Design of a virtual prototyping framework for composable heterogeneous systems

Cédric Ben Aoun

THÈSE DE DOCTORAT DE
L'UNIVERSITÉ PIERRE ET MARIE CURIE

Spécialité

Informatique

École doctorale Informatique, Télécommunication et Électronique (Paris)

Présentée par

Cédric BEN AOUN

Pour obtenir le grade de :

DOCTEUR de l'UNIVERSITÉ PIERRE ET MARIE CURIE

Sujet de la thèse :

Principes et réalisation d'un environnement de prototypage virtuel de systèmes hétérogènes composables

soutenue le 12 Juillet 2017
devant le jury composé de :

M. Frédéric Pétrot	Rapporteur	ENSIMAG-TIMA/SLS
Mme. Cécile Belleudy	Rapporteur	Université de Nice-Sophia Antipolis
M. Ian O'Connor	Examinateur	Ecole Centrale Lyon
M. Matthieu Moy	Examinateur	Laboratoire Verimag
M. Filipe Vinci dos Santos	Examinateur	Ecole Centrale-Supelec
Mme. Emmanuelle Encrenaz	Examinateur	Université Pierre et Marie Curie
Mme. Marie-Minerve Louërat	Examinateur	Université Pierre et Marie Curie
M. François Pêcheux	Directeur de thèse	Université Pierre et Marie Curie

Abstract

Current and future microelectronics systems are more and more complex. In a aim to bridge the gap between the cyber/digital world and the physical world in which we evolve we observe the emergence of multi-disciplinary systems that interact more and more with their close surrounding environment. The conception of such systems requires the knowledge of multiple scientific disciplines (electrical, optical, thermal, mechanical, acoustic, chemical or biological) which tends to define them as heterogeneous systems. Designers of the upcoming digital-centric More-than-Moore systems are lacking a common design and simulation environment able to efficiently manage all the multi-disciplinary aspects of its components of various nature, which closely interact with each other.

In this thesis we explore the possibilities of developing and deploying a unified SystemC-based design environment for virtual prototyping of heterogeneous systems. In order to overcome the challenges related to their specification and dimensioning this environment must be able to simulate a complex heterogeneous system as a whole, for which each component is described and solved using the most appropriate Model of Computation (MoC).

We propose a simulator prototype called SystemC Multi Disciplinary Virtual Prototyping (MDVP) which is implemented as an extension of SystemC. It follows a correct-by-construction approach, relies on a hierarchical heterogeneity representation and interaction mechanisms with master-slave semantics in order to model heterogeneous systems. Generic algorithms allow for the elaboration, simulation and monitoring of such systems.

We also provide a methodology to incorporate new Models of Computation within the SystemC MDVP environment. We follow this methodology to integrate a Smoothed Particle Hydrodynamics (SPH) MoC that allows for the description of fluidic network. This MoC is then used to model a prototype of a point-of-care blood analysis system.

Eventually, we realized a case study of a passive RFID reading system that requires several interacting MoCs in order to be modeled. We compare the simulation results with measures acquired on a real physical prototype of a passive RFID reading system.

Résumé

Les systèmes électroniques d'aujourd'hui et de demain sont de plus en plus complexes. Dans le but de rapprocher le monde numérique et le monde physique dans lequel nous évoluons, nous observons l'émergence de systèmes multidisciplinaires qui interagissent de plus en plus avec leur environnement proche. La conception de tels systèmes nécessite la connaissance de multiples disciplines scientifiques (électronique, optique, thermique, mécanique, acoustique, chimie ou biologie) ce qui tend à les définir comme étant des systèmes hétérogènes. Pour le développement de ces systèmes à venir, il manque aux concepteurs un environnement de conception et de simulation commun permettant de gérer efficacement la multidisciplinarité de ces composants de natures variées qui interagissent fortement les uns avec les autres.

Dans cette thèse nous explorons la possibilité de développer et déployer un environnement de conception unifié, basé sur SystemC, pour le prototypage virtuel de systèmes hétérogènes. Afin de surpasser les contraintes liées à leur spécification et dimensionnement, cet environnement doit pouvoir simuler un système hétérogène dans son ensemble, dans lequel chaque composant est décrit et résolu en utilisant le Modèle de Calcul (MoC) le plus approprié.

Nous proposons un prototype de simulateur, appelé SystemC Multi Disciplinary Virtual Prototyping (MDVP), qui est implémenté comme une extension de SystemC. Il suit une approche correcte-par-construction, repose sur une représentation hiérarchique de l'hétérogénéité et sur un mécanisme d'interaction basé sur des sémantiques maitre-esclave afin de modéliser les systèmes hétérogènes. Des algorithmes génériques permettent l'élaboration, la simulation et le monitoring de tels systèmes.

Nous proposons également une méthodologie afin d'incorporer de nouveaux Modèles de Calcul au sein de l'environnement SystemC MDVP. Cette méthodologie est suivie dans le but d'ajouter à SystemC MDVP le MoC Smoothed Particle Hydrodynamics (SPH) qui permet la description de réseaux fluidique. Ce MoC est ensuite utilisé pour modéliser un prototype de dispositif permettant l'analyse de sang sur un lieu d'intervention.

Nous avons finalement réalisé un cas d'étude portant sur un système RFID passif qui nécessite, afin d'être modélisé, l'utilisation de plusieurs MoCs interagissant entre eux. Les résultats obtenus en simulation sont comparés avec des mesures acquises sur un vrai prototype physique.

Contents

List of Figures **XI**

List of Tables **XIII**

List of Algorithms **XV**

List of Listings **XVII**

1 Introduction **1**
1.1 Context . . . 2
1.2 Thesis Organization . . . 5

2 Problem Statement **7**
2.1 Introduction . . . 8
2.2 Heterogeneous Systems Virtual Prototyping Challenges . . . 8
2.2.1 Smooth Management of Heterogeneity . . . 8
2.2.2 Sound Management of Interacting Entities . . . 9
2.2.3 Flexible Virtual Prototyping Environment . . . 9
2.2.4 Multi-Disciplinary Monitoring . . . 10
2.3 Contributions . . . 10
2.4 Conclusion . . . 12

3 Related Work **13**
3.1 Introduction . . . 14
3.2 Coupled Simulation . . . 14
3.3 Frameworks . . . 16
3.3.1 Ptolemy II . . . 16
3.3.2 ModHel'X . . . 17
3.3.3 Modelica . . . 17
3.3.4 Matlab . . . 18
3.3.5 Metropolis . . . 19
3.4 SystemC-based Frameworks . . . 20
3.4.1 SystemC-A . . . 22
3.4.2 SystemC-H . . . 23
3.4.3 HetSC . . . 23
3.4.4 SystemC AMS . . . 23
3.5 Multi Disciplinary Monitoring Mechanism . . . 25
3.5.1 Aspect-Oriented Programming (AOP) . . . 26
3.5.2 LLVM - Clang . . . 28
3.6 Conclusion . . . 28

