Multi-level modeling for verification and synthesis of complex systems in a multi-physics context
“Só o sofrimento constroi”

Eli
Abstract

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 assignment 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

**Abstract** vi

**Acknowledgments** vii

**List of Figures** xiii

**Abbreviations** xv

1 Résumé en Français 1

1.1 Introduction 3

1.1.1 État de l’art 3

1.1.2 Génération de code SystemC à partir de SysML 4

1.1.3 Génération de code VHDL et VHDL-AMS à partir de SysML 4

1.2 Notre approche 5

1.3 Application à la génération de code SystemC-AMS 6

1.3.1 Cas d’étude 7

1.3.2 Résultats 9

1.3.3 Discussion 10

1.3.4 Réflexions sur l’approche 10

1.4 Application à la génération de code VHDL-AMS 10

1.4.1 Le Modèle SysML 12

1.4.2 Résultats de simulation 14

1.5 Généralisation de l’approche 15

1.5.1 Un formalisme intermédiaire 15

1.5.2 Exemple de définition de comportement 16

1.5.3 Adaptation sémantique 17

1.5.4 Une nouvelle approche 17

1.6 Conclusions 19

1.7 Perspectives 20

2 Systems Modeling Overview 23

2.1 Introduction 23

2.1.1 Motivation & Problems in industry 24

2.1.2 Models of Computation 26

2.1.3 Heterogeneous modeling languages & tools 28
## Contents

2.1.4 From Ptolemy and ownward .................................................. 29
2.1.5 Multiple MoCs with SystemC .................................................. 30

2.2 Graphical Modeling Languages .................................................. 33
  2.2.1 SysML ................................................................. 33
  2.2.2 MARTE & Gaspard 2 ....................................................... 35
  2.2.3 From SysML/UML to SystemC ............................................. 36
  2.2.4 From SysML/UML to VHDL and VHDL-AMS ............................... 37

2.3 Textual vs Graphical Modeling Languages ..................................... 37
2.4 Problem definition ............................................................... 39
  2.4.1 Simulation semantics ....................................................... 39
  2.4.2 Interactions among models ............................................... 40
2.5 Conclusions ................................................................. 41
  2.5.1 Our objective .............................................................. 41
2.6 Thesis outline ................................................................. 42

3 Tools & Methods ................................................................. 43
  3.1 Model Transformation ....................................................... 43
    3.1.1 ATL ................................................................. 44
      3.1.1.1 Anatomy of the language ........................................ 44
    3.1.2 ACCELEO ............................................................. 45
      3.1.2.1 A template-based language ..................................... 46
    3.1.3 Purpose of model transformations .................................... 47

4 Contributions ................................................................. 49
  4.1 Problem of mixed interfaces .................................................. 49
  4.2 Simulation : Continuous vs Discrete ....................................... 50
  4.3 Semantic adaptation ....................................................... 51
  4.4 Implementation ............................................................. 53
  4.5 SysML to SystemC-AMS Transformation ..................................... 53
    4.5.1 Introduction ............................................................ 53
    4.5.2 The approach ........................................................... 54
    4.5.3 A multi-paradigm semantics .......................................... 58
      4.5.3.1 The simulation engine .......................................... 58
      4.5.3.2 Continuous Time Semantics ..................................... 59
      4.5.3.3 Finite State Machine Semantics ................................ 60
    4.5.4 Semantic Adaptation .................................................. 60
    4.5.5 Case Study ............................................................. 61
      4.5.4.1 The model .......................................................... 61
    4.5.6 Results ................................................................. 64
    4.5.7 Partial Conclusions .................................................... 65
  4.6 SysML to VHDL-AMS Transformation .......................................... 66
    4.6.1 Introduction ............................................................. 66
    4.6.2 A case study of a MEMS Accelerometer ................................. 67
      4.6.2.1 Description of the system ....................................... 68
      4.6.2.2 Mechanical model .................................................. 69
      4.6.2.3 SysML Model ....................................................... 70
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Notre approche</td>
</tr>
<tr>
<td>1.2</td>
<td>Exemple du véhicule à vitesse contrôlée : modèles SysML annotés</td>
</tr>
<tr>
<td>1.3</td>
<td>Diagramme interne du bloc i_dynamics et du bloc i_control</td>
</tr>
<tr>
<td>1.4</td>
<td>Résultats obtenus par génération automatique de code</td>
</tr>
<tr>
<td>1.5</td>
<td>Modèles électrique et mécanique d’un MEMS</td>
</tr>
<tr>
<td>1.6</td>
<td>Modèle SysML [IBD]</td>
</tr>
<tr>
<td>1.7</td>
<td>Modèle SysML [BDD]</td>
</tr>
<tr>
<td>1.8</td>
<td>Fonction définie par morceaux &amp; langage d’instanciation de l’adaptateur</td>
</tr>
<tr>
<td>1.9</td>
<td>Résultats de la simulation</td>
</tr>
<tr>
<td>1.10</td>
<td>Métamoèle intermédiaire</td>
</tr>
<tr>
<td>1.11</td>
<td>Machine à états finis</td>
</tr>
<tr>
<td>1.12</td>
<td>L’approche complète</td>
</tr>
<tr>
<td>2.1</td>
<td>Different SystemC-AMS data models</td>
</tr>
<tr>
<td>2.2</td>
<td>SysML Diagrams</td>
</tr>
<tr>
<td>2.3</td>
<td>The Gaspard 2 transformation chain</td>
</tr>
<tr>
<td>3.1</td>
<td>Example ATL rule</td>
</tr>
<tr>
<td>3.2</td>
<td>Filtered transformation</td>
</tr>
<tr>
<td>3.3</td>
<td>Use of the do block in ATL</td>
</tr>
<tr>
<td>3.4</td>
<td>Example Acceleo template</td>
</tr>
<tr>
<td>4.1</td>
<td>Transformation chain</td>
</tr>
<tr>
<td>4.2</td>
<td>ATL rule: port from SysML to SystemC</td>
</tr>
<tr>
<td>4.3</td>
<td>SystemC-AMS simplified metamodel</td>
</tr>
<tr>
<td>4.4</td>
<td>Header generation with Acceleo</td>
</tr>
<tr>
<td>4.5</td>
<td>Continuous Time Building Blocks</td>
</tr>
<tr>
<td>4.6</td>
<td>Vehicle composition</td>
</tr>
<tr>
<td>4.7</td>
<td>Vehicle dynamics</td>
</tr>
<tr>
<td>4.8</td>
<td>Vehicle control</td>
</tr>
<tr>
<td>4.9</td>
<td>Vehicle Composed of the dynamics and control</td>
</tr>
<tr>
<td>4.10</td>
<td>Results obtained from an automatic code generation</td>
</tr>
<tr>
<td>4.11</td>
<td>Electrical vs Mechanical Model</td>
</tr>
<tr>
<td>4.12</td>
<td>SysML Model [IBD]</td>
</tr>
<tr>
<td>4.13</td>
<td>SysML Model [BDD]</td>
</tr>
<tr>
<td>4.14</td>
<td>Definition of piecewise equations (SysML vs VHDL-AMS)</td>
</tr>
<tr>
<td>4.15</td>
<td>Adaptor specification in SysML constraints</td>
</tr>
<tr>
<td>4.16</td>
<td>Our approach</td>
</tr>
</tbody>
</table>
List of Figures

4.17 Simulation Results ................................................. 75
4.18 Intermediary Metamodel ........................................... 76
4.19 Hierarchy in the Intermediary Metamodel ....................... 77
4.20 Equations subset .................................................. 78
4.21 Simplified Synchronous Data Flow ............................. 79
4.22 Simplified Finite State Machine ................................. 80
4.23 Semantic Adaptation .............................................. 81
4.24 Semantic Adaptition Specification ............................. 82
4.25 The Full Approach .................................................. 84
4.26 Electric Window Block Definition Diagram .................... 85
4.27 Semantic Adaptition Specification ............................. 85
4.28 Electric Window Internal Block Diagram ...................... 86
4.29 Semantic Adaptition Specification ............................. 86
4.30 Power Window Finite State Machine ........................... 86
4.31 Finite State Machine in the Intermediary Model representation ........ 87
4.32 SDF models in the Intermediary Model representation ........ 87
4.33 Finite State Machine mapping in the SystemC metamodel representation ... 88
## Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>CT</td>
<td>Continuous Time</td>
</tr>
<tr>
<td>DE</td>
<td>Discrete Event</td>
</tr>
<tr>
<td>MoC</td>
<td>Model of Computation</td>
</tr>
<tr>
<td>SDF</td>
<td>Synchronous Data Flow</td>
</tr>
<tr>
<td>FSM</td>
<td>Finite State Machine</td>
</tr>
<tr>
<td>TDF</td>
<td>Timed Data Flow</td>
</tr>
<tr>
<td>LSF</td>
<td>Linear Signal Flow</td>
</tr>
<tr>
<td>ELN</td>
<td>Electrical Linear Networks</td>
</tr>
<tr>
<td>CSP</td>
<td>Communication Sequential Processes</td>
</tr>
<tr>
<td>OMG</td>
<td>Object Management Group</td>
</tr>
<tr>
<td>XMI</td>
<td>XML Interchange</td>
</tr>
<tr>
<td>MDE</td>
<td>Model Driven Engineering</td>
</tr>
<tr>
<td>MDA</td>
<td>Model Driven Architecture</td>
</tr>
<tr>
<td>PIM</td>
<td>Platform Independent Model</td>
</tr>
<tr>
<td>PSM</td>
<td>Platform Specific Model</td>
</tr>
<tr>
<td>ATL</td>
<td>Atlas Transformation Language</td>
</tr>
<tr>
<td>TLM</td>
<td>Transaction Level Modeling</td>
</tr>
<tr>
<td>AMS</td>
<td>Analog Mixed Signal</td>
</tr>
</tbody>
</table>
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 développement 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


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 : Discrete Event (DE), Timed DataFlow (TDF) et Continuous Time (CT).

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 défini 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 à 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 réalisés dans le but de créer des outils de génération automatique de code à partir des descriptions

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 [1] 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].

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 générions automatiquement du code SystemC-AMS. Cette génération de code est basée sur les transformations de modèles et sur nos définitions sémantiques. Nous profitons ainsi
Résumé en Français

de SysML comme outil de modélisation et nous bénéficiions 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.

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...
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 permis 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].

![Image de résultats de simulation](Image)

**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 multidomaines 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
Les dispositifs MEMS (Micro Electro Mechanical Systems) sont un bon exemple d’un système hétérogène qui mêle 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 $C \propto \frac{1}{(g_0 \pm x)}$, où $g_0$ 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 entre la tension de la membrane et son déplacement. Nous présentons plus de détails du développement mathématique dans la section 4.6.2.

$$V_{middle} = V_0 \frac{x}{g_0} \quad (1.1)$$
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 \( V_{out} = \text{gain} \times V_{in} \). Nous considérons aussi la saturation de cet amplificateur dans une équation par morceaux définie 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...
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

Figure 1.7 – Modèle SysML [BDD]

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 définissons 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
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.
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 définit 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 fonctionnent 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.
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...
M2M1. 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 croyions 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.
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 travail fait 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. Uni-

verCM 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 a 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,
Chapter 1. Systems Modeling Overview

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 onward

Ptolemy influenced 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 heteroge-
neous 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 ana-
log 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 development of SystemC-AMS \cite{systemc-ams}, 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 blocks 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 \cite{asc-library}. 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” \cite{bridging-mocs}. 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
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,
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.
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 closely 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 applied 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 using models can vary from documentation and communication facilities to a decreased time to market delivery compared to conventional development techniques.

In Model Driven Engineering, model transformations are a central operation for handling models. They allow deeper analyses 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 transformations. 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`.

```plaintext
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”.

```java
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.

```java
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 systemic 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.

```java
[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 [/file]
12 [/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 its 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. [31]

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 though 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) which 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 which 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 [36] 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
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 (\texttt{sc\_port}) for every SysML flow port (\texttt{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.

\begin{verbatim}
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 }
\end{verbatim}

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 \cite{53} providing a structured definition of the language's 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 \cite{58} and \cite{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 \cite{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.

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 metamodel 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.

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

```plaintext
[template public genHeader(m : ComposedModule)]
[file (m.name.concat('.hpp'), false, 'UTF-8')]
...
SC_MODULE([m.name/])
{
[for (p : Port | m.port)]
sc_[p.direction/]<[p.type/]> [p.name/];
[/for]
...[/file]
[/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 \texttt{SC\_MODULE} macro. We do that with a \texttt{for} loop that iterates over the sequence of ports of the \textit{Module} ‘\texttt{m}’ and writes equivalent SystemC code. Note that ACCELEO will replace only the code inside the brackets except for internal commands such as \texttt{template}, \texttt{file}, or \texttt{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 \cite{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 \cite{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 subtrators, 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](image)

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.
Chapter 3. Contributions

**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 \( \textbf{Dynamics} \) block and one part \( i_{\.control} \) typed by the \( \textbf{Control} \) block.

![Vehicle composition](image)

**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, \( \text{init\_gain} \) is one instance of type \( \text{gain} \) 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.

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.
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.

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.

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.
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 \( \ll \text{analog} \rr \) and \( \ll \text{digital} \rr \). A third stereotype is dedicated to the description of \( \ll \text{adaptor} \rr \) 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.
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 $C \propto \frac{1}{(g_0 \pm x)}$, 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 $V_{\text{top}} = -V_{\text{bottom}} = V_0$, i.e if we apply a symmetric voltage on both top and bottom electrodes, as explained previously.

$$V_{\text{middle}} = V_0 \frac{C_{\text{top}} - C_{\text{bottom}}}{C_{\text{top}} + C_{\text{bottom}}}$$  \hspace{1cm} (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, $\epsilon$ 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.

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

Correspondingly, for the bottom capacitance:

$$C_{\text{bottom}} = \frac{\epsilon S}{g_0 + x} = \frac{\epsilon S}{g_0} \frac{1}{1 + x/g_0} = C_0 \frac{1}{1 + x/g_0}$$ \hspace{1cm} (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:

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

The interesting feature here is the linear relation between the output voltage $V_{\text{middle}}$ 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. $V_{\text{top}} = -V_{\text{bottom}}$. 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:
Several precautions must be taken to accurately extract $V_{\text{middle}}$. 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 exceed the supply voltages of $V_{DD} = +15\text{V}$ and $V_{SS} = -15\text{V}$. The piecewise equation is as follows:

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

### 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
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 ≪analog≫ and ≪digital≫ blocks to SysML as seen in figure 4.13. We have also added one stereotype ≪adaptor≫ 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.

![Figure 4.13 – SysML Model [BDD]](image-url)
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.

