),
234         eq.name + '==' + piece.value.name );
235     }
236     vha_equation.default <- vha_default;
237     archi.logicalObjects  <- vha_equation;
238   }
239 }
240
241 rule addPiecewiseFunction(vha_ifUse : VHDL!IfUse, condition : String,
    equation : String) {
```

```
242   to
243     vha_elseUse : VHDL!ElseUse (condition <- condition.trim(),
244                   codeLines <- equation)
245   do {
246     vha_ifUse.elseUses <-vha_elseUse;
247   }
248 }
249
250 ---------------------------------- FSM ----------------------------------
251 rule StateMachine {
252   from
253     glob_stateMachine : GLOB!StateMachine
254   to
255     -- Create the objects where we'll save the states
256     vhdl_states : VHDL!EnumerationType (
257       name <- 'stateType'),
258     -- Instances of Enumeration
259     vhdl_nst  : VHDL!Signal(
260       name <- 'next_state',
261       type <- 'stateType'),
262     vhdl_cst  : VHDL!Signal(
263       name <- 'current_state',
264       type <- 'stateType'),
265     -- A switch element
266     vhdl_case : VHDL!Case(
267       var <- vhdl_cst),
268     -- Process : Next State logic
269     vhdl_proc_logic : VHDL!Process(
270       name  <- 'next_state_logic',
271       triggers <- vhdl_cst,
272       codePiece <- vhdl_case),
273     -- Process : Update state
274     vhdl_proc_set_next : VHDL!Process(
275       name  <- 'set_next_state',
276       triggers <- vhdl_nst,
277       codeLines <- 'current_state <= next_state')
278   do {
279     -- TODO: Get rid of the "thisModule.aModule(uml_stm.owner)",
280     --   this should be a simple variable "m"
281     -- Should create a trigger for every input of the state machine.
282     for (p in thisModule.getEntity(glob_stateMachine.owner.owner).ports) {
283       if (p.direction.toString() = 'in') {
284         vhdl_proc_logic.triggers <- p;
285         --thisModule.createTrigger(vhdl_proc_logic,p);
286       }
287     }
288
289     vhdl_proc_logic.codePiece <- vhdl_case;
290
291     thisModule.getEntity(glob_stateMachine.owner.owner).architectures.first()
      .logicalObjects <- vhdl_states; --Enumeratioin
```

```
292     thisModule.getEntity(glob_stateMachine.owner.owner).architectures.first()
        .dataObjects <- vhdl_cst;    --Signal
293     thisModule.getEntity(glob_stateMachine.owner.owner).architectures.first()
        .dataObjects <- vhdl_nst;    --Signal
294     thisModule.getEntity(glob_stateMachine.owner.owner).architectures.first()
        .processes   <- vhdl_proc_logic;
295     thisModule.getEntity(glob_stateMachine.owner.owner).architectures.first()
        .processes   <- vhdl_proc_set_next;
296
297     for (s in glob_stateMachine.states) {
298       thisModule.createState(s, vhdl_case, vhdl_states);
299     }
300   }
301 }
302
303 rule createTrigger(process : SC!Process, p : SC!Port) {
304   to
305     vhdl_ev : VHDL!Event (name <- p.name)
306   do {
307     process.triggers <- vhdl_ev;
308   }
309 }
310
311 -- For every State, we have to add a Case to the Swhitch and a literal to the
        enumeration
312 rule createState(glob_state : GLOB!State, case : VHDL!Case, enum : VHDL!
        EnumerationType) {
313   to
314     vhdl_when : VHDL!When (
315       item <- glob_state.name,
316       codeLines <- glob_state.invariant)
317   do {
318
319     for (t in glob_state.transition) {
320       thisModule.createIf(vhdl_when,t);
321     }
322     case.choices <- vhdl_when;
323     enum.labels  <- glob_state.name;
324   }
325 }
326
327 -- Create the if clause corresponding to the transition
328 rule createIf(c : VHDL!When, t : GLOB!Transition) {
329   to
330 --    code_fragment : SC!CodeFragment (line <- 'next_state = ' + t.target.
        name),
331     vhdl_if : VHDL!If (
332       condition <- t.guard.name.regexReplaceAll('=', '=='),
333       codeLines <- 'next_state = ' + t.target.name)
334   do {
335     c.codePiece <- vhdl_if;
336   }
```

```
337 }
```

C.4 ACCELEO : SystemC-AMS code generator

Same as Appendix A.2.