4 SystemC MDVP: Principles **31**
4.1 Introduction . . . 32
4.2 Models of Computation . . . 33
4.2.1 Discrete Event (DE) . . . 34
4.2.2 Timed Data Flow (TDF) . . . 35
4.2.3 Bond Graph (BG) . . . 36
4.2.4 Electrical Network (EN) . . . 36
4.2.5 Abstract representation of a MoC . . . 37
4.3 Interaction Mechanism . . . 38
4.4 MDVP Hierarchical Approach . . . 43
4.5 User Profiles . . . 45
4.5.1 Simulator Architect . . . 45
4.5.2 MoC Architect . . . 46
4.5.3 SoC Architect . . . 48
4.6 Conclusion . . . 48

5 SystemC MDVP: Implementation **51**
5.1 Introduction . . . 52
5.2 SystemC MDVP Kernel Representation . . . 52
5.2.1 SystemC MDVP MoC Abstraction . . . 53
5.2.2 SystemC MDVP core classes . . . 55
5.3 Elaboration Phase . . . 57
5.3.1 Composability / Static Analysis Sub-phase . . . 58
5.3.2 Clustering Sub-phase . . . 62
5.3.3 Solver Instantiation Sub-phase . . . 67
5.3.4 Basic Behavior Block Elaboration Sub-phase . . . 70
5.3.5 Port and Channel Elaboration Sub-phase . . . 72
5.3.6 Elaboration conclusion . . . 73
5.4 Simulation Phase . . . 74
5.4.1 Simulation Mechanism . . . 74
5.4.2 Simulation Opportunities . . . 77
5.5 Conclusion . . . 78

6 SystemC MDVP: Monitoring **81**
6.1 Introduction . . . 82
6.2 Principles . . . 83
6.2.1 SystemC Monitoring . . . 84
6.2.2 SystemC MDVP Monitoring . . . 86
6.3 Monitoring Mechanism . . . 87
6.3.1 Monitor Handler Instantiation . . . 88
6.3.2 Monitor Slot Instantiation . . . 89
6.3.3 Initialization . . . 91
6.3.4 kernel Routine . . . 94
6.3.5 End of Simulation . . . 96
6.4 Conclusion . . . 96

7 New MoC Integration Methodology **99**
7.1 Introduction . . . 100
7.2 New MoC integration methodology . . . 100
7.2.1 Interfacing Step . . . 101
7.2.2 Implementation Step . . . 105
7.2.3 Interaction Step . . . 106

7.3 Application to SPH MoC . . . 108
7.3.1 Interfacing Step . . . 109
7.3.2 Implementation Step . . . 110
7.3.3 Interaction Step . . . 114
7.4 Conclusion . . . 114

8 Validation Case Studies **117**
8.1 Introduction . . . 118
8.2 Application Based on SPH . . . 118
8.2.1 Case Study Description . . . 118
8.2.2 Results . . . 120
8.3 Application based on RFID . . . 124
8.3.1 Case Study Description . . . 124
8.3.2 Results . . . 129
8.4 Conclusion . . . 135

9 Conclusion **137**
9.1 Conclusion . . . 138
9.2 Perspective . . . 140

Publications **143**

Bibliography **151**

List of Figures

1.1 Exploded view of an Apple Watch . . . 2
1.2 Apple Watch chip's layout . . . 3
1.3 System on Chip as a set of interacting disciplines. . . . 4

3.1 Hierarchical Modeling in Ptolemy II . . . 16
3.2 Design languages and their main purpose . . . 20
3.3 SystemC Kernel Phases . . . 21
3.4 SystemC AMS language Standard Architecture, adapted from [52]. . . . 24
3.5 Classical Object Oriented Programing. . . . 26
3.6 Aspect Oriented Programing. . . . 26
3.7 Aspect Oriented Programing Mechanism. . . . 27
3.8 Influence of other Frameworks on SystemC MDVP. . . . 29

4.1 Running Example: System on Chip representing a 3-axis vibration sensor. . . . 33
4.2 Simple model described with DE MoC . . . 34
4.3 Simple model described with TDF MoC . . . 35
4.4 Simple model described with BG MoC . . . 36
4.5 Simple electrical network described with EN MoC . . . 37
4.6 Interaction between two MoCs . . . 39
4.7 Authorized (a, b and c) and Forbidden (d, e and f) master-slave relationships . . . 40
4.8 System on Chip representing a 3-axis vibration sensor as an assembly of MoCs. . . . 41
4.9 Running Example: Sensor description . . . 42
4.10 Running Example: Scope of EN, TDF and DE MoCs . . . 44
4.11 SystemC MDVP: Simulator Architect . . . 45
4.12 SystemC MDVP: MoC Architect . . . 47
4.13 SystemC MDVP: SoC Architect . . . 49

5.1 SystemC MDVP base classes to abstract a MoC . . . 53
5.2 SystemC MDVP intrinsic classes . . . 56
5.3 SystemC MDVP kernel elaboration phases. . . . 57
5.4 Running Example: Improved with units measures . . . 61
5.5 Representation of a Cluster . . . 62
5.6 Running Example: Cluster Tree associated with the vibration sensor . . . 65
5.7 Running Example: Branch of the cluster Tree with instantiated solver . . . 69
5.8 Running Example: solver abstraction of the vibration sensor . . . 70
5.9 Running Example: Minimal class hierarchy to perform the elaboration of all the primitive blocks. . . . 71
5.10 SystemC MDVP kernel simulation phases. . . . 75
5.11 Running Example: simulation process. . . . 76
5.12 Activity Diagram of a roll back algorithm. . . . 77

5.13 Activity Diagram of a postponed execution algorithm. 78
5.14 Activity Diagram of a multiple execution algorithm. 78

6.1 SystemC MDVP Monitor classes 83
6.2 Running Example: Monitoring tree associated with the vibration sensor. 84
6.3 SystemC MDVP kernel handling of SystemC monitoring. 85
6.4 SystemC MDVP kernel monitoring principle. 86
6.5 SystemC MDVP kernel monitoring phases. 88
6.6 SystemC MDVP monitoring kernel routine. 94
6.7 Running Example: Monitoring execution for the vibration sensor. 95
6.8 Running Example: Simulation Traces for the `Sensor X` of the vibration sensor . . 97