```plaintext
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
```

**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.

```plaintext
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...
Chapter 3. Contributions

73

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 µs.

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 $T_3$ 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 \texttt{IS} and is parameterized by the list of parameters listed after \texttt{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 \texttt{comparator} works correctly as the binary stream output follows the sign of the \texttt{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 \texttt{clk} generates events every $2 \mu s$, 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.
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.

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.

![Hierarchy in the Intermediary Metamodel](image)

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.

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
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 Adap- tor 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,
Chapter 3. Contributions

**Figure 4.23 – Semantic Adaptation**

<table>
<thead>
<tr>
<th>Name</th>
<th>Symbol</th>
<th>Numerical Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Picosecond</td>
<td>ps</td>
<td>1 second $\times 10^{-12}$</td>
</tr>
<tr>
<td>Nanosecond</td>
<td>ns</td>
<td>1 second $\times 10^{-9}$</td>
</tr>
<tr>
<td>Microsecond</td>
<td>us</td>
<td>1 second $\times 10^{-6}$</td>
</tr>
<tr>
<td>Millisecond</td>
<td>ms</td>
<td>1 second $\times 10^{-3}$</td>
</tr>
<tr>
<td>Second</td>
<td>s, sec</td>
<td>1 second</td>
</tr>
<tr>
<td>KiloSecond</td>
<td>ks</td>
<td>1 second $\times 10^{3}$</td>
</tr>
<tr>
<td>MegaSecond</td>
<td>Ms, MegaSec</td>
<td>1 second $\times 10^{6}$</td>
</tr>
<tr>
<td>GigaSecond</td>
<td>Gs</td>
<td>1 second $\times 10^{9}$</td>
</tr>
<tr>
<td>PetaSecond</td>
<td>Ps</td>
<td>1 second $\times 10^{12}$</td>
</tr>
</tbody>
</table>

**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 15 ms or 2 ns 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, \ldots, 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 Adaptation 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 semicolons 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 constraint 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.
Chapter 3. Contributions

Figure 4.25 – The Full Approach
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:

\[
\text{winPosDE}(\text{motorCmd}) = \begin{cases} 
-5 & : \text{motorCmd} \geq 1 \\
+5 & : \text{motorCmd} \leq -1 \\
\int (\text{motorCmd} \times 0.2) & : \text{elsewhere}
\end{cases}
\] (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
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 4.28 – Electric Window Internal Block Diagram

\[
\text{PIECEWISE FUNCTION } y \equiv \begin{cases} 
+5, & x > 1 \\
-5, & x < -1 \\
x, & \text{else}
\end{cases}
\]

Figure 4.29 – Semantic Adaptation Specification

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.

![Finite State Machine in the Intermediary Model representation](image)

**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.

![SDF models in the Intermediary Model representation](image)

**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.

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: ≪ adaptor ≫, ≪ digital ≫ and ≪ analog ≫. 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 the 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

```plaintext
module run;
create OUT : SC from IN : UML, IN2 : SYSML;

helper def : allModules : Sequence(SC!ComposedModule) = Sequence{};
helper def : signals : Sequence(SC!Signal) = Sequence{};
helper def : ports : Sequence(SC!Port) = Sequence{};

rule lsfModules {
  from
  sml_block : SYSML!Block(sml_block.base_Class.name.startsWith('sca_lsf'))
  to
  sc_module : SC!lsfModule (name <- sml_block.base_Class.name)
do {
  -- Save this module in a Sequence :
```
thisModule.allModules <- Sequence(thisModule.allModules, sc_module)->flatten();
}
}
rule tdfModules {
    from
    sml_block : SYSML!Block(sml_block.base_Class.name.startsWith('sca_tdf'))
to
    sc_module : SC!tdfModule (name <- sml_block.base_Class.name)
do {
    -- Save this module in a Sequence :
    thisModule.allModules <- Sequence(thisModule.allModules, sc_module)->flatten();
}
}

--- Create all the Composed Modules (those composed of the base modules)
rule ComposedModules {
    from
    (not sml_block.base_Class.isFinalSpecialization)
sml_block : SYSML!Block(not sml_block.base_Class.name.startsWith('sc'))
to
    (not name.startsWith('sc'))
    sc_module : SC!ComposedModule(name <- sml_block.base_Class.name)
do {
    if (not sml_block.base_Class.ownedRule.isEmpty()) {
        if (sml_block.base_Class.ownedRule.first().specification.value.toString().endsWith('LSF'))
            sc_module.formalism <- 'LSF';
    }
    -- Save this module in a Sequence :
    thisModule.allModules <- Sequence(thisModule.allModules, sc_module)->flatten();
}

--- Create all ports for the Composed Modules
rule Ports {
    from
    sml_port : SYSML!FlowPort
to
    sc_port : SC!Port {
        name <- sml_port.base_Port.name,
        type <- sml_port.base_Port.type.name,
        direction <- sml_port.direction
    }
do {
    -- First, set the specified formalism, if not specified, default value is 'DE'
--- TODO
--- sc_port.formalism <=

for (m in thisModule.allModules) {
  sml_port.base_Port.owner.name.debug();
  if (not thisModule.allModules->
      select(e | e.name = sml_port.base_Port.owner.name).first().
      oclIsUndefined() )
  {
    thisModule.allModules->
    select(e | e.name = sml_port.base_Port.owner.name).first().
    port <= sc_port;
  }
}

for (q in sml_port.base_Port.qualifier) {
  if (q.name.startsWith('isAdaptor'))
  {
    thisModule.createAdaptor(thisModule.aModule(sml_port.base_Port.owner)
        , q, sc_port);
  }
}

rule createAdaptor(m : SC!ComposedModule, q : UML!Property, bindedTo : SC!
    Port) {
  to
  sc_adaptor : SC!Adaptor {
    bindsTo <= bindedTo,
    direction <= bindedTo.direction,
    name <= q.name.replaceAll('('','@').split('@').last().replaceAll(')','
      @').split('@').first().concat('_' + bindedTo.name),
    from <= q.name.replaceAll('(','@').split('@').last().replaceAll(')','
      @').split('@').first().replaceAll('2').first().toUpper(),
    to <= q.name.replaceAll('(','@').split('@').last().replaceAll(')','
      @').split('@').first().split('2').last().toUpper()
  }
  do {
    m.subModules <= sc_adaptor;
    m.adaptors <= sc_adaptor;
    bindedTo.hasAdaptor <= sc_adaptor;
  }
}

rule Adaptors {
  from
  uml_adapt : SYSML!ConstraintProperty(uml_adapt.base_Property.name.
    startsWith('isAdaptor'))
  to
  sc_adaptor : SC!Adaptor {
    name <= uml_adapt.base_Property.owner.name,
    direction <= uml_adapt.base_Property.owner.direction
  }
do {
  -- If Ports direction is ‘in’, use the first formalism, if ‘out’ use the
  -- in the adaptor
  if (thisModule.aModule(uml_adapt.base_Property.owner.owner).port->
    select(e | e.name = uml_adapt.base_Property.owner.name).first().
    direction.toString() = ‘in’) {
    thisModule.aModule(uml_adapt.base_Property.owner.owner).port->
    select(e | e.name = uml_adapt.base_Property.owner.name).first().
    formalism
    <- uml_adapt.base_Property.name.replaceAll(‘(‘, ‘@’).split(‘@’).
    last().replaceAll(‘)’).split(‘@’).first().split(‘2’).first().toUpper()
  } else {
    thisModule.aModule(uml_adapt.base_Property.owner.owner).port->
    select(e | e.name = uml_adapt.base_Property.owner.name).first().
    formalism
    <- uml_adapt.base_Property.name.replaceAll(‘(‘, ‘@’).split(‘@’).
    last().replaceAll(‘)’).split(‘@’).first().split(‘2’).last().toUpper()
  }
}

--- Every Composed module can have parts. If any, create subModules

rule Parts {
  from
  sml_part : UML!Property(
    sml_part.isComposite and
    sml_part.type.getAppliedStereotypes()->
    collect(e | e.name).includes(‘Block’) )
  to
  sc_module : SC!Module (name <- sml_part.name)
  do {
    sc_module.instanceOf <- thisModule.allModules->
    select(e | e.name = sml_part.type.name).first();
    thisModule.allModules <- Sequence(thisModule.allModules, sc_module)->
    flatten();
    for (p in thisModule.aModule(sml_part.type).port) {
      thisModule.copyPort(sc_module, p);
    }
    thisModule.allModules->
    select(e | e.name = sml_part.owner.name).first().
    subModules <- sc_module;
rule copyPort(m : SC!Component, p : SC!Port) {
  to
  sc_port : SC!Port {
    name <- p.name,
    type <- p.type,
    bindsTo <- p.bindsTo,
    direction <- p.direction,
    formalism <- p.formalism
  }
  do {
    m.port <- sc_port;
  }
}

rule Signals {
  from
  uml_connector : UML!Connector
to
  sc_signal : SC!Signal {
    name <- uml_connector.name
    type <- uml_connector.end.last().role.type.name
  }
  do {
    -- For every end of the connector (there should be only two of them)
    -- bind the signal to the corresponding port.
    -- If the property partWithPort is Undefined, that
    -- means that this port belongs to the owner of the connector
    if (thisModule.aModule(uml_connector.owner).formalism.toString().toLower () = 'de') {
      sc_signal.type <- uml_connector.end.last().role.type.name;
    } else if (thisModule.aModule(uml_connector.owner).formalism.toString().toLower () = 'lsf') {
      sc_signal.type <- 'lsf_signal';
    }
    for (end in uml_connector.end) {
      if (end.partWithPort.oclIsUndefined()) {
        sc_signal.bindsTo <- thisModule.aModule(uml_connector.owner).
        port->select(e | e.name = end.role.name);
        thisModule.aModule(uml_connector.owner).
        port->select(e | e.name = end.role.name).first().bindsTo <-
        sc_signal;
      } else {
        sc_signal.bindsTo <- thisModule.aModule(end.partWithPort).
        port->select(e | e.name = end.role.name);
        thisModule.aModule(end.partWithPort.type).debug().
        port->select(e | e.name = end.role.name).debug().first().bindsTo
        <- sc_signal;
      }
    }
    -- Register signal in the appropriate module.
  }
}
thisModule.aModule(uml_connector.owner).variable <- sc_signal;

--thisModule.cropDuplicatedSignals();
-- TODO: Find duplicates and merge them
for (sig_a in thisModule.aModule(uml_connector.owner).variable.debug('first'))
  for (sig_b in thisModule.aModule(uml_connector.owner).variable.
    excluding(sig_a).debug())
    if (sig_a.name = sig_b.name)
      sig_b.bindsTo <- sig_a.bindsTo->excluding(sig_b.bindsTo).debug('inside');

rule cropDuplicatedSignals() {
  to
  do {
    for (m in thisModule.allModules) {
      for (v in m.variable) {
        if (v.oclIsTypeOf(SC!Signal))
          thisModule.signals <- Sequence{thisModule.signals, v}->flatten();
      }
    }
    for (sig in thisModule.signals) {
      if (thisModule.signals.debug()->select(e | e.name=sig.name).debug()->size() > 1) {
        for (s in thisModule.signals.debug()->select(e | e.name=sig.name).
          excluding(thisModule.signals.debug())->select(e | e.name=sig.
            name).first() )
          {
            thisModule.signals->select(e | e.name=sig.name).first().bindsTo
            <- s.bindsTo;
            -- s.destroy();
          }
      }
      thisModule.signals <- Sequence{};
    }
  }
}

rule StateMachine {
  from
    uml_stm : UML!StateMachine
  using {

Appendix A. Code: SysML to SystemC-AMS

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!Switch(  
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 }
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
equation
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

```plaintext
[template public run(m : ComposedModule)]
[comment @main /]
"[if (m.name.startsWith('sc'))]"
"do nothing.."
"[else]"
[genHeader(m) /]
[genBody(m) /]
[genMain(m) /]
[/if] /
[/template]

[template public genHeader(m : ComposedModule)]
[comment Header /]
[file (m.name.concat('.hpp'), false, 'UTF-8')] #ifndef [m.name.toUpper()/_HPP
#define [m.name.toUpper()/_HPP
#include "systemc.h"
#include "systemc-ams.h"
[for (sm : Component | m.subModules)]
[includeSubModules(sm)] [/for]
SC_MODULE([m.name /])
/[if (m.port->size() <> 0)]
"// Ports"
/[if] /
[for (p : Port | m.port)]
sc_[p.direction/][p.type/][p.name/];
[/for]
[comment /]
/[if (m.subModules->selectByType(Adaptor)->size() <> 0)]
"// Adaptors"
[declareAdaptors(m) /]
[/if] /
/[if (m.variable->size() <> 0)]
"// Variables"
[/if] /
[for (v : Variable | m.variable)]
[declareVar(v) /]
[/for]
[comment /]
```
if (m.process->size() <> 0)