C.5 ACCELEO : VHDL-AMS code generator

Same as Appendix B.2.

Bibliography

[1] Ieee standard for ip-xact, standard structure for packaging, integrating, and reusing ip within tool flows. *IEEE Std 1685-2009* (Feb 2010), 1–374.

[2] Albert, V. Traduction d'un modèle de système hybride basé sur réseau de Petri en VHDL-AMS. *Master de conception en architecture de machines et systèmes informatiques, Université Paul Sabatier, LAAS-CNRS* (2005).

[3] Aljunaid, H., and Kazmierski, T. J. Seams-a systemc environment with analog and mixed-signal extensions. In *Circuits and Systems, 2004. ISCAS'04. Proceedings of the 2004 International Symposium on* (2004), vol. 5, IEEE, pp. V–281.

[4] Arden, W., Brillouët, M., Cogez, P., Graef, M., Huizing, B., and Mahnkopf, R. Morethan-moore white paper. *Version 2* (2010), 14.

[5] Ashenden, P. J. *The designer's guide to VHDL*, vol. 3. Morgan Kaufmann, 2010.

[6] Bamakhrama, M. A., Zhai, J. T., Nikolov, H., and Stefanov, T. A methodology for automated design of hard-real-time embedded streaming systems. In *Proceedings of the Conference on Design, Automation and Test in Europe* (2012), EDA Consortium, pp. 941–946.

[7] Berry, G., and Gonthier, G. The synchronous programming language esterel: design, semantics. Tech. rep., implementation. Technical Report 842, INRIA, 1988.

[8] Bjørnsen, J., Bonnerud, T. E., and Ytterdal, T. Behavioral modeling and simulation of mixed-signal system-on-a-chip using systemc. *Analog Integrated Circuits and Signal Processing 34*, 1 (2003), 25–38.

[9] Bondé, L., Dumoulin, C., and Dekeyser, J.-L. Metamodels and MDA transformations for embedded systems. In *Advances in design and specification languages for SoCs*. Springer, 2005, pp. 89–105.

[10] Boulanger, F., and Hardebolle, C. Simulation of multi-formalism models with ModHel'X. In *Software Testing, Verification, and Validation, 2008 1st International Conference on* (2008), IEEE, pp. 318–327.

[11] Boulanger, F., and Hardebolle, C. Execution of models with heterogeneous semantics. In *Tutorial on critical systems simulation at CSDM'12* (December 2012).

[12] Bouquet, F., Gauthier, J.-M., Hammad, A., and Peureux, F. Transformation of SysML structure diagrams to VHDL-AMS. In *Design, Control and Software Implementation for Distributed MEMS (dMEMS), 2012 Second Workshop on* (2012), IEEE, pp. 74–81.

[13] Brooks, C., Lee, E. A., Liu, X., Neuendorffer, S., Zhao, Y., and Zheng, H. Heterogeneous concurrent modeling and design in java (volume 3: Ptolemy II domains). *EECS Department, University of California, Berkeley, UCB/EECS-2008-37* (2008).

[14] Cafe, D. C., dos Santos, F. V., Hardebolle, C., Jacquet, C., and Boulanger, F. Multi-paradigm semantics for simulating SysML models using SystemC-AMS. In *Specification & Design Languages (FDL), 2013 Forum on* (2013), IEEE, pp. 1–8.

[15] Café, D. C., Hardebolle, C., Jacquet, C., Dos Santos, F. V., and Boulanger, F. Discrete-continuous semantic adaptations for simulating sysml models in vhdl-ams. In *MPM 2014* (2014), vol. 1237, pp. 11–20.

[16] Chollet, F., and Liu, H. A (not so) short introduction to Micro Electro Mechanical Systems. `http://memscyclopedia.org/introMEMS.html`, pages 149-152. Nov 2013.

[17] Christen, E., and Bakalar, K. VHDL-AMS - a hardware description language for analog and mixed-signal applications. *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on 46*, 10 (1999), 1263–1272.

[18] Damm, M., Haase, J., Grimm, C., Herrera, F., and Villar, E. Bridging MoCs in SystemC specifications of heterogeneous systems. *EURASIP Journal on Embedded Systems 2008* (2008), 7.