7.1 MoC Integration Process 101
7.2 Inheritance pattern to define behavior of a new MoC. 102
7.3 Inheritance pattern to define communication channel of a new MoC. 102
7.4 Inheritance pattern to define composition mechanism of a new MoC. 103
7.5 Inheritance pattern to define monitoring of a new MoC. 104
7.6 Inheritance pattern to define an interface solver within a new MoC. 106
7.7 Inheritance pattern to define converter ports within a new MoC. 107
7.8 SPH modeling primitives. 109
7.9 Smoothing Kernel and support radius in SPH. 110
7.10 SPH – TDF transducer. 114
7.11 SPH MoC inheritance diagram. 116

8.1 Micro-fluidic Network. 119
8.2 SPH micro-fluidic chip. 120
8.3 Experimental, PFN and SPH results for constant flow. 121
8.4 Experimental, PFN and SPH results for pressure at 200 Pa. 122
8.5 Experimental, PFN and SPH results for pressure at 400 Pa. 122
8.6 Experimental, PFN and SPH results for Methanol at 400 Pa. 123
8.7 A passive Radio Frequency Identification (RFID) system. 125
8.8 Simple representation of the RFID system modeled. 126
8.9 Detailed representation of the RFID transceiver modeled. 127
8.10 Detailed representation of the RFID transponder modeled. 128
8.11 Model of the whole passive RFID reading system. 129
8.12 Existing physical RFID tag reader. 130
8.13 Measures on the physical prototype during the transmission of a tag marker. . . . 132
8.14 Closer look on the measures on the physical prototype during the transmission of a tag marker. 132
8.15 Simulation results for the transmission of a tag marker with the virtual RFID prototype. 133
8.16 Closer look on the simulation results for the transmission of a tag marker with the virtual RFID prototype. 133
8.17 Simulation time to emit a RFID tag depending of the simulation timestep. 134

List of Tables

5.1 Dictionary of solvers available for the clusters hierarchy shown in Figure 5.6. . . . 67

8.1 Modeling and Simulation times. 124
8.2 Parameters values associated with the circuit shown in Figure 8.9. 126
8.3 Parameters values associated with the circuit shown in Figure 8.10. 127
8.4 Simulation time of the passive RFID reading system. 135

List of Algorithms

5.1 Recursive clustering algorithm. 64
5.2 Recursive solver instantiation algorithm. 68
5.3 Elaboration of MoC interfaces algorithm. 71
5.4 Elaboration of ports and channels algorithm. 73
5.5 Generic Elaboration Algorithm. 74

6.1 Generic Algorithm to Instantiate Monitor Handlers. 89
6.2 Generic Recursive Algorithm to Create Monitor Slots. 90
6.3 Generic Algorithm to Instantiate Monitor Slots. 91
6.4 Recursive Algorithm to Initialize the Monitor Slots. 93

7.1 SPH Simulation Loop. 112

Listings

5.1 Definition of physical types 60
6.1 End-user monitoring specification 91
7.1 Code snippet of SPH primitives composition 113

Chapter

Introduction

Contents

1.1 Context 2
1.2 Thesis Organization 5

1.1 Context

Nowadays, the design of new embedded systems relies on the sound assembly of different IPs, each IP being developed separately and independently. The conception of each of these IPs may involve different physical domains. Figure 1.1 represents an exploded view of an Apple Watch as an example of the IP assembling performed in current available devices. The variety of physical domains that may be involved is quite well-represented. Different kinds of sensors (force touch, optical pulse) are integrated in the system as well as analogical or electronic components (battery, wireless charging coil, antenna). Purely digital systems no longer exist.

Figure 1.1: Exploded view of an Apple Watch, source from [1].

If we take a closer look on the Apple Watch's chip, we notice that even the chip itself is no longer fully digital. The chip layout is illustrated in Figure 1.2; this figure highlights some of the components embedded in the chip. In addition to the digital components (processor, memory) and different controllers for the aforementioned sensors (which are also digital) we find non-digital components such as the wireless charger or several Micro-Electro-Mechanical Systems (MEMS) that act as accelerometer and gyroscope.

According to this example, design engineers who want to build tomorrow's more-than-Moore embedded systems must think, create and design differently than they do today. Electronic design automation tooling must evolve with the upcoming needs and even go so far as to anticipate them. Current and future microelectronics systems are increasingly complex and interact more and more with their close surrounding environment. The conception of such systems requires the knowledge of multiple scientific disciplines (electrical, optical, thermal, mechanical, acoustic,