// Process declaration

[/if]

for (p : Process | m.process) {
  p.returnType/ [p.name/]();
[/for]

[comment]

if (m.subModules->size() <> 0)

// SubModules

[/if]

for (sm : Module | m.subModules->selectByType(Module))
  declareSubModule(sm)/
[/for]

SC_CTOR([m.name/])

for (sm : Component | m.subModules)
  before (': ') separator(',' )[initModules(sm)]//for]

if (m.variable->size() <> 0)

// Variables Initialization

[/if]

for (v : Variable | m.variable)
  if (v.type <> 'lsf_signal')
    initVar(v)/
[/if]

[comment]

if (m.subModules->selectByType(Adaptor)->size() <> 0)

// Input Adaptors

[/if]

for (a : Adaptor | m.subModules->selectByType(Adaptor))
  if (a.direction.toString() = 'in')
    instantiateAdaptor(a)/
[/for]

[comment]

if (m.subModules->size() <> 0)

// SubModules Instantiation

[/if]

for (sm : Module | m.subModules->selectByType(Module))
  instantiateModule(sm)/
[/for]

[comment]

if (m.subModules->selectByType(Adaptor)->size() <> 0)

// Output Adaptors

[/if]

for (a : Adaptor | m.subModules->selectByType(Adaptor))
  if (a.direction.toString() = 'out')
Appendix A. Code: SysML to SystemC-AMS

103  [instantiateAdaptor(a)/]
104 [/if][/for]
105 [/comment]
106 [/if (m.process->size() <> 0)]
107 // Process Registration
108 [/if]
109 [/for (p : Process | m.process)]
110 [/if (p.oclIsTypeOf(Method))]
111 SC_METHOD([p.name/]);
112 [/elif (p.oclIsTypeOf(Thread))]
113 SC_THREAD([p.name/]);
114 [/if]
115 [/for (t : Event | p.triggers)
116 before (' sensitive << ')
117 separator (' << ')
118 after (';')][t.name/][/for]
119 [/for]
120 }
121 [/template]
122 [/file]
123 [/template]
124 [template public instantiateAdaptor(a : Adaptor)]
125 [a.name/].[/if]
126 [/for (a : Adaptor | m.subModules->selectByType(Adaptor))]
127 [template public declareAdaptors(m : ComposedModule)]
128 [for (a : Adaptor | m.subModules->selectByType(Adaptor))]
sca_if
(a.direction.toString() = 'in') [a.to.toString().toLowerCase();
  sca_if
  (a.direction.toString() = 'out') [a.from.toString().toLowerCase();
  sca_source [else]
  (a.direction.toString() = 'out') [a.to.toString().toLowerCase();
  sca_sink [/if][a.name;/];
[/for]
[/template]
[template public includeSubModules(sm : Component)]
[template public includeSubModules(sm : Module)]
[if (sm.instanceOf.name.startsWith('sc_') or
  sm.instanceOf.name.startsWith('sca_'))][else]
#include "[sm.instanceOf.name/].hpp"
[/if]
[/template]
[template public instantiateModule(sm : Component)]
[template public instantiateModule(sm : Module)]
[for (port : Port | sm.port)]
[for (signal : Signal | port.bindsTo)]
[sm.name/][port.name/][signal.name/];
[/for]
[/for]
[/template]
[template public initModules(m : Component)]
[template public initModules(sm : Module)]
[sm.name/]["[sm.name/]"]
[/template]
[template public initModules(sm : lsfModule)]
[sm.name/]["[sm.name/],[sm.rate/]"]
[/template]
[template public initModules(sm : tdfModule)]
[sm.name/]["[sm.name/],[sm.rate/]"]
[/template]
[template public initModules(sm : Adaptor)]
[sm.name/]["[sm.name/]"]
[/template]
[template public declareSubModule(sm : Component)]
[template public declareSubModule(sm : Module)]
[sm.instanceOf.name] [sm.name/];
[/template]
[template public genBody(m : ComposedModule)]
[file (m.name.concat('.cpp'), false, 'UTF-8')]  
#ifndef [m.name.toUpper/]_CPP
#define [m.name.toUpper/]_CPP
#endif
```cpp
#include "[m.name/.hpp"

[for (p : Process | m.process]
[p.returnType/] [m.name/]:[p.name/]()
{ }
[for (s : Switch | p.switch)]
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')]}
#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/];
```
/// Trace
sc_trace_file *tf = sc_create_vcd_trace_file("wave");
[for (p : Port | m.port) ?(p.direction<>'out')]
    sc_trace(tf, [p.name/], [p.name/]);
[/for]

// Start the main simulation thread
sc_core::sc_start(100, SC_NS);

// Post processing — parsing the log files etc
sc_close_vcd_trace_file(tf);
return 0;
}
[/comment]

#include "System.hpp"
int sc_main(int argc, char* argv[] )
{
    // Elaboration
    // Instantiate the top-level SystemC modules here along
    // with the port bindings etc
    // Instantiate Module
    System i_system("system");
    
    // Trace
    sc_trace_file *tf = sc_create_vcd_trace_file("wave");
    sc_trace(tf, i_system.connector1,"v");
    sc_trace(tf, i_system.connector2,"d");
    sc_trace(tf, i_system.connector3,"F");
    
    // Start the main simulation thread
    sc_core::sc_start(5, SC_SEC);
    
    // Post processing — parsing the log files etc
    sc_close_vcd_trace_file(tf);
    return 0;
}
[/file]

[template]
[template public genTest(m : ComposedModule)]
{| ports_in : OrderedSet(Port) = m.port->select(direction<>'out');
ModuleName : String = m.name.concat('_tb');
|}[/template]

[file (ModuleName.concat('.cpp'), false, 'UTF-8')]
#ifndef [m.name.toUpper()/]CPP
#define [ModuleName.toUpper()/]_CPP
#endif
#include "systemc.h"
```cpp
#include "math.h"
SC_MODULE([ModuleName/])
{
    // Ports
    [for (p : Port | ports_in)]
    sc_out<[p.type/>] [p.name/];
[/for]
    SC_CTOR([ModuleName/])
    {
        SC_THREAD(run);
    }
    void run() {
        while(true) {
            [for (p : Port | ports_in)]
            [p.name/].write(rand() % 10);
[/for]
            wait(5,SC_NS);
        }
    }
    #endif
[/file]
[/template]
[template public declareVar(v : Variable)]
[v.type/] [v.name/];
[/template]
[template public declareVar(v : Signal)]
;if (v.type = 'lsf_signal')
sca_lsf::sca_signal [v.name/];[else]
sc_signal<[v.type/>] [v.name/];[/if]
[/template]
[template public declareVar(v : Enumeration)]
enum [v.name/] {
    [for (item : NamedElement| v.literals) separator (',')] [item.name/];[/for] }
[/template]
[template public genSwitch(s : Swich)]
switch ([s.var.name/]) {
    [for (c : Case| s.case)]
    case [c.item/]:
        [for (ic : IfClause | c.ifClause)]
        if ([ic.condition/]) {
            [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 [/for]
365 default :
366     break;
367 [/template]
368 [/template]
369 template public initVar{v : Enumeration/}
Appendix B

Code: SysML to VHDL-AMS

B.1 ATL model to model transformation

```plaintext
module m2m;
create OUT : VHDL from IN : SYSML, IN2 : UML;

helper def : allModels : Sequence(VHDL!Model) = Sequence{};

rule block {
  from
    sml_block : SYSML!Block(\text{not} sml_block.base-Class.isAbstract)
  to
    vha_model : VHDL!Model,
    vha_entity : VHDL!Entity (name \text{<~} sml_block.base-Class.name), \text{---}
    vha_archi : VHDL!Architecture (name \text{<~} 'behavior')
  do {
    \text{--- Save this model in a sequence}
  }
```

thisModule.allModels ← Sequence{thisModule.allModels, vha_model}→
   flatten();

vha_model.entity ← vha_entity;
vha_model.architecture ← vha_archi;

— Get all attributes (including inherited ones)
for (e in sml_block.base_Class.member) {
   — Constraints are equations
   if (e.oclIsTypeOf(SYSML!Constraint)) {
      — If using a piecewise function, create a conditioned equation.
      if (e.specification.value.toString()→startsWith('PIECEWISE FUNCTION'))
         thisModule.createConditionedEquation(e.specification.value, vha_archi);
      — If using the adaptor language, create a 'use' equation
      if (e.specification.value.toString()→startsWith('ADAPTOR'))
         thisModule.createAdaptor(e.specification.value, vha_archi);
      — If not, create a simple equation
      else
         thisModule.createEquation(e.specification.value, vha_archi);
   }
   if (e.oclIsTypeOf(SYSML!Property)) {
      — If it has a default value, create a generic
      if (not e.defaultValue.oclIsUndefined())
         thisModule.createGeneric(e, vha_entity);
      — If not, create a quantity
      else
         thisModule.createQuantity(e, vha_archi);
   }
   — if (e.oclIsTypeOf(SYSML!Port) and e.getAppliedStereotypes().first().oclIsUndefined())
      — Workaround (papyrus bug 23566) : Heritage ports do not appear on internal block diagram.
      — If there is no terminal with the same name, create the terminal.
      if (vha_model.entity.ports→select(terminal | terminal.name = e.name).isEmpty())
         — Actually create the terminal.
         thisModule.createTerminal(e, vha_model);
   }
}

rule createQuantityPort {
   from
   sml_port : SYSML!FlowPort (sml_port.base_Port.type.name = 'Force'
   or
      sml_port.base_Port.type.name = 'Displacement' or
      sml_port.base_Port.type.name = 'Velocity' or
Appendix B. Code: SysML to VHDL-AMS

```plaintext
sml_port.base_Port.type.name = 'Acceleration'

to

vha_port : VHDL!QuantityPort ( name <- sml_port.base_Port.name,
    type <- sml_port.base_Port.type.name,
    direction <- sml_port.direction.name)
do {
    thisModule.aModel(sml_port.base_Port.owner).entity.ports <- vha_port;
    thisModule.addLibraryOnce(thisModule.aModel(sml_port.base_Port.owner),
        'ieee','mechanical_systems'); -- Simpler
    thisModule.addLibraryOnce(thisModule.aModel(sml_port.base_Port.owner),
        'disciplines','kinematic_system'); -- Hamster
}

rule createTerminal (sml_port : SYSML!Port, model : VHDL!Model ) {
    to

    vha_port : VHDL!Terminal ( name <- sml_port.name,
        type <- sml_port.type.name)
do {
    model.entity.ports <- vha_port;
    if (sml_port.type.name = 'Electrical') {
        thisModule.addLibraryOnce(model, 'ieee','electrical_systems'); -- Simpler
        thisModule.addLibraryOnce(model, 'disciplines','electromagnetic_system'); -- Hamster
    }
}

rule createSubComponent {
    from

    sml_composite : UML!Association
    -- Test if both ends of the connector are << blocks >>
    if (sml_composite.memberEnd.first().type.getAppliedStereotypes() >
        collect(e | e.name).includes('Block')
    and
        sml_composite.memberEnd.last().type.getAppliedStereotypes() >
        collect(e | e.name).includes('Block'))
do {
    -- Add the composite one to the "submodules" property of the other
    if (sml_composite.memberEnd.first().isComposite) {
        thisModule.aModel(sml_composite.memberEnd.last().type).architecture.
        first().submodules <-
            thisModule.aModel(sml_composite.memberEnd.first().type);
        for (lib in thisModule.aModel(sml_composite.memberEnd.
            first().type).lib
        ) {
            for (package in lib.usePackage) {
                thisModule.addLibraryOnce(thisModule.aModel(sml_composite.memberEnd.
                    last().type), lib.name,package.name);
            }
        }
    }
}
```

```java
122     }
123     }
124     } else if (sml_composite.memberEnd.last().isComposite) {
125         thisModule.aModel(sml_composite.memberEnd.first().type).architecture.
126             first().submodules <-
127             thisModule.aModel(sml_composite.memberEnd.last().type);
128     }