[19] Di Guglielmo, L., Fummi, F., Pravadelli, G., Stefanni, F., and Vinco, S. Univercm: the universal versatile computational model for heterogeneous system integration. *Computers, IEEE Transactions on 62*, 2 (2013), 225–241.

[20] Eker, J., Janneck, J. W., Lee, E. A., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S., and Xiong, Y. Taming heterogeneity-the ptolemy approach. *Proceedings of the IEEE 91*, 1 (2003), 127–144.

[21] Ferrari, A., Mangeruca, L., Ferrante, O., and Mignogna, A. Desyreml: a SysML profile for heterogeneous embedded systems. *Embedded Real Time Software and Systems, ERTS* (2012).

[22] Friedenthal, S., Moore, A., and Steiner, R. *A practical guide to SysML: the systems modeling language*. Morgan Kaufmann, 2011.

[23] GEMOC. On the globalization of modeling languages, May 2011.

[24] Ghenassia, F., and Clouard, A. Tlm: An overview and brief history. In *Transaction Level Modeling with SystemC*. Springer, 2005, pp. 1–22.

[25] Gilles, K. The semantics of a simple language for parallel programming. In *In Information Processing'74: Proceedings of the IFIP Congress* (1974), vol. 74, pp. 471–475.

[26] Girault, A., Lee, B., and Lee, E. A. Hierarchical finite state machines with multiple concurrency models. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on 18*, 6 (1999), 742–760.

[27] Golomb, S. W. Mathematical models: Uses and limitations. *Reliability, IEEE Transactions on 20*, 3 (1971), 130–131.

[28] Grimm, C., Barnasconi, M., Vachoux, A., and Einwich, K. An introduction to modeling embedded analog/mixed-signal systems using SystemC AMS extensions. In *DAC2008 International Conference* (2008).

[29] Grimm, C., Meise, C., Heupke, W., and Waldschmidt, K. Refinement of mixed-signal systems with systemc. In *Proceedings of the conference on Design, Automation and Test in Europe-Volume 1* (2003), IEEE Computer Society, p. 11170.

[30] Guihal, D and Andrieux, L and Esteve, D and Cazarre, A. VHDL-AMS model creation. In *Mixed Design of Integrated Circuits and System, 2006. MIXDES 2006. Proceedings of the International Conference* (2006), IEEE, pp. 549–554.

[31] Halbwachs, N. *Synchronous programming of reactive systems*. No. 215. Springer Science & Business Media, 1992.

[32] Hardebolle, C., and Boulanger, F. ModHel'X: A component-oriented approach to multi-formalism modeling. In *Models in Software Engineering*. Springer, 2008, pp. 247–258.

[33] Harel, D. Statecharts: A visual formalism for complex systems. *Science of computer programming 8*, 3 (1987), 231–274.

[34] Harel, D., and Rumpe, B. Meaningful modeling: what's the semantics of" semantics"? *Computer 37*, 10 (2004), 64–72.

[35] Herrera, F., and Villar, E. A framework for embedded system specification under different models of computation in systemc. In *Proceedings of the 43rd annual Design Automation Conference* (2006), ACM, pp. 911–914.

[36] Herrera, F., and Villar, E. A framework for embedded system specification under different models of computation in SystemC. In *Proceedings of the 43rd annual Design Automation Conference* (2006), ACM, pp. 911–914.

[37] Herrera, F., and Villar, E. A framework for heterogeneous specification and design of electronic embedded systems in SystemC. *ACM Transactions on Design Automation of Electronic Systems (TODAES) 12*, 3 (2007), 22.

[38] Herrera, F., Villar, E., Grimm, C., Damm, M., and Haase, J. Heterogeneous specification with HetSC and SystemC-AMS: Widening the support of MoCs in SystemC. In *Embedded Systems Specification and Design Languages*. Springer, 2008, pp. 107–121.

[39] IEEE. SystemC language reference manual. *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)* (2012), 1–614.

[40] Jouault, F., and Kurtev, I. Transforming models with ATL. *Satellite Events at the MoDELS 2005 Conference* (2006), 128–138.

[41] Kirke, T. Signal processing using c++. *Source Forge* (1993-2005).

[42] Lee, E. A. Disciplined heterogeneous modeling. In *Model Driven Engineering Languages and Systems*. Springer, 2010, pp. 273–287.

[43] Lee, E. A., Davis, I., Muliadi, L., Neuendorffer, S., Tsay, J., et al. Ptolemy ii, heterogeneous concurrent modeling and design in java. Tech. rep., DTIC Document, 2001.