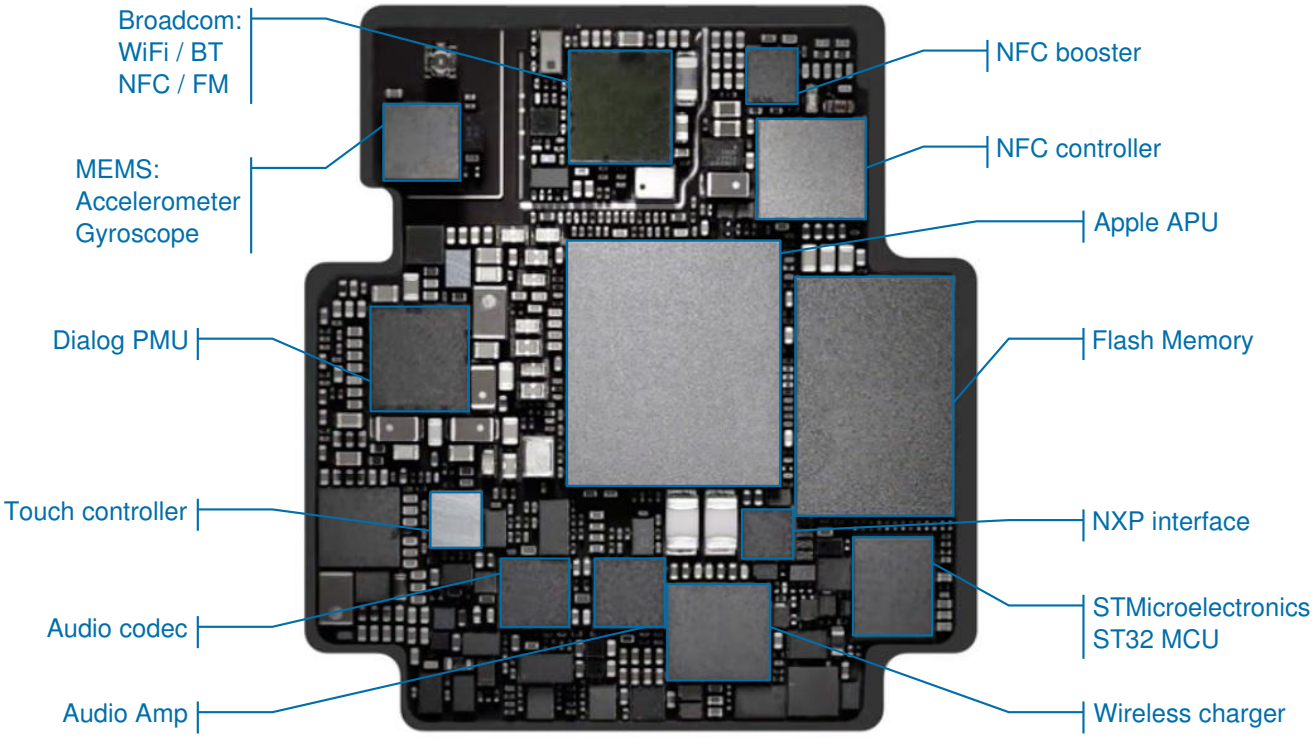

Figure 1.2: Apple Watch chip's layout, adapted from [2, 3, 4].

chemical or biological).

The emergence of multi-disciplinary systems is the result of the desire to bridge the gap between the cyber/digital world and the physical world in which we evolve. This approach leads to the concept of Cyber-Physical Systems (CPS) [5] and Internet-of-Things (IoT) [6].

We are now witnessing the evolution of an ecosystem of billions of interconnected devices[1] [7]. These systems communicate, interact with and hence impact their environment in a multitude of ways. The use of sensors allows such devices to probe and interact with their immediate environment thus collecting information, whilst the role of actuators is to apply constraints and forces to the real world. Sensors and actuators act as the interface at the border of the physical and the cyber world.

Figure 1.3 provides a generic representation of a System on Chip (SoC) and highlights several interacting disciplines that may be involved in the design of an embedded system. One can see that the SoC device embeds digital components (processor, memory, peripheral) as well as components from other engineering disciplines such as RF transceiver and sensors (MEMS, optical, biological). This figure illustrates how a system can interact with its environment.

New types of emerging applications requiring microelectronics that closely interact with the surrounding environment in different physical domains (optical, mechanical, acoustical, biological, etc.) are referred to as *heterogeneous systems*. They qualify as heterogeneous because

[1]Gartner, Inc. forecasts that 6.4 billion connected things will be in use worldwide in 2016, up 30 percent from 2015, and will reach 20.8 billion by 2020. In 2016, 5.5 million new things will get connected every day.

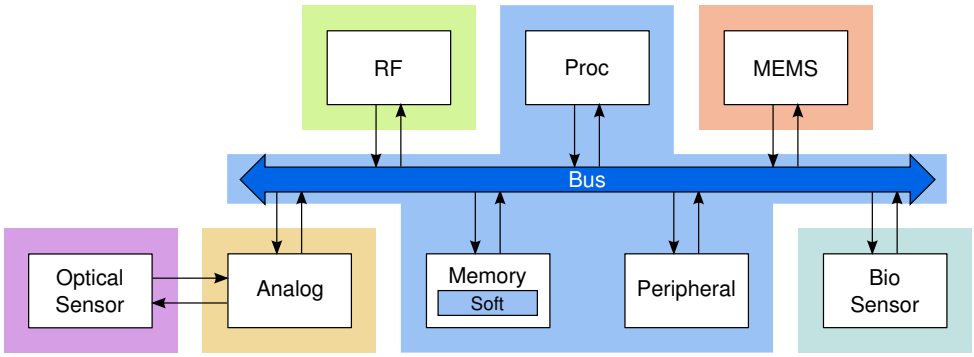

Figure 1.3: System on Chip as a set of interacting disciplines.

they couple digital and physical elements. Moreover, they describe different semantics and time abstraction. Since these systems represent non-homogeneous applications, various physical modeling environments may be used to model and simulate them.

The design of these multi-disciplinary microelectronics-assisted systems is often an iterative process in which the individual parts for each physical domain are developed independently and then combined in the very last stage to realize the final product. Very often, design errors such as functional incorrectness, wrong interfaces, or non-compliance with the initial product specifications are identified too late in the design cycle. This leads to additional design spins and delayed schedules as a result of a necessary reimplementation which hampers the whole market introduction process. It is therefore important to allow the modeling and the simulation of such systems as a whole to truly appreciate their complexity before their expensive fabrication.

Clearly, the main issue is that a global system representation including all involved physical/engineering domains is missing, especially in the early design stages. In such a context, it would be highly beneficial to prove the correctness of the heterogeneous system architecture up-front and to have an accurate view of the way heterogeneous entities interact as a whole. This requires a modeling/design, simulation, and verification environment that can assist system designers to dimension, partition, and thus to "architect" such heterogeneous systems appropriately. Such a solution must be flexible and open in order to allow for the extension of the environment; extending the framework should not involve complex interfacing mechanisms with the existing infrastructure.

To meet these requirements, prior to constructing physical prototypes, virtual prototyping at a high level of abstraction becomes inevitable, especially for such systems which feature a tight coordination between the computational and physical elements. Virtual prototyping provides numerous functionalities and benefits when it comes to the design and conception of new systems. This mechanism consists in the creation of virtual models to describe a design, allowing for the performance of software simulation of the device under development [8, 9]. Virtual prototyping allows for architectural exploration with the possibility of studying several design alternatives quickly and easily. Furthermore, it provides the designer with the possibility to carry out performance analysis and early test design on the developed system, leading to the capacity

to endure testing and validation procedures usually not achieved with a real physical prototype platform due to the cost of such procedures.

Eventually, since virtual prototyping enables the simulation of heterogeneous systems in the early stage of development, it can enable the development of the software associated with the platform prior to the availability of a physical prototype.

This thesis is performed in the frame of the CATRENE European project - Heterogeneous INCEPTION (H-Inception) [10]. The main goal of this project is to develop and deploy a unified design environment for virtual prototyping of multi-domain microelectronics-assisted systems to overcome the challenges related to their specification, dimensioning and verification. This project aims to benefit the European industry in the production of their products with application in several domains such as automotive, wireless, avionics and biomedical areas.

1.2 Thesis Organization

After presenting, in Chapter 1, the context in which this work is performed, the document is organized as follows:

In Chapter 2, the challenges related to the simulation of heterogeneous systems are highlighted; they represent the different problem statements that need to be addressed. They cover a wide variety of issues ranging from the characterization of the simulation environment to the monitoring of such systems. Requirements that need to be met in order to ensure a correct simulation are introduced. The contributions brought together in the frame of this thesis, in order to address these challenges, will then be presented.