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         }

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().
150                 connectors
151                     ->select(connector | connector.name = uml_connector.name).isEmpty()
152                 )
153             {
154                 -- Create a connector inside the given architecture
155                 thisModule.createConnector(
156                     uml_connector,
157                     thisModule.aModel(uml_connector.owner).architecture.first()
158                 );
159             }
160             else -- If there is already a connector with the same name, then extend
161                 the existing one.
162                 {
163                     for (uml_port in uml_connector.end) {
164                         thisModule.aModel(uml_connector.owner).entity;
165                         if (thisModule.aModel(uml_connector.owner).architecture.first().
166                             connectors -- from existing connectors
167                             ->select(connector | connector.name = uml_connector.name).first()
168                             -- select the current one (with the same name as the uml_connector)
169                             .binds->select( port | port.name = uml_port.role.name).isEmpty() )
170                             -- and verify that there is no port with the same name as the
171                             uml_connector ends
172                 }
```
rule createConnector(uml_connector : UML!Connector, vha_arch : VHDL!Architecture) {
    to vha_connector : VHDL!Connector do {
        -- Set the name of the connector
        vha_connector.name <- uml_connector.name;
        vha_connector.arch <- vha_arch;
        -- Bind the two ends to corresponding ports
        for (uml_port in uml_connector.end) {
            vha_connector.binds <- thisModule.aModel(uml_port.role.owner).entity.ports
                -> select(vha_port | vha_port.name = uml_port.role.name);
        }
        -- And set the type accordingly
        vha_connector.type <- uml_port.role.type.name;
        if (uml_port.role.type.name = 'Electrical')
            vha_connector.nature <- 'Terminal';
        else if (uml_port.role.type.name = 'Signal')
            vha_connector.nature <- 'Signal';
        else -- Voltage, Current, Force, etc..
            vha_connector.nature <- 'Quantity';
    }
    -- Add the connector to the corresponding model’s body (architecture)
    vha_arch.connectors <- vha_connector;
}
}

rule createEquation(eq : String, archi : VHDL!Architecture) {
    to vha_equation : VHDL!Equation do {
        vha_equation.lhs <- eq.split('=').first().regexReplaceAll('\\^', '∗∗');
        vha_equation.rhs <- eq.split('=').last().regexReplaceAll('\\^', '∗∗');
    }
}

rule createConditionedEquation(eq : String, archi : VHDL!Architecture) {

to

vha_equation : VHDL!Equation
do {
  — Piecewise Function
  —  (interval a) : (equation a),
  —  (interval b) : (equation b),
  —  ... : ...
  —  elsewhere : (equation n),
  —
  vha_equation.lhs <- eq.split('elsewhere').last().split(':').last().split('=').first().trim().regexReplaceAll('\^', '\\^');
  vha_equation.rhs <- eq.split('elsewhere').last().split(':').last().split('=').last().trim();
  for(piecewise in eq.split('PIECEWISE FUNCTION').last().split('')).{
    if (not piecewise.trim().startsWith('elsewhere'))
      thisModule.addPiecewiseFunction(vha_equation,piecewise.split(':').first(),piecewise.split(':').last());
  }
  archi.equations <- vha_equation;
}

rule createAdaptor(eq : String, archi : VHDL!Architecture) {
  vha_adaptor : VHDL!Adaptor(name <- 'adaptor')
do {
  — ADAPTOR
  —  FROM moc1 TO moc2
  —  IS adaptorType
  —  PARAMS
  —  p1 : value1,
  —  . : .
  —  pN : valueN
  —
  vha_adaptor.MoC_to <- eq.split('FROM').last().split('IS').first().split('TO').last().trim();
  vha_adaptor.MoC_from <- eq.split('FROM').last().split('IS').first().split('TO').first().trim();
  vha_adaptor.type <- eq.split('IS').last().split('PARAMS').first().trim();
  for(param in eq.split('PARAMS').last().split(','))
    thisModule.addParam(vha_adaptor,param.split(':').first().trim(),param.split(':').last().trim());
  archi.adaptors <- vha_adaptor;
}

rule addParam(adaptor : VHDL!Adaptor, name : String, value : String) {
  vha_param : VHDL!Parameter
do {
  }
rule addPiecewiseFunction(vha_equation : VHDL!Equation, condition : String, equation : String) {
  to vha_cond_eq : VHDL!ConditionedEquations (condition <- condition.trim(),
     lhs <- equation.split('=').first().trim(),
     rhs <- equation.split('=').last().trim()) do {vha_equation.piecewise <- vha_cond_eq;}
}

rule createQuantity(quantity : SYSML!Property, archi : VHDL!Architecture) {
  do {
    if (quantity.name.split('from').size() > 1) {
      -- If type is Voltage
      if (quantity.type.name = 'Voltage') {
        self.addQuantityBranch('across',
          quantity.name.split('from').first().trim(),
          quantity.name.split(' from ').last().split(' to ').first().trim(),
          quantity.name.split(' from ').last().split(' to ').last().trim(),
          archi);
      }  
      -- If type is Current
      if (quantity.type.name = 'Current') {
        self.addQuantityBranch('through',
          quantity.name.split('from').first().trim(),
          quantity.name.split(' from ').last().split('to').first().trim(),
          quantity.name.split(' from ').last().split('to').last().trim(),
          archi);
      }  
      -- If it is not a branch quantity, it is a free quantity
      else if (quantity.type.name = 'Real' or
        quantity.type.name = 'Capacitance' or
        quantity.type.name = 'Force' or
        quantity.type.name = 'Displacement' or
        quantity.type.name = 'Velocity' or
        quantity.type.name = 'Acceleration') {
        self.addQuantity(
          quantity.name,
          quantity.type.name,
          archi);
      } else if (quantity.type.name = 'Electrical') {
        self.addNode(
          quantity.name,
          quantity.type.name,
          archi);
      }
    } else if (quantity.type.name = 'Electrical') {  
      self.addNode(
        quantity.name,
        quantity.type.name,
        archi);
    }
  }
}
rule createGeneric(gen : SYSML!Property, entity : VHDL!Entity) { to
  vha_generic : VHDL!Generic (name <- gen.name,
    type <- gen.type.name)
do {
  if (gen.defaultValue.name.toString().endsWith('p'))
    vha_generic.initialValue <− gen.defaultValue.name.toString().substring(1,gen.defaultValue.name.toString().size()-1).concat(‘.0E-12’);
  else if (gen.defaultValue.name.toString().endsWith('n'))
    vha_generic.initialValue <− gen.defaultValue.name.toString().substring(1,gen.defaultValue.name.toString().size()-1).concat(‘.0E-9’);
  else if (gen.defaultValue.name.toString().endsWith('u'))
    vha_generic.initialValue <− gen.defaultValue.name.toString().substring(1,gen.defaultValue.name.toString().size()-1).concat(‘.0E-6’);
  else if (gen.defaultValue.name.toString().endsWith('m'))
    vha_generic.initialValue <− gen.defaultValue.name.toString().substring(1,gen.defaultValue.name.toString().size()-1).concat(‘.0E-3’);
  else if (gen.defaultValue.name.toString().endsWith('k'))
    vha_generic.initialValue <− gen.defaultValue.name.toString().substring(1,gen.defaultValue.name.toString().size()-1).concat(‘.0E03’);
  else if (gen.defaultValue.name.toString().endsWith('M'))
    vha_generic.initialValue <− gen.defaultValue.name.toString().substring(1,gen.defaultValue.name.toString().size()-1).concat(‘.0E06’);
  else if (gen.defaultValue.name.toString().endsWith('G'))
    vha_generic.initialValue <− gen.defaultValue.name.toString().substring(1,gen.defaultValue.name.toString().size()-1).concat(‘.0E09’);
  else if (gen.defaultValue.name.toString().endsWith('T'))
    vha_generic.initialValue <− gen.defaultValue.name.toString().substring(1,gen.defaultValue.name.toString().size()-1).concat(‘.0E12’);
  else
    vha_generic.initialValue <− gen.defaultValue.name.toString().substring(1,gen.defaultValue.name.toString().size()).concat(‘.0’);
  entity.generics <− vha_generic;
}
}

rule addQuantityBranch(branchType : String, name : String, plus_terminal : String, minus_terminal : String, archi : VHDL!Architecture) { to
  vha_quantity : VHDL!QuantityBranch {
    name <− name,
    branchType <− branchType,
    plusTerminal <− plus_terminal }
do {

352     if (not (plus_terminal = minus_terminal))
353         vha_quantity.minusTerminal <= minus_terminal;
354     archi.quantities <= vha_quantity;
355 }
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     do {
363         archi.quantities <= vha_quantity;
364     }
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     do {
372         archi.nodes <= vha_node;
373     }
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
384             ->select(package | package.name = aPackage).isEmpty()) {
385             thisModule.addPackage(model, alibrary, aPackage);
386         }
387     }
388 }
389
390 rule addLibrary (model : VHDL!Model, alibrary : String) {
391     to
392         vha_lib : VHDL!Library (name <= alibrary)
393     do {
394         model.lib <= vha_lib;
395     }
396 }
397
398 rule addPackage (model : VHDL!Model, alibrary : String, aPackage : String) {
399     to
400         vha_pack: VHDL!Package (name <= aPackage)
401     do {
402         model.lib->select(lib | lib.name = alibrary).first().usePackage <=
403             vha_pack;
401 } \\
402 )
B.2 ACCELEO Code Generation

```java
[template public aModel(aModel : Model)]
[comment @main/

file (self.entity.name.replaceAll(' ', '_').concat('.vhd'), false, 'UTF-8')
[comment

This template generates one file for every model found.
It can be separated in three parts:
- Library declaration
- Entity
- Architecture
[/]

[for (lib : Library | self.lib)]
library [lib.name/];
[for (pack : Package | lib.usePackage)]use [lib.name/].[pack.name/].all;
[/for][/for]

[comment if has entity /

entity [self.entity.name.replaceAll(' ', '_')/] is
[for (g : Generic | self.entity.generics) before ('\tgeneric (\n')
   separator(';\n') after ('\nt);\n'))
   [g.name/] : [g.type/] := [g.initialValue/][comment
[/for]
[for (p : Port | self.entity.ports) before ('\tport (\n') separator (';\n')
   after ('\nt);\n'))
   declarePort(p)/[comment
[/for]
end entity;
[comment end if has entity /

[comment if has architecture /

architecture [a.name/] of [a.entity.name.replaceAll(' ', '_')/] is
[comment :
   Declare quantities
[/]
[for (node : Terminal | a.nodes) before ('\n')]
terminal [node.name/] : [node.type/];
[/for]
[comment :
   Declare quantities
[/
[for (q : Quantity | a.quantities) before ('\n')]
declareQuantity(q/)
[/for]
[comment :
   Components declaration
```
for (e : Model | a.submodules) separator('
')
component [e.entity.name.replaceAll(' ', '_')]
  for (g : Generic | e.entity.generics) before ('\t'generic\n('n') separator ('\n') after ('\n')]
    [g.name] : [g.type] := [g.initialValue][comment
  ][/for]
  [for (p : Port | e.entity.ports) before ('\t'port\n('n') separator ('\n') after ('\n'))]
    [declarePort(p)][comment
  ][/for]
end component;
[/for]

begin
  Component Instantiation
  [for (instance : Model | a.submodules) before ('\n') separator ('\n')]
    i_[instance.entity.name.replaceAll(' ', '_')] : component [instance.entity
      .name.replaceAll(' ', '_')]
    [for (g : Generic | instance.entity.generics) before ('\t'generic\n      map(\n'\n') separator ('\n') after ('\n'))]
      [g.name] => [g.initialValue][comment
    ][/for]
    [for (p : Port | instance.entity.ports) before ('\t'port\n('n') separator ('\n') after ('\n'))]
      [p.name] => [p.con->select(c : Connector | c.arch.entity.name = aModel.
        entity.name).name][comment
    ][/for]
  [/for]
[comment : Declare adaptors
  ][/for]
[comment :]
[for (adapt : Adaptor | a.adaptors)]
[if (adapt.type = 'Threshold_Detector')]  
  sampling_clock : process
    begin
      clk <= '1';
      wait for [adapt.params->select(name = 'timestep').value];
      clk <= '0';
      wait for [adapt.params->select(name = 'timestep').value];
      end process;
[/for]
Appendix B. Code: SysML to VHDL-AMS

95  output <= output_signal;
96  compare : process (clk) is
97  begin
98    if vin > threshold then
99      output_signal <= '1';
100     else
101        output_signal <= '0';
102    end if;
103  [/if]
104  [/for]
105  [comment :]
106  Declare equations
107  [/]
108  [for (eq : Equation | a.equations) before ('\n')]
109  [declareEquations{eq}/]
110  [/for]
111  [comment : End "if" has architecture /
112  [/file]
113  [/template]
114  [comment ———————————————————————————————————— ]
115  [template public declarePort(p : Port)]
116  Should never happen
117  [/template]
118  [template public declarePort(p : QuantityPort)]
119  quantity [p.name/]: [p.direction/] [p.type/]
120  [/template]
121  [template public declarePort(p : DigitalPort)]
122  [p.name/] : [p.direction/] [p.type/]
123  [/template]
124  [template public declarePort(p : Terminal)]
125  terminal [p.name/]: [p.type/]
126  [/template]
127  [comment ———————————————————————————————————— ]
128  [template public declareQuantity(q : Quantity)]
129  quantity [q.name/]: [q.type/];
130  [/template]
131  [template public declareQuantity(q : QuantityBranch)]
132  quantity [q.name/][q.branchType/][q.plusTerminal/] [if (not q.minusTerminal.
oclIsUndefined())] to [q.minusTerminal/][/if];
133  [/template]
134  [comment ———————————————————————————————————— ]
135  [template public declareEquations(eq : Equation)]
if (eq.piecewise->isEmpty())
[eq.lhs/] == [eq.rhs/];
else
for (cond_eq : ConditionedEquations | eq.piecewise) before ('if') separator ('else if')]
[cond_eq.condition/] use
[cond_eq.lhs/] == [cond_eq.rhs/];
[/for]
else
[eq.lhs/] == [eq.rhs/];
end use;
[/if]
[/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 module globber;
6 create OUT : GLOB from IN : SysML, IN1 : UML;
7
8 — allModules : A sequence of objects containing all created Modules
9 — getModule : This function takes the name of a Module and provides
10 — the Module from the sequence ‘allModules’.
11 helper def : allModules : Sequence(GLOB!State) = Sequence{};
12 helper def : getModule(name : String) : GLOB!Module =
13 thisModule.allModules->select(e | e.name = name).first();

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();

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 seemed to be the best option
29 — because it provides clear semantics without ambiguity neither the
30 — hassle of creating complicated transformation rules.

32 — Here, we distinguish one rule for each MoC
Creation of Finite State Machines (FSM) behavior:

* When a state machine is detected it triggers the creation of a FSM behavior with state machine inside it.
* This rule will convert a hierarchical state machine into a flat one, as long as hierarchy is limited to two levels.