[44] Lee, E. A., and Messerschmitt, D. G. Synchronous data flow. *Proceedings of the IEEE 75*, 9 (1987), 1235–1245.

[45] Lee, E. A., and Sangiovanni-Vincentelli, A. A framework for comparing models of computation. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on 17*, 12 (1998), 1217–1229.

[46] Lee, E. A., and Sangiovanni-Vincentelli, A. L. Component-based design for the future. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011* (2011), IEEE, pp. 1–5.

[47] Mischkalla, F., He, D., and Mueller, W. Closing the gap between UML-based modeling, simulation and synthesis of combined HW/SW systems. In *Design,*

Automation & Test in Europe Conference & Exhibition (DATE), 2010 (2010), IEEE, pp. 1201–1206.

[48] Mueller, W., Ruf, J., Hoffmann, D., Gerlach, J., Kropf, T., and Rosenstiehl, W. The simulation semantics of SystemC. In *Design, Automation and Test in Europe, 2001. Conference and Exhibition 2001. Proceedings* (2001), IEEE, pp. 64–70.

[49] Musset, J., Juliot, E., Lacrampe, S., Piers, W., Brun, C., Goubet, L., Lussaud, Y., and Allilaire, F. Acceleo user guide, 2006.

[50] Nagel, L. W. Spice2: A computer program to simulate semiconductor circuits. *ERL Memo ERL-M520* (1975).

[51] Nikolov, H., Stefanov, T., and Deprettere, E. Systematic and automated multiprocessor system design, programming, and implementation. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on 27*, 3 (2008), 542–555.

[52] OMG. MOF model to text transformation language (MOFM2T), 1.0. *OMG standards, formal/08-01-16* (2008).

[53] OMG. Systems modeling language (SysML) specification. *OMG standards, formal/2012-06-01* (2012).

[54] Patel, H., and Shukla, S. K. *SystemC kernel extensions for heterogeneous system modeling: a framework for Multi-MoC modeling & simulation*. Kluwer Academic Pub, 2004.

[55] Piel, É., Atitallah, R. B., Marquet, P., Meftali, S., Niar, S., Etien, A., Dekeyser, J.-L., and Boulet, P. Gaspard2: from marte to systemc simulation. In *proc. of the DATE* (2008), vol. 8.

[56] Prevostini, M., and Zamsa, E. SysML profile for SoC design and SystemC transformation. *ALaRI, Faculty of Informatics University of Lugano via G. Buffi 13*, 5 (2007).

[57] Raslan, W., and Sameh, A. System-level modeling and design using SysML and SystemC. In *Integrated Circuits, 2007. ISIC'07. International Symposium on* (2007), IEEE, pp. 504–507.

[58] Riccobene, E., Rosti, A., and Scandurra, P. Improving SoC design flow by means of MDA and UML profiles. In *3rd Workshop in Software Model Engineering (WiSME 2004)* (2004).

[59] Savaton, G., Delatour, J., and Courtel, K. Roll your own hardware description language. In *OOPSLA & GPCE Workshop Best Practices for Model Driven Software Development* (2004).

[60] Schlebusch, H.-J. SystemC based hardware synthesis becomes reality. In *Euromicro Conference, 2000. Proceedings of the 26th* (2000), vol. 1, pp. 434 vol.1–.

[61] Selic, B., and Gérard, S. *Modeling and Analysis of Real-Time and Embedded Systems with UML and MARTE: Developing Cyber-Physical Systems*. Elsevier, 2013.

[62] Sjoholm, S., and Lindh, L. *VHDL for Designers*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[63] Thomas, D. E., and Moorby, P. R. *The Verilog® Hardware Description Language*, vol. 2. Springer Science & Business Media, 2002.

[64] Verries, J. *Approche pour la conception de systèmes aéronautiques innovants en vue d'optimiser l'architecture. Application au système portes passager*. PhD thesis, Université Paul Sabatier-Toulouse III, 2010.

[65] Weilkiens, T. *Systems engineering with SysML/UML: modeling, analysis, design*. Morgan Kaufmann, 2011.

[66] Zeigler, B. P., Praehofer, H., and Kim, T. G. *Theory of modeling and simulation: integrating discrete event and continuous complex dynamic systems*. Academic press, 2000.