In Chapter 3, the state-of-the-art related to the simulation of heterogeneous systems is summarized. We present several approaches that address the challenges described in Chapter 2. We describe different frameworks and design environments which allow for the simulation of multi-physical systems. For each framework, the modeling and the behavior of the framework are discussed.

In Chapter 4, we introduce our solution for the simulation of heterogeneous systems. Our work leads to the conception of a new simulator prototype called SystemC MDVP. This chapter focuses on the principles that are the foundations of our framework. We present the different abstraction and representation defined within SystemC MDVP. We explain the hierarchical representation of the system we chose and the underlying semantics, which lead to an environment permitting the simulation of real software coupled with virtual hardware architecture and virtual physical models.

In Chapter 5, we present the implementation details underlying SystemC MDVP that support the principles introduced in Chapter 4. SystemC MDVP is implemented as an extension of SystemC which allows us to achieve the simulation of heterogeneous systems. This chapter presents

the different data structures and generic algorithms developed within our framework.

Chapter 6 describes the mechanism defined within SystemC MDVP that allows for the monitoring of multi-disciplinary systems. We present an approach wherein we provide a unified access to the information through the whole system. Information related to digital or physical components of the heterogeneous system are addressed the same way. This chapter presents the principles and the implementation details underlying the monitoring mechanism of our framework.

In Chapter 7, we present a methodology to allow the addition of new MoCs within our virtual prototyping environment. This methodology relies on the principles defined within SystemC MDVP and described in Chapter 4. To support this methodology and our simulator, we present a new MoC called SPH. This MoC allows the modeling and the simulation of fluidic elements through the description of a fluidic network.

Chapter 8 presents two validation case studies to illustrate the possibilities offered by SystemC MDVP and support the principles underlying our framework. First, we describe a case study related to the MoC SPH where a fluidic network is described, simulated and then compared with other solutions. Second, we depict a complete passive RFID reading system involving several MoCs in order to realize its modeling. The RFID system is modeled, simulated and then compared to a real physical prototype of the passive RFID reading system.

Finally, by way of a conclusion, in Chapter 9 we resume the work presented and introduce perspectives on future work.

Chapter

2 *Problem Statement*

Contents

2.1 Introduction 8

2.2 Heterogeneous Systems Virtual Prototyping Challenges 8

2.2.1 Smooth Management of Heterogeneity 8

2.2.2 Sound Management of Interacting Entities 9

2.2.3 Flexible Virtual Prototyping Environment 9

2.2.4 Multi-Disciplinary Monitoring 10

2.3 Contributions 10

2.4 Conclusion 12

2.1 Introduction

Current and future microelectronics systems are increasingly complex, and interact more and more with their immediate environment. These emerging systems are intrinsically complex in their development and require a lot of time and effort to achieve their conception.

Since the development of multi-disciplinary microelectronics assisted systems involves several disciplines, the individual parts are usually conceived independently and assembled in the final stage of the development process. In such a context, in order to truly appreciate the complexity of multi-disciplinary systems before their expensive fabrication, it is important to allow the modeling and the simulation of such systems as a whole.

With such an approach, we would benefit from an accurate view of a heterogeneous system including the manner in which heterogeneous entities interact with one another. In addition to this, a global system representation including all involved physical/engineering domains, especially in the early design stages, would be beneficial to prove the correctness of the heterogeneous system architecture up-front. In order to assist system designers in the design process of heterogeneous systems, i.e. to help them to dimension, partition and thus to "architect" these systems, a modeling/design, simulation and verification environment is needed.

The simulation of heterogeneous systems raises specific challenges that may be difficult to address due to their very nature. These challenges are presented in Section 2.2. To address them, Section 2.3 introduces the contributions presented in the framework of this thesis. Subsequently, Section 2.4 concludes this chapter and provides an overview of the challenges faced and the resulting contributions.

2.2 Heterogeneous Systems Virtual Prototyping Challenges

How does one simulate a heterogeneous system? The answer is not trivial. The heterogeneity aspect can be difficult to address due to the gap between the different disciplines involved in the design of heterogeneous systems (e.g. software engineering vs biological engineering).

We have identified several challenges which represent the different issues that may occur when dealing with the simulation of heterogeneous systems and the requirements induced by these issues.

2.2.1 Smooth Management of Heterogeneity

In the context of heterogeneous systems, one key aspect is to define what it is intended by the term *heterogeneous*. Heterogeneity can be defined as the use of several simulation tools, or the use

of different languages, semantics, disciplines, etc. In such a context, a clear definition of the term heterogeneity and the composition of this heterogeneity are required.

These components (entity), which defined the heterogeneity, must be well-defined, not only from the simulator viewpoint, but also from the user viewpoint. The simulator requires a well-known distinction between components in order to correctly simulate each component within its associated environment. The user should have access to well-defined components in order to correctly and easily design the system he wants to simulate. Our approach relies on the definition of these heterogeneous entities by means of interacting Models of Computation (MoCs). We must therefore provide a well-defined interface for these MoCs to interact with the kernel, without neglecting the user's interface.

2.2.2 Sound Management of Interacting Entities

Once heterogeneity has been clearly defined, one of the key challenges lies in the interaction between the entities that define this heterogeneity and their composition. They usually express different semantics, with potentially different time abstraction. Incompatibility between physical domains or design errors should be detected by the simulator. It is necessary to be able to identify when two components are connected together when they should not be.

As such, we need a well-defined interaction mechanism between different heterogeneous entities. It should express the constraints associated with each entity and, consequently, it should enable and define their composition. Interaction is not limited to the composition of the heterogeneous entities; it is also required to be able to express semantics information if needed. This information is associated with the data handled within each entity (such as dimension or unit).

2.2.3 Flexible Virtual Prototyping Environment

Since multi-physical systems represent non-homogeneous applications, various physical modeling environments may be used to model and simulate them. Multiple approaches exist to perform simulation of heterogeneous systems, from a global simulation framework to the use of multiple specific simulation tools dedicated to each discipline. This situation can be simply summarized as the choice between a unified, unique simulation environment and a co-simulation environment.

Both approaches have their pros and cons, but we believe performing the simulation of the whole system within the same simulation environment allows us to truly appreciate the dependencies and all the interactions involved in the system. It is crucial that each discipline take part in the design exploration phase of the system. Consequently we have to set up an environment which enables the coupling of all the disciplines within the same simulation framework.

In the scope of heterogeneous systems simulation, we have to provide a highly flexible

framework. Flexibility is understood, in this work, in the sense that the simulator must be able to evolve and to simulate any heterogeneous system. Naturally, we want to perform the simulation of a system which involves several heterogeneous entities, therefore we cannot afford to provide a rigid, static simulator.