```
rule AtomicFSMModule {
  from sml_block : SysML!Block (sml_block.base_Class.isLeaf and sml_block.base_Class.getAppliedStereotypes() ->select(s | s.name = 'fsm').size() = 1) to
  glob_module : GLOB!Module (name <- sml_block.base_Class.name),
  glob_interface : GLOB!Interface,
  glob_fsmBehavior : GLOB!FSM do {
  glob_module.interface <- glob_interface;
  glob_module.behavior <- glob_fsmBehavior;
  glob_fsmBehavior.owner <- glob_module;
  // Save this module to the list of all Modules
  thisModule.allModules <- thisModule.allModules.append(glob_module);
}
}
```

```
rule StateMachine {
  from sml_stmachine : UML!StateMachine to
  --glob_fsmBehavior : GLOB!FSM,
  glob_stmachine : GLOB!StateMachine (name <- sml_stmachine.name) do {
  for (region in sml_stmachine.region) {
    -- Get all States /
    for (state in region.subvertex) {
      -- Atomic states
      if (state.region.size()==0)
        thisModule.createState(glob_stmachine,state);
      -- Hierarchical States
      else {
        for (subRegion in state.region) {
          for (subState in subRegion.subvertex) {
            thisModule createState(glob_stmachine,subState);
          }
        }
      }
    }
  }
}
```
rule createState(stMachine : GLOB!StateMachine, uml_states : UML!State) {

to

-- The name is composed of the state and it’s owner.
glob_state : GLOB!State (name <- uml_states.container.owner.name.toString().concat('_').concat(uml_states.name.toString()))
do {

-- If there is an invariant, propagate it to the intermediary model
if (not uml_states.stateInvariant.oclIsUndefined())
glob_state.invariant <- uml_states.stateInvariant.name;

-- The invariant of the parent state must also be propagated
if (uml_states.owner.owner.oclIsKindOf(UML!State))
if (not uml_states.owner.owner.stateInvariant.oclIsUndefined())
glob_state.invariant <- uml_states.owner.owner.stateInvariant.name;

-- Add this state to the state machine ..
stMachine.states <- glob_state;
-- ... and to the list of all States
thisModule.allStates <- thisModule.allStates.append(glob_state);
}
}

rule SingleTransitions {

from

uml_transition : UML!Transition (uml_transition.source.region.size() = 0)
to

glob_guard : GLOB!Event(name <- uml_transition.guard.specification.value)
,

glob_transition : GLOB!Transition {
target <- thisModule.getState(uml_transition.target.container.owner.name.concat('_').concat(uml_transition.target.name))

--
guard <- uml_transition.guard.specification.value
}
do {

glob_transition.guard <- glob_guard;
thisModule.getState(uml_transition.source.container.owner.name.concat('_').concat(uml_transition.source.name)).transition <- glob_transition;
}
}

rule MultiTransitions {

from

Appendix C. Code: SysML to All

```java
uml_transition : UML!Transition (uml_transition.source.region.size() <> 0)
do {
  for (region in uml_transition.source.region) {
    for (state in region.subvertex) {
      thisModule.createSingleTransition(
        thisModule.getState(region.owner.name.concat('_').concat(state.name)),
        thisModule.getState(uml_transition.target.container.owner.name.
          concat('_').concat(uml_transition.target.name)),
        uml_transition.guard.specification.value);
    }
  }
}

rule createSingleTransition(stateFrom : GLOB!State, stateTo : GLOB!State,
  guard : String ) {
  glob_guard : GLOB!Event (name <- guard),
  glob_transition : GLOB!Transition (target <- stateTo)
    --,guard <- guard)
do {
  glob_transition.guard <- glob_guard;
  stateFrom.transition <- glob_transition;
}

-- Creation of Continuous Time (CT) behavior:
-- * A CT module is composed of equations and piecewise functions
-- * Equations continuous functions of time
-- * Piecewise functions are defined in ranges of an argument.

rule AtomicCTModule {
  from
    sml_block : SysML!Block (sml_block.base_Class.isLeaf and
      sml_block.base_Class.getAppliedStereotypes() 
        --select(s | s.name = 'ct').size() = 1)
to
    glob_module : GLOB!Module (name <- sml_block.base_Class.name),
    glob_interface : GLOB!Interface,
    glob_ctBehavior : GLOB!CT
  do {
    -- Link interface and behavior to the module.
    glob_module.interface <- glob_interface;
    glob_module.behavior <- glob_ctBehavior;
  }
```
-- Save this module to the list of all Modules
thisModule.allModules <- thisModule.allModules.append(glob_module);

for (constraint in sml_block.base_Class.ownedRule) {
    if (constraint.specification.value.startsWith('PIECEWISE'))
        thisModule.createPiecewiseFunction(glob_module, constraint);
    else
        thisModule.createSingleEquation(glob_module, constraint);
}

rule createSingleEquation(glob_block : GLOB!Module, uml_constraint : UML!Constraint) {
    to
    glob_var : GLOB!Variable(name <- uml_constraint.specification.value)
    do {
        if (glob_block.behavior.oclIsTypeOf(GLOB!SDF))
            glob_block.behavior.equations <- glob_var;
        if (glob_block.behavior.oclIsTypeOf(GLOB!CT))
            glob_block.behavior.equations <- glob_var;
    }
}

rule createPiecewiseFunction(glob_block : GLOB!Module, uml_constraint : UML!Constraint) {
    to
    argument : GLOB!Variable (name <- uml_constraint.specification.value
        .split('0').last()
        .split('IS').first()
        .trim() ),
    glob_piecewiseFunction : GLOB!PiecewiseFunction (name <- uml_constraint.
        specification.value
        .split('FUNCTION').last()
        .split('0').first()
        .trim() )
    do {
        glob_piecewiseFunction.argument <- argument;
        glob_piecewiseFunction.defaultValue <- uml_constraint.specification.value
            .split('else').last().split(':').last().trim();
        glob_block.behavior.piecewiseFunctions <- glob_piecewiseFunction;

        for (expression in uml_constraint.specification.value.toString().split('IS').last().split(',')) {
            if (not expression->contains('else'))
                thisModule.createPiece(glob_piecewiseFunction,
                    uml_constraint.specification.value
                    .split('0').last()
                    .split('IS').first()
                    .trim(),
                    expression.split(':')[1].first().trim(),
                    expression.split(':').last().trim());
        }
    }
}
rule createPiece(container : GLOB!PiecewiseFunction, var : String, range : String, value : String) {
to
  glob_piece : GLOB!Piece,
  glob_range : GLOB!Range,
  glob_value : GLOB!Variable (name <- value)
do {
  — The case : a > x > b (2× greater than)
  if (range.split('>').size() = 3) {
    glob_range.top <- range.split('>').first();
    glob_range.bottom <- range.split('>').last();
  }
  — The case : a < x < b (2× lesser than)
  if (range.split('>').size() = 3) {
    glob_range.top <- range.split('<').last();
    glob_range.bottom <- range.split('<').first();
  }
  — The case : a < x
  if (range.split('<').size() = 2 and range.endsWith(var)) {
    glob_range.top <- 65565; — (+∞)
    glob_range.bottom <- range.split('<').first().toReal();
  }
  — The case : a > x
  if (range.split('>').size() = 2 and range.endsWith(var)) {
    glob_range.top <- range.split('>').first().toReal();
    glob_range.bottom <- -65565; — (-∞)
  }
  — The case : x < a
  if (range.split('<').size() = 2 and range.startsWith(var)) {
    glob_range.top <- range.split('>').last().toReal();
    glob_range.bottom <- -65565; — (-∞)
  }
}
The case: $x > a$

```
if (range.split('>').size() == 2 and
    range.startsWith(var))
{
    glob_range.top <- 65535; /* (+inf); */
    glob_range.bottom <- range.split('>').last().toReal();
}
```

```
glob_piece.interval <- glob_range;
glob_piece.value <- glob_value;
container.pieces <- glob_piece;
}
```

---

Creation of Synchronous Data Flow (SDF) behavior:

```
rule AtomicSDFModule {
from
sml_block : SysML!Block (sml_block.base_Class.isLeaf and
    sml_block.base_Class.getAppliedStereotypes()
    ->select(s | s.name = 'sdf').size() == 1)

to
glob_module : GLOB!Module (name <- sml_block.base_Class.name),
glob_interface : GLOB!Interface,
glob_sdfBehavior : GLOB!SDF
--glob_sdfProcess : GLOB!Process

do {

    -- First create all the metamodel links.
    --glob_sdfBehavior.process <- glob_sdfProcess;
glob_module.interface <- glob_interface;
glob_module.behavior <- glob_sdfBehavior;

    -- Then, save this module to the list of all Modules
    thisModule.allModules <- thisModule.allModules.append(glob_module);

    -- For the moment, we allow only the use of single equations in SDF modules
    -- Piecewise equations should be carefully studied before implementing
    this function.
```
for (constraint in sml_block.base_Class.ownedRule) {
  if (constraint.specification.value.startsWith('PIECEWISE'))
    thisModule.createPiecewiseFunction(glob_module, constraint);
  else
    thisModule.createSingleEquation(glob_module, constraint);
}

--- Creation of Discrete Event (DE) behavior:
--- * Discrete Event are fairly simple. Modules will react exclusively on events presented at their inputs.

rule AtomicDEModule {
  from sml_block : SysML!Block (sml_block.base_Class.isLeaf and sml_block.base_Class.getAppliedStereotypes()
    ->select(s | s.name = 'de').size() = 1)
  to glob_module : GLOB!Module (name <= sml_block.base_Class.name),
       glob_interface : GLOB!Interface,
       glob_deBehavior : GLOB!DE
  do {
    glob_module.interface <= glob_interface;
    glob_module.behavior <= glob_deBehavior;
    -- Save this module to the list of all Modules
    thisModule.allModules <= thisModule.allModules.append(glob_module);
    -- I have absolutely no idea of what else to treat here..
  }
}

--- Composite Homogeneous Modules
--- * Communication is defined by the innerMoC
--- * Only modules of the same MoC are allowed inside an homogeneous behavior
--- * There is no need to adapt data, control nor time
--- * Internal parts are considered to be instances of atomic modules or other
--- composite modules
Appendix C. Code: SysML to All

367  rule CompositeHomogeneousModules {
368    from
369    sml_block : SysML!Block (not sml_block.base.Class.isLeaf and
370                           sml_block.base.Class.getAppliedStereotypes()
371                           ->select(s | s.name = 'adaptor').size() = 0)
372    to
373    glob_module : GLOB!Module (name <- sml_block.base.Class.name),
374    glob_interface : GLOB!Interface(),
375    glob_behavior : GLOB!Homogeneous
376    do {
377
378      if (sml_block.base.Class.getAppliedStereotypes()->select(s | s.name = 'de').size() = 1)
379        glob_behavior.innerMoC <- 'de';
380      if (sml_block.base.Class.getAppliedStereotypes()->select(s | s.name = 'fsm').size() = 1)
381        glob_behavior.innerMoC <- 'fsm';
382      if (sml_block.base.Class.getAppliedStereotypes()->select(s | s.name = 'sdf').size() = 1)
383        glob_behavior.innerMoC <- 'sdf';
384      if (sml_block.base.Class.getAppliedStereotypes()->select(s | s.name = 'ct').size() = 1)
385        glob_behavior.innerMoC <- 'ct';
386
387      glob_module.interface <- glob_interface;
388      glob_module.behavior <- glob_behavior;
389      glob_behavior.owner <- glob_module;
390
391      -- Save this module to the list of all Modules
392      thisModule.allModules <- thisModule.allModules.append(glob_module);
393    }
394  }
395
396
397  rule CompositeHeterogeneousModules {
398    from
399    sml_block : SysML!Block (not sml_block.base.Class.isLeaf and
sml_block.base.Class.getAppliedStereotypes()
        ->select(s | s.name = 'adaptor').size() == 1)
    to
    glob_module : GLOB!Module (name <- sml_block.base.Class.name),
    glob_interface : GLOB!Interface(),
    glob_adaptor : GLOB!Adaptor
    do {
        if (sml_block.base.Class.ownedRule.size() <> 0) {
            if (sml_block.base.Class.ownedRule->first().specification.value.
                startsWith('ADAPTOR')) {
                glob_adaptor.innerMoC <- sml_block.base.Class.ownedRule->first().
                    specification.value
                    .split('FROM').last().split('TO').first().
                    trim();
                for (param in sml_block.base.Class.ownedRule.
                    first().specification.value.split('PARAMS').last().split(',')) {
                    thisModule.createAdaptorParameter(glob_adaptor, param);
                }
            }
        }
        if (sml_block.base.Class.getAppliedStereotypes()->select(s | s.name = 'de'.
            size() == 1)
            thisModule.createAdaptorParameter(glob_adaptor, 'adaptFrom : de');
        if (sml_block.base.Class.getAppliedStereotypes()->select(s | s.name = 'sdf'.
            size() == 1)
            thisModule.createAdaptorParameter(glob_adaptor, 'adaptFrom : sdf');
        if (sml_block.base.Class.getAppliedStereotypes()->select(s | s.name = 'ct'.
            size() == 1)
            thisModule.createAdaptorParameter(glob_adaptor, 'adaptFrom : ct');
        if (sml_block.base.Class.getAppliedStereotypes()->select(s | s.name = 'fsm'.
            size() == 1)
            thisModule.createAdaptorParameter(glob_adaptor, 'adaptFrom : fsm');
        thisModule.createAdaptorParameter(glob_adaptor, 'adaptTo : ' +
            sml_block.base.Class.getValue(sml_block.base.Class.
            getAppliedStereotypes())
            ->select(s | s.name = 'adaptor').first(), 'adaptTo');
    };
    glob_adaptor.innerMoC <- sml_block.base.Class.getValue(sml_block.
        base.Class.getAppliedStereotypes())
        ->select(s | s.name = 'adaptor').first(), 'adaptTo');
    glob_module.interface <- glob_interface;
    glob_module.behavior <- glob_adaptor;
    glob_adaptor.owner <- glob_module;
    -- Save this module to the list of all Modules
    thisModule.allModules <- thisModule.allModules.append(glob_module);
rule createAdaptorParameter(adaptor : GLOB!Adaptor, param : String) {
  to
  glob_param : GLOB!Parameter (name <- param.split(':').first().trim(),
    value <- param.split(':').last().trim())
  do {
    adaptor.params <- glob_param;
  }
}

--- Interfaces definition
--- * Ports
--- * Connectors
--- * Instances?

rule Ports {
  from
  sml_port : SysML!FlowPort
  to
  glob_port : GLOB!Port ( name <- sml_port.base_Port.name,
    owner <- thisModule.getModule(sml_port.base_Port.owner.
      name),
    direction <- sml_port.direction.toString())
  do {
    thisModule.getModule(sml_port.base_Port.owner.name).interface.ports <-
    glob_port;
  }
}
---rule connector {
--- from
--- sml_connector : UML!Connector
--- to
--- glob_connection : GLOB!Connection(name <- sml_connector.name,
--- owner <- thisModule.getModule(sml_connector.owner.
--- name))
--- do{
--- for (component in sml_connector.end) {
---  glob_connection.binds <-
---    thisModule.getModule(component.role.owner.name).interface.ports
---    =>select(p | p.name = component.role.name).first();
---  }
---  thisModule.getModule(sml_connector.owner.name).behavior.link <-
---    glob_connection;
--- }
rule connectors {
  from
  uml_connector : UML!Connector
do {
  -- If the connector does not yet exists,
  if (thisModule.getModule(uml_connector.owner.name).behavior.link
    ->select(connector | connector.name = uml_connector.name).isEmpty())
  {
    -- Create a connector inside the given architecture
    thisModule.createConnector(uml_connector, thisModule.getModule(
      uml_connector.owner.name).behavior);
  }
  else -- If there is already a connector with the same name, then
    extend the existing one.
    for (uml_port in uml_connector.end) {
      thisModule.getModule(uml_connector.owner.name).interface;
      if (thisModule.getModule(uml_connector.owner.name).behavior.link
        ->select(connector | connector.name = uml_connector.name).first() 
        -- from existing connectors
        .binds->select(port | port.name = uml_port.role.name).isEmpty() 
        -- select the current one (with the same name as the uml_connector)
        .binds->select(port | port.name = uml_port.role.name).isIsEmpty() )
        -- and verify that there is no port with the same name as the
        uml_connector ends
        
        thisModule.getModule(uml_connector.owner.name).behavior.link
        ->select(connector | connector.name = uml_connector.name).first()
        .binds
        <- thisModule.getModule(uml_port.role.owner.name).interface.
        ports
        ->select(vha_port | vha_port.name = uml_port.role.name);
    }
  }
}

rule createConnector(uml_connector : UML!Connector, glob_behavior : GLOB!Behavior) {
to
glob_connector : GLOB!Connection
do {
  -- Set the name of the connector
  glob_connector.name <- uml_connector.name;
  glob_connector.owner <- thisModule.getModule(uml_connector.owner.name);
  -- Bind the two ends to corresponding ports
  for (uml_port in uml_connector.end) {
    glob_connector.binds <- thisModule.getModule(uml_port.role.owner.name).
    interface.ports
    ->select(port | port.name = uml_port.role.name);
  }
}
And set the type accordingly