Heterogeneous entities which may take part in the conception of a multi-physical system are numerous, ranging from biological to optical through mechanical, etc., without forgetting the different semantics or time abstraction associated. It seems unlikely that we would be able to define every existing heterogeneous entity and, furthermore, to define those which do not yet exist. In consequence, heterogeneous simulation requires a way to easily integrate new heterogeneous entities into the simulator, in a convenient manner.

2.2.4 Multi-Disciplinary Monitoring

In the context of systems simulation and, more specifically, in the case of heterogeneous systems, the monitoring mechanism constitutes an important feature of the simulator. This mechanism must adapt to, and fit, the different digital/physical parts involved in the design. With the term *monitoring*, we refer to a mechanism which aims to observe and record information about the signals of a system, independently of the outcome of this observation (tracing, profiling, etc.).

The information associated with a digital component (such as a micro-controller) is different from that associated with an analog component (such as a thermal sensor). The relevant data may vary from one domain to another and the way to express them may also differ. Thus, setting up the monitoring mechanism of such systems represents a challenge in itself since it is dependent on the domain modeled, and yet it must remain a generic mechanism in order to manage all the different disciplines and ensure the flexibility of the simulator.

2.3 Contributions

The simulation of multi-disciplinary systems is not trivial. Bringing together different physical/engineering disciplines with different semantics is rather difficult. Designing a system that integrates digital and analog parts becomes a complicated process, where heterogeneity represents a key issue.

The objective of this thesis is to explore the possibilities of the simulation and the composition of digital centric multi-disciplinary systems. The contributions brought in the framework of this work are listed below.

- Creation of a reliable framework for the virtual prototyping of heterogeneous systems (SystemC MDVP) based on interoperable Models of Computation (MoCs).

Essentially, we can break down the design of a simulator into two aspects: composition and synchronization. In this thesis, we address the challenges linked to the compositional aspects, some of the synchronization aspects are addressed in another work [11]. We establish a compositional analysis in order to provide a correct-by-construction approach. This analysis relies on validation checks and the definition of interacting mechanisms between MoCs. Within our simulation framework, all the mechanisms provided are generic to guarantee the flexibility of the simulator.

When dealing with heterogeneity, we have to identify the border between the different disciplines involved in the system. We must perform compatibility checks between these interacting disciplines and also provide a mechanism to express the specific semantics associated with a discipline such as dimension and units. The compositional analysis that must be performed should be carried out in the beginning of the life cycle of the simulator in order to ensure a sane structure for the followings simulation steps. The development of the simulator is a joint work with the PhD student in charge of the synchronization [11].

- Definition of a new Model of Computation: Smoothed Particle Hydrodynamics (SPH).

In order to support and verify the principles and mechanisms proposed within our framework SystemC MDVP, we need a various set of several Models of Computation. In the framework of the H-Inception project, the following MoCs were developed by persons involved in the project: Timed Data Flow (TDF) [11], Electrical Network (EN) [12] and Ordinary Differential Equations (ODE).

As part of this thesis, we have accomplished the development of a Model of Computation called SPH, which allows for fluidic networks to be described. The SPH MoC enables the expression of a physical discipline, with its unit and dimension. It elicits a good understanding of the complex interactions between several disciplines. This MoC is included in a proof-of-concept application, which aims to prototype a Lab-on-Chip through a point-of-care blood analysis system.

- Development of a generic multi-disciplinary monitoring mechanism.

A global and unified framework for the simulation of heterogeneous systems must provide an efficient monitoring mechanism. In order to support the scalability of the framework, i.e. to handle the upcoming disciplines, the monitoring capabilities must be generic. As such, its development passes through the definition of an internal abstraction within the framework and a generic interface to enable the expression of the discipline's specificities.

2.4 Conclusion

Keeping in mind the objective of performing the simulation of multi-disciplinary systems, specific issues related to the very nature of these systems have been raised. The simulation of heterogeneous systems must be based on a solid set of principals and must rely on a convenient and flexible infrastructure It requires the provision of a clear definition of the entities involved in the simulation, that is to say a clear definition within the simulation tool and also from the end-user perspective. It must be supported by a strong and efficient interaction mechanism, allowing for a high level of flexibility and, therefore, the capacity to enhance the simulator with new entities. Finally, an efficient monitoring mechanism that fits the specificities of multi-disciplinary systems must be provided. The difficulty, when you wish to perform the simulation of heterogeneous systems, lies in the method of addressing these challenges together in order to meet the aforementioned requirements.

After describing the contributions of this thesis in order to tackle the challenges raised by the simulation of heterogeneous systems, different approaches to address these issues are discussed in Chapter 3. The following chapters discuss the solutions provided in this thesis in more detail. We introduce our simulation framework SystemC MDVP and the underlying mechanisms (Chapter 4 and Chapter 5), we clearly define the simulation environment and the notion of heterogeneity before describing our interaction mechanism that fits the purpose of simulating heterogeneous systems, without forgetting our multi-disciplinary monitoring mechanism (Chapter 6). We subsequently demonstrate the flexibility of our solution with the integration of a new MoC (Chapter 7), followed by a case study to illustrate the efficiency of our solution (Chapter 8)

Chapter

3 *Related Work*

Contents

3.1 Introduction . . . 14
3.2 Coupled Simulation . . . 14
3.3 Frameworks . . . 16
3.3.1 Ptolemy II . . . 16
3.3.2 ModHel'X . . . 17
3.3.3 Modelica . . . 17
3.3.4 Matlab . . . 18
3.3.5 Metropolis . . . 19
3.4 SystemC-based Frameworks . . . 20
3.4.1 SystemC-A . . . 22
3.4.2 SystemC-H . . . 23
3.4.3 HetSC . . . 23
3.4.4 SystemC AMS . . . 23
3.5 Multi Disciplinary Monitoring Mechanism . . . 25
3.5.1 Aspect-Oriented Programming (AOP) . . . 26
3.5.2 LLVM - Clang . . . 28
3.6 Conclusion . . . 28

3.1 Introduction

This chapter presents a non-exhaustive state-of-the-art concerning the different approaches that allow us to simulate multi-disciplinary systems. This state-of-the-art aims at highlighting the different features required in order to design a virtual prototyping environment for heterogeneous systems.

Section 3.2 introduces a kind of simulation called coupled simulation, where the purpose is to describe different parts of the system to model using different tools and to simulate them in their respective dedicated simulator. We especially detail the *Functional Mock-up Interface (FMI)* standard which follows this approach.