```plaintext
--glob_connector.type <- uml_port.role.type.name;
--if (uml_port.role.type.name = 'Electrical')
  vha_connector.nature <- 'Terminal';
--else if (uml_port.role.type.name = 'Signal')
  vha_connector.nature <- 'Signal';
--else -- Voltage, Current, Force, etc ...
  vha_connector.nature <- 'Quantity';
```

Add the connector to the corresponding model’s body (architecture)

```plaintext
glob_behavior.link <- glob_connector;
for (p in glob_connector.binds ) {
  p.bindedBy <- glob_connector;
}
```

```plaintext
rule Instances {
  from
  uml_composition : UML!Association
  using {
    owner : GLOB!Module = thisModule.getModule(uml_composition.endType->last().name);
    owned : GLOB!Module = thisModule.getModule(uml_composition.endType->first().name);
  }
  do {
    -- Attach the child module to the parent by the relation "instances"
    thisModule.getModule(uml_composition.endType->last().name).behavior.instances <-
      thisModule.getModule(uml_composition.endType->first().name);
    -- The other way around : Make the child know its parent through the relation "parent"
    thisModule.getModule(uml_composition.endType->first().name).parent <-
      thisModule.getModule(uml_composition.endType->last().name);
    owner.behavior.instances <- owned;
  }
}
```
C.2 ATL : From Intermediary Representation to SystemC-AMS

```plaintext
module glob2systemc;
create OUT : SC from IN : GLOB;

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{};

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;

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
24 — simulator
25 — SystemC-AMS, the Analog and Mixed-Signal extension, adds the notion of
26 — three different models of computation on top of SystemC. These are :
27 — (a) Timed Data Flow (TDF), much similar to a Synchronous Data Flow (SDF)
28 — but tokens are tagged with time values.
29 — (b) Linear Signal Flow LSF, very close to what is done in simulink. This
30 — is continuous time model of computation with a periodic or dynamic timestep
31 — solver.
32 — (c) Electrical Linear Networks, witch is inspired in SPICE netlists. It
33 — uses
34 — the topology of the system (circuit) to deduce the differential equations
35 — from it.
36 — The ELN model of computation uses the same numerical solver as LSF.
37 — Here I’ll detail each one separetly.
38
39 rule abstractTdfModules {
40 from
41    glob_module : GLOB!Module(glob_module.behavior.oclIsTypeOf(GLOB!SDF))
42 do {
43    if (glob_module.behavior.equations.size() <> 0)
44    {
45```

```
if (glob_module.behavior.equations.first().name.split('=').last().trim() = 'x\'integ')
  thisModule.tdfAtomicIntegral(glob_module);
else if (glob_module.behavior.equations.first().name.split('=').last().trim() = 'x\'dot')
  thisModule.tdfAtomicDerivative(glob_module);
else
  thisModule.tdfModules(glob_module);
}
else {
  thisModule.tdfModules(glob_module);
}
}

rule tdfModules(glob_module : GLOB!Module) {
  to
    sc_module : SC!TimedDataFlowModule (name <- glob_module.name),
    proc_attributes : SC!Process(name <- 'set_attributes',returnType <- 'void '),
    proc_initialize : SC!Process(name <- 'initialize', returnType <- 'void ')
  proc_processing : SC!Process(name <- 'processing', returnType <- 'void ')
  do {
    -- Save this module in a Sequence :
    thisModule.allModules <- Sequence{thisModule.allModules, sc_module}<- flatten();

    --glob_module.debug();
    --glob_module.parent.debug();
    -- This complicated code is supposed to propagate the adaptor information , such as the type of adaptor
    -- and timestep to the inner modules inside an adaptor interface.
    if (glob_module.parent.behavior.oclIsTypeOf(GLOB!Adaptor))
      if (glob_module.parent.behavior.params->select(p | p.name = 'adaptFrom').first().value = 'de') {
        for (port in glob_module.parent.interface.ports) {
          if (port.bindedBy.binds->select(p | p <> port).first().owner =
            glob_module)
            thisModule.createPortInnerTDFOuterDE(sc_module, port.bindedBy.
              binds->select(p | p <> port).first());
        }

        sc_module.ports.first().timestepValue <-
          glob_module.parent.behavior.params->select(p | p.name = 'timestep').
            first().value.split(' ').first().toReal();
      }
      sc_module.ports.first().timestepScale <-
        glob_module.parent.behavior.params->select(p | p.name = 'timestep').
          first().value.split(' ').last().toUpperCase().trim();
Appendix C. Code: SysML to All

```python
proc_attributes.codeLines = sc_module.ports.first().name + '.
set_timestep(' +
    sc_module.ports.first().timestepValue.toString() + ',SC' +
    ''');
for (port in glob_module.interface.ports->select(p | p.name <> sc_module.
    ports.first().name)) {
    thisModule.createTDFPorts(sc_module, port, glob_module.behavior);
}
for (equation in glob_module.behavior.equations) {
    proc_processing.codeLines = equation.name;
}
for (equation in glob_module.behavior.piecewiseFunctions) {
    thisModule.createPiecewiseEquation(proc_processing, equation);
}
sc_module.process = proc_attributes;
--sc_module.process = proc_initialize;
sc_module.process = proc_processing;
}
rule createPiecewiseEquation(container : SC!Process, equation : GLOB!PiecewiseFunction) {
    inputVariable : SC!Variable (name = 'inputVar', type = 'double',
        initialValue = equation.argument.name + '.read(0)'),
    outputVariable : SC!Variable (name = 'outputVar', type = 'double'),
    sc_if : SC!If(codeLines = outputVariable.name + ' = ' + equation.pieces.
        first().value.name),
    sc_else : SC!Else(codeLines = outputVariable.name + ' = ' +
        inputVariable.name)
    do {
        container.variables = inputVariable;
        container.variables = outputVariable;
        --container.codeLines = inputVariable.name + '=' + equation.argument.
            name + '.read(0)';
        if (equation.pieces.first().interval.top >= 65565.0)
            sc_if.condition = inputVariable.name + ' > ' + equation.pieces.first()
                .interval.bottom;
        else if (equation.pieces.first().interval.bottom <= -65565.0)
            sc_if.condition = inputVariable.name + ' < ' + equation.pieces.first()
                .interval.top;
        else
            sc_if.condition = inputVariable.name + ' > ' + equation.pieces.first()
                .interval.bottom + '&&'
```
Appendix C. Code: SysML to All

+ inputVariable.name + ' < ' + equation.pieces.first().interval.top;
— Add the If clause to the process
container.codePiece <- sc_if;
— And generate the other cases in the form of 'else if' clauses
for (piece in equation.pieces.excluding(equation.pieces.first())) {
  thisModule.createPiece(sc_if,piece);
}
— And add the final 'else' for the default value.
sc_if.elseClause <- sc_else;
container.codeLines <- equation.name + '=' + outputVariable.name;
}
}
rule createPiece(container : SC!If, piece : GLOB!Piece) {
to
  sc_elseIf : SC!ElseIf (codeLines <- 'outputVar = ' + piece.value.name)
do {
  if (piece.interval.top >= 65565.0)
    sc_elseIf.condition <- 'inputVar > ' + piece.interval.bottom;
  else if (piece.interval.bottom <= -65565.0)
    sc_elseIf.condition <- 'inputVar < ' + piece.interval.top;
  container.elseIfClause <- sc_elseIf;
}
}
— The integrator block in the TDF formalism is actually only an interface
to the continuous-time domain LSF.
rule tdfAtomicIntegral(glob_module : GLOB!Module) {
to
  TDFinterface : SC!Module (name <- glob_module.name),
  TDFportIn : SC!TDF_Port (name <- glob_module.interface.ports->select(p | p.direction.toString() = 'in').first().name,
    direction <- 'in', rate <- 1,type <- 'double',owner <- TDFinterface),
  TDFportOut : SC!TDF_Port (name <- glob_module.interface.ports->select(p | p.direction.toString() = 'out').first().name,
    direction <- 'out',rate <- 1,type <- 'double',owner <- TDFinterface),
— Input adaptor
  LSFAdaptorIn : SC!Source (name <- 'tdf2lsf'),
  LSFAdaptInPortIn : SC!TDF Port (name <- 'inp',owner <- LSFAdaptorIn),
  LSFAdaptInPortOut : SC!LSF Port (name <- 'y',owner <- LSFAdaptorIn),
— Output adaptor
  LSFAdaptorOut : SC!Sink (name <- 'lsf2tdf'),
  LSFAdaptOutPortIn : SC!LSF Port (name <- 'x',owner <- LSFAdaptorOut),
  LSFAdaptOutPortOut: SC!TDF Port (name <- 'outp',owner <- LSFAdaptorOut),
— The heart of it: the integrator
LSFIntegrator : SC!Integ (name <- 'LSF_integrator'),
LSFPortIn : SC!LSF_Prot(name <- 'x',owner <- LSFIntegrator),
LSFPortOut : SC!LSF_Prot(name <- 'y',owner <- LSFIntegrator),
    -- Internal connections
SignalIn : SC!TDFHierarchicalSignal(name <- 'tdf2lsfSignal',type <- 'double'),
SignalOut : SC!TDFHierarchicalSignal(name <- 'lsf2tdfSignal',type <- 'double'),
LSFSignalIn : SC!LSF_Signal(name <- 'internal_x'),
LSFSignalOut : SC!LSF_Signal(name <- 'internal_y')

do {
    -- Save this module in a Sequence :
    thisModule.allModules <- Sequence(thisModule.allModules, TDFinterface) -> flatten();
    --for (port in glob_module.interface.ports) {
    --    thisModule.createTDFPorts(TDFinterface,port,glob_module.behavior);
    --}
    -- TDF interface bindings
    TDFinterface.ports <- TDFportIn;
    TDFinterface.ports <- TDFportOut;
    -- Adaptors
    LSFAdaptorIn.ports <- LSFAdeptInPortIn;
    LSFAdaptorIn.ports <- LSFAdeptInPortOut;
    LSFAdaptorOut.ports <- LSFAdeptOutPortIn;
    LSFAdaptorOut.ports <- LSFAdeptOutPortOut;
    -- LSF Integrator bindings
    LSFIntegrator.ports <- LSFPortIn;
    LSFIntegrator.ports <- LSFPortOut;
    -- Internal bindings : instances
    TDFinterface.instances <- LSFIntegrator;
    TDFinterface.instances <- LSFAdaptorIn ;
    TDFinterface.instances <- LSFAdaptorOut;
    -- Internal bindings : connections
    -- Input port bindings
    -- TODO : Check if there are multiple input ports, and scream if it is the case
    SignalIn .binds <- TDFportIn;
    SignalIn .binds <- LSFAdeptInPortIn;
    -- Main LSF block bindings
    LSFSignalIn.binds <- LSFAdeptInPortOut;
    LSFSignalIn.binds <- LSFPortIn;
    LSFSignalOut.binds <- LSFPortOut;
    LSFSignalOut.binds <- LSFAdeptOutPortIn;
--- Output port bindings
--- TODO: Check if there are multiple output ports, and scream if it is the case
SignalOut.binds ← TDFportOut;
SignalOut.binds ← LSFAdaptOutPortOut;

--- All signals belong to the TDFinterface
TDFinterface.signals ← SignalIn;
TDFinterface.signals ← SignalOut;
TDFinterface.signals ← LSFSignalIn;
TDFinterface.signals ← LSFSignalOut;

--- The integrator block in the TDF formalism is actually only an interface
to the continuous-time domain LSF.
rule tdfAtomicDerivative(glob_module : GLOB!Module) {
  TDFinterface : SC!TimedDataFlowModule (name ← 'tdfInterface' +
glob_module.name),
  LSFDerivator : SC!Dot(name ← glob_module.name),
  LSFPortIn : SC!LSF_Port(name ← 'x'),
  LSFPortOut : SC!LSF_Port(name ← 'y'),
  LSFSignalIn : SC!LSF_Signal(name ← 'internal_x'),
  LSFSignalOut : SC!LSF_Signal(name ← 'internal_y'),
  LSFAdaptorIn : SC!Source(name ← 'tdf2lsf' + glob_module.name + 'Adaptor'),
  LSFAdaptorOut : SC!Sink (name ← 'lsf2tdf' + glob_module.name + 'Adaptor'),
  SignalIn : SC!Signal(name ← 'tdf2lsfSignal'),
  SignalOut : SC!Signal(name ← 'lsf2tdfSignal')
  do {
  --- Save this module in a Sequence:
  thisModule.allModules ← Sequence{thisModule.allModules, TDFinterface}→
  flatten();
  LSFDerivator.ports ← LSFPortIn;
  LSFDerivator.ports ← LSFPortOut;
  TDFinterface.instances ← LSFDerivator;
  TDFinterface.instances ← LSFAdaptorIn;
  TDFinterface.instances ← LSFAdaptorOut;
  --- Internal connections
  SignalIn.binds ← LSFAdaptorIn.ports→select(p | p.name = 'y');
  SignalIn.binds ← LSFDerivator.ports→select(p | p.name = 'x');
  SignalOut.binds ← LSFAdaptorOut.ports→select(p | p.name = 'y');
  SignalOut.binds ← LSFAdaptorOut.ports→select(p | p.name = 'x');
Appendix C. Code : SysML to All

rule abstractRuleLSFatomic {
  from
  glob_module : GLOB!Module(glob_module.behavior.oclIsTypeOf(GLOB!CT))
  do {
    if (glob_module.behavior.equations.size() <> 0) {
      if (glob_module.behavior.equations.first().name.split('=').last().trim() == 'x'integ') thisModule.lsfAtomicIntegrator(glob_module);
      else if (glob_module.behavior.equations.first().name.split('=').last().trim() == 'x'dot') thisModule.lsfAtomicDerivator(glob_module);
      else
        thisModule.lsfCustomModules(glob_module);
    }
    else
      thisModule.lsfCustomModules(glob_module);
  }
}

rule lsfCustomModules(glob_block : GLOB!Module) {
  --from
  -- glob_block : GLOB!Module(glob_block.behavior.oclIsTypeOf(GLOB!CT))
  to
  sc_module : SC!LinearSignalFlowModule (name <- glob_block.name)
  do {
    -- Save this module idn a Sequence :
    thisModule.allModules <- Sequence(thisModule.allModules, sc_module) -> flatten();
    for (port in glob_block.interface.ports) {
      thisModule.createPorts(sc_module, port, glob_block.behavior);
    }
    for (equations in glob_block.behavior.equations) {
      'LSF equations — Yet to do'.debug();
    }
    }
  rule lsfAtomicIntegrator(glob_module : GLOB!Module) {
    
```
sc_lsf_integrator : SC!Integ (name <- glob_module.name)