Section 3.3 puts forward different simulation frameworks which aim at performing the simulation of heterogeneous systems. *Ptolemy II*, considered as the pioneer in the field of heterogeneous simulation, is a software environment based on a hierarchical heterogeneous approach. *Metropolis* is based on meta-models and promotes a reusability approach in order to support the simulation of embedded heterogeneous systems. *Modhel'X* relies on the association of sub-models described using different modeling languages. *Matlab*, coupled with *Simulink*, constitutes a commercial solution to perform multi-physical simulation. *Modelica* is an object-oriented language for hierarchical physical modeling.

In Section 3.4 we explore existing SystemC-based frameworks which intended to extend its capacities in order to perform analog simulation. We outlines two sets of frameworks, the first one gathers the frameworks which modify the SystemC kernel to enhance the simulator with analog capacities: *HetSC* and *SystemC-A*. The other set gathers those which do not alter the SystemC kernel: *SystemC-H* and *SystemC AMS*.

In Section 3.5 we discuss different technologies that could have been used in order to achieve the monitoring of multi-disciplinary systems such as *Aspect-Oriented Programming* and *LLVM/Clang* approaches.

Thereafer, Section 3.6 provides an overview of the state-of-the-art presented and concludes this chapter.

3.2 Coupled Simulation

The *Coupled Simulation* approach, also referred as *co-simulation*, consists in modeling and simulating a system composed of different subsystems in a distributed manner; these different subsystems form a coupled problem. The co-simulation intends to couple different tools in a co-simulation environment. The idea is that each model which makes up the system is developed with different tools and is simulated independently from the other models in its own simulator.

This approach allows for the modeling at the subsystems level without addressing the issue of the coupling of the different subsystems involved in the model. The Functional Mock-up Interface (FMI) defines a standard that follows the co-simulation approach.

The Functional Mock-up Interface (FMI) [13, 14, 15] is a tool-independent standard for both the model exchange and co-simulation of dynamic models [16]. The concept of dynamic models exchange provides a modeling environment with the ability to generate an input/output block to represent a dynamic model which can be used later within another modeling environment. The FMI for co-simulation adopts a different approach where instead of transmitting a model to another tool, each model is simulated using its own tool and data resulting from this simulation can be transmitted to other tools.

The FMI for co-simulation consists of two distinct parts: a co-simulation interface and a co-simulation description schema. The first one, the interface, is defined through a set of `C` functions that allows for controlling the different tools. The data exchange of input and output values can be controlled through this interface. The second one describes, as an XML file, the information that characterizes a tool (input, output, solver capacities ...). Although these models can communicate between them, the data exchange is restricted to discrete communication points. Between these communication points each subsystem is solved independently.

The Functional Mock-up Interface does not rely on direct coupling in order to achieve its co-simulation environment. FMI assumes the existence of a master located between each simulator involved in the co-simulation environment. This master, which is not included in the FMI for co-simulation standard [17], has the responsibility to synchronize, to control and to manage the different tools involved in the simulation. As such, the master appears as an interface which establishes the connections and the data exchange between tools. The FMI standard assumes that the different tools, referred as slaves, only communicate with the master.

Although the master is not considered as part of the standard, there exists a *prototype master* implementation realized in the framework of the MODELISAR European research project [18]. This implementation prototype master provides three simulation algorithms with fixed step size: a data flow algorithm, a fixed point iteration algorithm and a simple implementation of Newton's method. This prototype is developed for commercial purposes.

This approach is interesting, but in term of efficiency using a single simulation engine remains a more efficient approach [17]. Indeed, efficiency and simulation speed strongly depend on the problem to be solved. While simulation of graphs without feedback can be simulated quite efficiently using the non-iterative method, the presence of cycles and the incapacity to use iterative method lead to low accuracy and low numerical stability. Furthermore, the master timestep may have to be very small which increases the computational cost and, hence, may lead to a global simulation speed quite low. In addition, the overhead induced by the synchronization between tools is often too high. Consequently, this approach does not fit our objective and is not considered in this work.

3.3 Frameworks

3.3.1 Ptolemy II

One of the pioneering works in the field of heterogeneous systems simulation was done in Ptolemy Classic [19] and its sequel, Ptolemy II [20, 21]. Ptolemy II is a proof of concept simulator, which addresses the complex issue of modeling heterogeneity in a hierarchy of connected entities. Ptolemy II introduces the notion of *Hierarchical Heterogeneity*. This approach exhibits a heterogeneous composition where the interaction and communication between models are represented while preserving the properties of each individual model. It allows for the decomposition of a complex heterogeneous system as a tree of nested sub-models, each sub-model representing a network of interacting components. Beneath the hierarchical representation, different synchronization mechanisms between models may be used at different levels of abstraction. Each sub-model, i.e. each level in the hierarchy, safeguards the intrinsic properties of each model; the interconnection between the different sub-models gives the opportunity to evaluate the system as a whole.

Ptolemy II relies mainly on an actor-oriented view to describe a heterogeneous system [22, 23]. Two kinds of actor are available:

- *Atomic Actor*: describes a basic behavior block of the system. It represents a leaf in the hierarchical tree representing the system.
- *Composite Actor*: represents a netlist of sub-actors which can either be atomic or composite actors; it is a sub-model representing a network of interacting components.

Each local sub-model is managed by a specific and well-defined Model of Computation (MoC) which actually defines how computation and model solving are performed. MoCs are implemented by means of *Domains*: *Receivers*, which encompass the communication semantics, and *Directors*, which define the execution order of actors. Together, they define the environment of actors. Directors are eventually responsible for the instantiation of domain-specific receivers. Ptolemy II includes domains such as Discrete Event (DE), Continuous Time (CT), Synchronous Data Flow (SDF), amongst others. A list of the available domains is defined in [24].

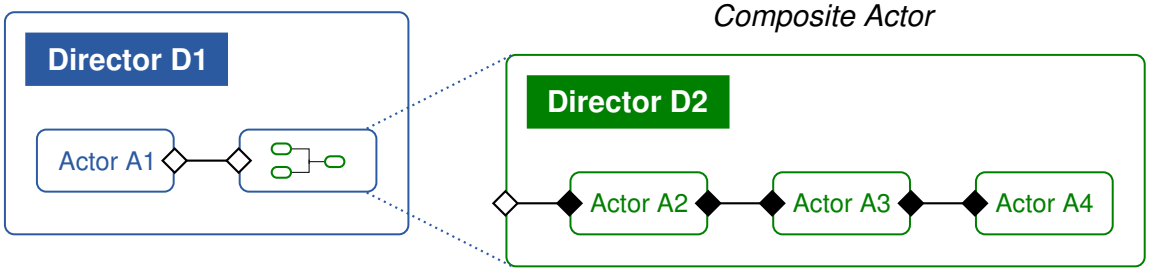

Figure 3.1: Hierarchical Modeling in Ptolemy II (adapted from [20]).