do {
    -- Save this module in a Sequence:
    thisModule.allModules <- Sequence(thisModule.allModules,
        sc_lsf_integrator) -> flatten();
    for (port in glob_module.interface.ports) {
        thisModule.createPorts(sc_lsf_integrator, port, glob_module.behavior);
    }
}

rule lsfAtomicDerivator(glob_module : GLOB!Module) {
    to
    sc_lsf_derivator : SC!Dot (name <- glob_module.name)
    do {
        -- Save this module in a Sequence:
        thisModule.allModules <- Sequence(thisModule.allModules, sc_lsf_derivator)
        -> flatten();
        for (port in glob_module.interface.ports) {
            thisModule.createPorts(sc_lsf_derivator, port, glob_module.behavior);
        }
    }
}

rule fsmModules {
    from
    glob_block : GLOB!Module(glob_block.behavior.oclIsTypeOf(GLOB!FSM))
    to
    sc_module : SC!Module (name <- glob_block.name)
    do {
        -- Save this module in a Sequence:
        thisModule.allModules <- Sequence(thisModule.allModules, sc_module) -> flatten();
        for (port in glob_block.interface.ports) {
            thisModule.createDEPorts(sc_module, port, glob_block.behavior);
        }
    }
}

rule StateMachine {
    from
    glob_stateMachine : GLOB!StateMachine
    to
    -- Create the objects where we’ll save the states
    sc_enum : SC!Enumeration (name <- 'stateType'),
    -- Instances of Enumeration
    sc_nst : SC!Signal
name <- 'next_state',
type <- 'stateType'),
sc_cst : SC!Signal(
    name <- 'current_state',
type <- 'stateType'),
-- A switch element
sc_swch : SC!Switch(
    var <- sc_cst.name),
sc_ev_cst : SC!Event(name <- 'current_state'),
sc_ev_nst : SC!Event(name <- 'next_state'),
-- Process: Next State logic
sc_proc : SC!Method(
    name <- 'next_state_op_logic',
    triggers <- sc_ev_cst,
    codePiece <- sc_swch),
-- Process: Update state
--code_frag : SC!CodeFragment(codeLines <- 'current_state = next_state;')
sc_proc_set_next : SC!Method(
    name <- 'set_next_state',
    triggers <- sc_ev_nst,
    codeLines <- 'current_state = next_state')
do {
    -- TODO: Get rid of the "thisModule.aModule(uml_stm.owner)",
    -- this should be a simple variable "m"
    -- Should create a trigger for every input of the state machine.
    for (p in thisModule.aModule(glob_stateMachine.owner.owner).ports) {
        if (p.direction.toString() = 'in') {
            thisModule.createTrigger(sc_proc,p);
        }
    }
    thisModule.aModule(glob_stateMachine.owner.owner).codePiece <- sc_enum;
    thisModule.aModule(glob_stateMachine.owner.owner).signals <- sc_cst;
    thisModule.aModule(glob_stateMachine.owner.owner).signals <- sc_nst;
    thisModule.aModule(glob_stateMachine.owner.owner).process <- sc_proc;
    sc_proc_set_next;
    for (s in glob_stateMachine.states) {
        thisModule createState(s, sc_swch, sc_enum);
    }
}
rule createTrigger(process : SC!Process, p : SC!Port) {
    to sc_ev : SC!Event (name <- p.name)
do {
    process.triggers <- sc_ev;
}
}
Appendix C. Code : SysML to All

For every State, we have to add a Case to the Switch and a literal to the enumeration.

```
rule createState(glob_state : GLOB!State, swch : SC!Switch, enum : SC!Enumeration) {
    to
    --sc_code : SC!CodeFragment(line <- glob_state.invariant),
    sc_case : SC!Case {
        item <- glob_state.name,
        -- code <- sc_code),
        codeLines <- glob_state.invariant),
    sc_label : SC!Label (name <- glob_state.name)
    do {
        for (t in glob_state.transition) {
            thisModule.createIf(sc_case,t);
        }
        swch.cases <- sc_case;
        enum.labels <- sc_label;
    }
}
```

Create the if clause corresponding to the transition.

```
rule createIf(c : SC!Case, t : GLOB!Transition) {
    to
    -- code_fragment : SC!CodeFragment (line <- 'next_state = ' + t.target.
    name),
    sc_if : SC!If {
        condition <- t.guard.name.regexReplaceAll('=', '=='),
        -- code <- code_fragment)
        codeLines <- 'next_state = ' + t.target.name
    do {
        c.codePiece <- sc_if;
    }
}
```

--- DE --------------------------------------------

```
rule deModules {
    from
glob_block : GLOB!Module(glob_block.behavior.oclIsTypeOf(GLOB!DE))
to
sc_module : SC!Module (name <- glob_block.name)
do {
    -- Save this module in a Sequence :
    thisModule.allModules <- Sequence{thisModule.allModules, sc_module}->
    flatten();
    for (port in glob_block.interface.ports) {
        thisModule.createDEPorts(sc_module, port, glob_block.behavior);
    }
}
```
Heterogeneous Modules are interface modules that allow different models of computation to communicate. Usually, an heterogeneous module is contained within a MoC and defines a different MoC at its interior.

```
rule HeterogeneousModules { 
  from 
  glob_block : GLOB!Module(glob_block.behavior.oclIsKindOf(GLOB!Heterogeneous)) | or 
  to 
  sc_module : SC!Module(name < glob_block.name) 
  do { 
    -- Save this module in a Sequence: 
    thisModule.allModules <- Sequence{thisModule.allModules, sc_module}->flatten(); 
    if(glob_block.behavior.params->select(p | p.name = 'adaptFrom').first().value = 'de') { 
      for (port in glob_block.interface.ports) { 
        thisModule.createDEPorts(sc_module, port, glob_block.behavior); 
      }
    } 
    if(glob_block.behavior.params->select(p | p.name = 'adaptFrom').first().value = 'sdf') { 
      for (port in glob_block.interface.ports) { 
        thisModule.createTDFPorts(sc_module, port, glob_block.behavior); 
      }
    }
    for (instance in glob_block.behavior.instances) { 
      sc_module.instances <- thisModule.aModule(instance); 
    }
  }
}
```

Depending on the MoC, we should create appropriate ports.

DE Modules use simple DE ports, which differs from TDF and LSF ports.

```
rule HomogeneousModules { 
  from 
  glob_block : GLOB!Module(glob_block.behavior.oclIsKindOf(GLOB!Homogeneous)) | or 
  to 
  sc_module : SC!Module(name < glob_block.name) 
  do { 
    -- Save this module in a Sequence: 
    thisModule.allModules <- Sequence{thisModule.allModules, sc_module}->flatten(); 
    if(glob_block.behavior.innerMoC = 'de') { 
      for (port in glob_block.interface.ports) { 
        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    sc_module.ports <- sc_port;
539  }
540 )
541 
542 rule createTDFPorts(sc_module : SC!Module, port : GLOB!Port, behavior : GLOB!Behavior) {
543  to
544    sc_port : SC!TDF_Port (name <- port.name, type <- 'double', direction <- port.direction, owner <- sc_module)
545  do {
546    sc_module.ports <- sc_port;
547  }
548  }
549 )
550 
551 -- Signals:
552 -- By construction, signals only exist in composite modules
553 -- so we must check the inner model of computation to instantiate
554 -- the proper signal. TDF modules use TDF signals, LSF modules use
555 -- LSF signals, etc..
556 rule DESignals {
from
glob_connection : GLOB!Connection(glob_connection.owner.behavior.innerMoC
    .toString() = 'de')
to
sc_signal : SC!Signal (name <- glob_connection.name,type <- 'double')
do {
  --- signal owner
  thisModule.aModule(glob_connection.owner).signals <- sc_signal;

  --- Binding new connections
  '-'.debug();
  --glob_connection.owner.debug();
  --glob_connection.binds.debug();
  for (port in glob_connection.binds) {
    ---if (port.owner = glob_connection.owner)
    --- ('port owner : ' + port.owner.name).debug();
    --- ('connection owner : ' + glob_connection.owner.name).debug();
    sc_signal.binds <- thisModule.aModule(port.owner).ports->select(p | p.
        name = port.name);
  }
}

rule SDFSignals {
  from
glob_connection : GLOB!Connection(glob_connection.owner.behavior.innerMoC
    .toString() = 'sdf')
do {
  --- Binding new connections
  '-'.debug();
  if (glob_connection.binds->select(p | p.owner = glob_connection.owner).
      size() <> 0)
    thisModule.createHierarchicalSignalTDF(glob_connection);
  else
    thisModule.createPureSignalTDF(glob_connection);
}

rule createPureSignalTDF(glob_connection : GLOB!Connection) {
  to
  sc_signal : SC!TDF_Signal (name <- glob_connection.name,type <- 'double')
do {
  thisModule.aModule(glob_connection.owner).signals <- sc_signal;
  for (port in glob_connection.binds) {
    sc_signal.binds <- thisModule.aModule(port.owner).ports->select(p | p.
        name = port.name);
  }
}
rule createHierarchicalSignalTDF(glob_connection : GLOB!Connection) {
    sc_signal : SC!TDFHierarchicalSignal (name <- glob_connection.name, type <- 'double')
    do {
        thisModule.aModule(glob_connection.owner).signals <- sc_signal;
        for (port in glob_connection.binds) {
            sc_signal.binds <- thisModule.aModule(port.owner).ports->select(p | p.
                name = port.name);
        }
    }
}

rule CTSignals {
    from
    glob_connection : GLOB!Connection(glob_connection.owner.behavior.innerMoC 
        .toString() = 'ct')
    to
    sc_signal : SC!LSF_Signal (name <- glob_connection.name, type <- 'double' 
    )
    do {
        -- signal owner
        thisModule.aModule(glob_connection.owner).signals <- sc_signal;
        -- Binding new connections
        for (port in glob_connection.binds) {
            sc_signal.binds <- thisModule.aModule(port.owner).ports->select(p | p.
                name = port.name);
        }
    }
}

-- Yet to do

-- Port Adaptors

rule createPortInnerTDFOuterDE(sc_module : SC!Module, port : GLOB!Port) {
    sc_port : SC!TDF_DE_Adaptor(name <- port.name, 
        direction <- port.direction, 
        type <- 'double', 
        owner <- sc_module)
        --bindsTo <- sc_port
    do {
        -- sc_port.hasAdaptor <- sc_adaptor;
        -- sc_module.adaptors <- sc_adaptor;
        sc_module.ports <- sc_port;
    }
}
C.3 ATL : From Intermediary Representation to VHDL-AMS

```vhd```

1 /* atlcompiler atl2006 */
2 */
3 /* path VHDL=/Metamodels/vhdl-ams.ecore */
4 /* path GLOB=/Metamodels/glob.ecore */
5 module glob2vhdl;
6
create OUT : VHDL from IN : GLOB;
7
declare all objects for further manipulation
8 helper def : allEntities : Sequence(VHDL!Entity) = Sequence{};
9
usefull functions
10 aModule : This function takes a named object in its input and provide
11 the corresponding module from the Sequence allModules that maches
12 the object name.
13 helper def : getEntity(o : OclAny) : VHDL!Entity =
14 if (o.oclIsUndefined()) then OclUndefined else
15 thisModule.allEntities->select(e | e.name = o.name).first() endif;
16
17 rule Entity {
18 from glob_interface : GLOB!Interface
19 to vhdl_entity : VHDL!Entity (name -> glob_interface.owner.name)
20 do {
21 thisModule.allEntities <- Sequence(thisModule.allEntities, vhdl_entity)->
22 flatten();
23 for (port in glob_interface.ports) {
24 thisModule.createComponent(vhdl_entity, port);
25 }
26 }
27 }
28
29 rule ArchitectureHierarchical {
30 from glob_behavior : GLOB!Composite
31 to vhdl_archi : VHDL!Architecture ()
32 do {
33 vhdl_archi.ofEntity <- thisModule.getEntity(glob_behavior.owner);
34 thisModule.getEntity(glob_behavior.owner).architectures <- vhdl_archi ;
35 for (instance in glob_behavior.instances) {
36 self.createComponent(vhdl_archi, instance);
37 }
38 }
```
rule ArchitectureContinuousTime {
  from 
glob_behavior : GLOB!CT
to 
vhdl_archi : VHDL!Architecture ()
do {
  vhdl_archi.ofEntity <- thisModule.getEntity(glob_behavior.owner);
  thisModule.getEntity(glob_behavior.owner).architectures <- vhdl_archi ;
  for (equation in glob_behavior.equations) {
    --if (equation.pieces->size() > 1)
    -- thisModule.createConditionedEquation(equation,vha_archi);
    --else
    thisModule.createEquation(equation,vhdl_archi);
    -- If using the adaptor language, create a 'use' equation
    if (e.specification.value.toString()->startsWith('ADAPTOR'))
    thisModule.createAdaptor(e.specification.value,vha_archi);
  }
  for (equation in glob_behavior.piecewiseFunctions) {
    thisModule.createConditionedEquation(equation, vhdl_archi);
  }
  --if (e.oclIsTypeOf(SYSML!Property)) {
  -- If it has a default value, create a generic
  --if (not e.defaultValue.oclIsUndefined())
  -- thisModule.createGeneric(e,vha_entity);
  -- If not, create a quantity
  --else
  thisModule.createQuantity(e,vha_archi);
  }
}

rule ArchitectureSynchronousDataFlow {
  from 
glob_behavior : GLOB!SDF
to 
vhdl_archi : VHDL!Architecture ()
do {
  vhdl_archi.ofEntity <- thisModule.getEntity(glob_behavior.owner);
  thisModule.getEntity(glob_behavior.owner).architectures <- vhdl_archi ;
}
}

rule ArchitectureStateMachines {
  from 
glob_behavior : GLOB!FSM
to
vhdl_archi : VHDL!Architecture ()
do {
vhdl_archi.ofEntity <- thisModule.getEntity(glob_behavior.owner);
thisModule.getEntity(glob_behavior.owner).architectures <- vhdl_archi ;
}

--rule block {

-- from
glob_module : GLOB!Module(not glob_module.behavior.oclIsKindOf(GLOB!Composite))
to

--vha_model : VHDL!Model,
-- vha_entity : VHDL!Entity (name <- glob_module.name), --.replaceAll(' ', '_')

-- vha_archi : VHDL!Architecture (name <- 'behavior')
do {
-- Save this model in a sequence
--thisModule.allEntities <- Sequence{thisModule.allEntities, vha_entity}->flatten();

--
--vha_model.entity <- vha_entity;
--vha_model.architecture <- vha_archi ;
--vha_archi.ofEntity <- vha_entity;
--vha_entity.architectures <- vha_archi ;

-- if (glob_module.behavior.oclIsTypeOf(GLOB!CT) or glob_module.behavior.oclIsTypeOf(GLOB!SDF) ) {
    for (equation in glob_module.behavior.equations) {
        --if (equation.pieces->size() > 1)
        -- thisModule.createConditionedEquation(equation, vha_archi);
        --else
        thisModule.createEquation(equation, vha_archi);
        -- If using the adaptor language, create a 'use' equation
        if (e.specification.value.toString() ->startsWith('ADAPTOR'))
            thisModule.createAdaptor(e.specification.value, vha_archi);

    }
    for (equation in glob_module.behavior.piecewiseFunctions) {
        thisModule.createConditionedEquation(equation, vha_archi);
    }

    if (eoclIsTypeOf(SYSML!Property)) {
        -- If it has a default value, create a generic
        if (not e.defaultValue.oclIsUndefined())
            thisModule.createGeneric(e, vha_entity);
        -- If not, create a quantity
        else
            thisModule.createQuantity(e, vha_archi);
    }
}


for (port in glob_module.interface.ports) {
    if (port.direction = 'inout')
        thisModule.createTerminal(port, vha_entity);
    if (glob_module.behavior.oclIsTypeOf(GLOB!SDF))
        thisModule.createPort(port, vha_entity);
}

rule createPort(port : GLOB!Port, vha_entity : VHDL!Entity) {
    to
    vha_port : VHDL!QuantityPort {
        name <- port.name,
        direction <- port.direction
    }
    do {
        vha_entity.ports <- vha_port;
    }
}

rule CompositeBlocks {
    from
    glob_module : GLOB!Module(glob_module.behavior.oclIsKindOf(GLOB!Composite))
    to
    vha_entity : VHDL!Entity (name <- glob_module.name), ---replaceAll(' ','_')
    vha_archi : VHDL!Architecture (name <- 'behavior')
    do {
        --- Save this model in a sequence
        --thisModule.allEntities <- Sequence(thisModule.allEntities, vha_entity)-->flatten();
        --- Save the proper references.
        --vha_archi.ofEntity <- vha_entity;
        --vha_entity.architectures <- vha_archi;
        for (instance in glob_module.behavior.instances) {
            self.createComponent(vha_archi,instance);
        }
    }
}

rule createComponent(architecture : VHDL!Architecture ,instance : GLOB!Module)
    to
    vha_comp : VHDL!Component (type <- thisModule.getEntity(instance))
    do {
        vha_comp.portMap;
architecture.components <- vha_comp;
}

rule createEquation(eq : GLOB!Expression, archi : VHDL!Architecture) {
    vha_equation : VHDL!Equation
do {
    vha_equation.lhs <- eq.pieces.first().value.name.split('=').first().
regexReplaceAll('\\^', '**');
    vha_equation.rhs <- eq.pieces.first().value.name.split('=').last( ).
regexReplaceAll('\\^', '**');
    vha_equation.lhs <- eq.name.split('=').first();
    vha_equation.rhs <- eq.name.split('=').last();
    archi.equations <- vha_equation; }
}

rule createConditionedEquation(eq : GLOB!PiecewiseFunction, archi : VHDL!
Architecture) {
    vha_equation : VHDL!IfUse( condition <- eq.pieces.first().interval.bottom .toString() + '<' +
        eq.argument.name + '<' +
        eq.pieces.first().interval.top.toString(),
        codeLines <- eq.name + '==' + eq.pieces.first().value.name
    ),
    vha_default : VHDL!Else(codeLines <- eq.name + '==' + eq.defaultValue)
do {
    Target object should look like this:
    —
    — if ug > usatp use
    — uout = usatp;
    — elsif ug < usatm use
    — uout = usatm;
    — else
    — uout = ug - ROUT*iout;
    — end use;
    for(piece in eq.pieces.excluding(eq.pieces.first())) {
        thisModule.addPiecewiseFunction{
            vha_equation,
            piece.interval.bottom.toString() + '<' + eq.argument.name + '<' +
            piece.interval.top.toString(),
            eq.name + '==' + piece.value.name );
        }
    vha_equation.default <- vha_default;
    archi.logicalObjects <- vha_equation;
}
}

rule addPiecewiseFunction(vha_ifUse : VHDL!IfUse, condition : String,
equation : String) {

to
    vha_elseUse : VHDL!ElseUse (condition <- condition.trim(),
        codeLines <- equation)
  do {
    vha_ifUse elseUses <- vha_elseUse;
  }
}

--- FSMSM

rule StateMachine {
  from
glob_stateMachine : GLOB!StateMachine
to
  -- Create the objects where we’ll save the states
vhdl_states : VHDL!EnumerationType {
    name <- 'stateType',
  }  -- Instances of Enumeration
vhdl_nst : VHDL!Signal{
    name <- 'next_state',
    type <- 'stateType',
}
vhdl_cst : VHDL!Signal{
    name <- 'current_state',
    type <- 'stateType',
}  -- A switch element
vhdl_case : VHDL!Case{
    var <- vhdl_cst,
}  -- Process: Next State logic
vhdl_proc_logic : VHDL!Process{
    name <- 'next_state_logic',
    triggers <- vhdl_cst,
    codePiece <- vhdl_case,
}  -- Process: Update state
vhdl_proc_set_next : VHDL!Process{
    name <- 'set_next_state',
    triggers <- vhdl_nst,
    codeLines <- 'current_state <= next_state'
}  do {
    -- TODO: Get rid of the "thisModule.aModule(uml_stm.owner)"
    -- this should be a simple variable "m"
    -- Should create a trigger for every input of the state machine.
    for (p in thisModule.getEntity(glob_stateMachine.owner.owner).ports) {
      if (p.direction.toString() = 'in') {
        vhdl_proc_logic.triggers <- p;
        --thisModule.createTrigger(vhdl_proc_logic,p);
      }
    }
    vhdl_proc_logic.codePiece <- vhdl_case;
}
thisModule.getEntity(glob_stateMachine.owner.owner).architectures.first().
.logicalObjects <- vhdl_states;  --Enumeration
thisModule.getEntity(glob_stateMachine.owner.owner).architectures.first().dataObjects <- vhdl_cst; — Signal
thisModule.getEntity(glob_stateMachine.owner.owner).architectures.first().dataObjects <- vhdl_nst; — Signal
thisModule.getEntity(glob_stateMachine.owner.owner).architectures.first().processes <- vhdl_proc_logic;
thisModule.getEntity(glob_stateMachine.owner.owner).architectures.first().processes <- vhdl_proc_set_next;

for (s in glob_stateMachine.states) {
    thisModule createState(s, vhdl_case, vhdl_states);
}

rule createTrigger(process : SC!Process, p : SC!Port) {
to
    vhdl_ev : VHDL!Event (name <- p.name)
do {
    process.triggers <- vhdl_ev;
}

— For every State, we have to add a Case to the Swhitch and a literal to the enumeration
rule createState(glob_state : GLOB!State, case : VHDL!Case, enum : VHDL!EnumerationType) {
to
    vhdl_when : VHDL!When {
        item <- glob_state.name,
        codeLines <- glob_state.invariant)
do {

    for (t in glob_state.transition) {
        thisModule.createIf(vhdl_when,t);
    }
    case.choices <- vhdl_when;
    enum.labels <- glob_state.name;
}

— Create the if clause corresponding to the transition
rule createIf(c : VHDL!When, t : GLOB!Transition) {
to
        name),
    vhdl_if : VHDL!If {
        condition <- t.guard.name.regexReplaceAll(‘=’, ‘==’),
        codeLines <- ‘next_state = ’ + t.target.name)
do {
    c.codePiece <- vhdl_if;
Appendix C. Code: SysML to All

\[
\text{337) }
\]

\[
\text{-----------------------------------------------}
\]
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


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