An illustration of a system designed with Ptolemy II semantics is shown in Figure 3.1. It describes the use of atomic actors and composite actors, highlighting the hierarchical representation of the system. The communication mechanisms through ports and channels are also represented; the different domains interacting are subsequently shown as directors, and receivers are depicted.

Since the whole environment is defined by the domain, actors represent abstract functionalities that are inherently reusable in many domains. The implications of this flexibility, however, can be tricky or overwhelming for the end-user. In addition to building the netlist of components the designer wishes to simulate, he also has to explicitly build the specific composite actors so as to encapsulate the subsystems and he must choose and instantiate the correct directors with respect to the created hierarchy and the simulated domains. Although actors that can be run under different domains are a good idea in order to increase the reusability of models, it is also a source of struggle. The user is forced to fully investigate the impact on the model behavior introduced by a change of directors and receivers as they implement the semantics of a different MoC. As a result of this, the models have to be completely revalidated. Another inconvenience is that the model hierarchy has to reflect the hierarchy of homogeneous domains under control of individual directors instead of following solely the natural decomposition of a system into its sub-systems. Thus, simulator-specific artefacts become intermingled with actual system components. These constraints raise two major issues. Firstly, it forces the user to fully apprehend the director's internals and the underlying simulator semantics. Secondly, the explicit definition of the hierarchy produces an intermingling of system components and simulation-specific artefacts.

3.3.2 ModHel'X

ModHel'X [25] is a framework designed to model heterogeneous systems. It relies on the association of sub-models, described using different modeling languages, to construct the whole model. Like Ptolemy II, it relies on the concept of Models of Computation and a hierarchical heterogeneity approach. To handle a new modeling language, an expert in this language must define the associated MoC with respect to the ModHel'X semantics. This must be done in order for the generic execution engine to perform a correct interpretation of the modeling language's semantics. The need for an expert to define a new Model of Computation is an obstacle to the development of this framework. Consequently, only a few MoCs are available.

3.3.3 Modelica

Modelica is an object-oriented language for hierarchical physical modeling [26]. It relies on several features including non-causal modeling and multi-domain modeling capability [27]. There are several commercial modeling and simulation environments for Modelica available, such as Dymola (Dynamic Modeling Laboratory) [28] or Math-Modelica.

Modelica adopts an approach similar to Ptolemy II with actor-like semantics. However, the

ports used to interconnect different bricks of a model are not specified as input or output, instead the connections are expressed with equation constraints on variables. While this approach has significant advantages in the definition of physical models, it appears to be harder to combine with other entities [29].

The conception of hybrid system with Modelica can prove to be tricky. Indeed, the same notions, depending on the environment they are declared in, can have different meanings. For example, events in an equation environment are simultaneous and cannot be treated sequentially, whereas in algorithm environment, simultaneous events could be lost [30].

3.3.4 Matlab

Matlab [31] is a commercial tool with its own programing language where the models and algorithms are described using mathematical notation. Although it is primary intended for numerical computing, additional optional toolboxes can be interfaced with Matlab in order to enhance the possibilities offered by the Matlab environment. It is worth noting that the Matlab language is an interpreted language which means that it is not compiled. This has an impact on performance with regards to the simulation speed.

Simulink [32] is a commercial toolbox associated with Matlab. It is a graphical programming environment for modeling, simulating and analyzing multi-domain dynamic systems. Simulink relies on libraries which provide access to a set of components from different engineering domains.

Matlab, when associated with Simulink, represents a relatively well-adopted solution for performing interactive design at system level. Designers can create, simulate and modify block diagrams. They are widely used for capturing system requirements and developing signal-processing algorithms [33]. Although they provide powerful possibilities for the description of analog and mixed-signal systems at system level, they support neither the modeling of digital hardware/software systems, nor the simulation of analog subsystems on the electrical circuit level [34]. A support for different Models of Computation is missing and hence Simulink is restricted to the evaluation of abstract models [35].

A study on languages and tools for hybrid system design [30] showed some drawbacks of Matlab. This study states that the behavior of the system is sensitive to the inner workings of the simulation engines, and, consequently, is liable to lead to erroneous results. An in-depth knowledge of the internal of the tool is important in order to prevent such behaviors.

Despite being user-friendly and providing a great set of primitives through the toolbox Simulink, Matlab's simulation performances do not fit our purpose. We are looking for a fast prototyping simulator and with Matlab, depending on the abstraction level within the model, the simulation speed may significantly drop.

3.3.5 Metropolis

Metropolis [36, 37] is a system design environment for heterogeneous embedded systems, which favors reusability of components in the systems through the decoupling of orthogonal aspects [38]. The decoupling of orthogonal aspect addresses three main characteristics:

- *Computation and Communication*: it represents an important separation since the computation and the communication do not follow the same refinement process.
- *Functionality and architecture*: the separation is suggested since these two aspects are often defined independently.
- *Behavior and Performance indices*: the separation between these two characteristics is motivated by the fact that performance indices as constraints are often specified independently from the behavior. And as results, they derive from a specific architectural mapping of a behavior.

To this aim, the Metropolis framework mainly relies on an internal representation called *Metropolis Meta Model (MMM)* that defines a set of abstract classes. This meta-model can be described following three aspects:

- *actions*: they can be defined in terms of computation (process), communication (medium) and coordination (scheduler or linear temporal logic).
- *constraints*: they rely on the definition of a quantity object associated with actions that can represent time or power. Constraints can then be specified using the form of predicate logic.
- *refinement*: through inheritance principle, one can define and model a well-separated computation and communication semantics. This mechanism allows for the definition of a more detailed behavior of the system.

Although Metropolis provides efficient features such as the separation between computation and communication, and its refinement mechanism, the process-based approach within this framework leads to a non-hierarchical modeling approach. Indeed, all processes should be implemented in the same hierarchical level to be interconnected with mediums. We believe a hierarchical approach (as within Ptolemy II) is best suited for heterogeneous modeling. The expression of the synchronization can be a little tricky since it is left up to the model designer. He should express the time synchronization through the definition of constraints upon quantity handled by a quantity manager. Moreover, the communication medium performs the data synchronization, and hence it may be used as a converter channel. Finally, the meta-model does not provide a predefined notion of time, which means that the definition of a global notion of time is left up to the model designer at the language level through a quantity object.

3.4 SystemC-based Frameworks

Besides Ptolemy, several solutions for the simulation of heterogeneous systems have been presented over the course of the last decade. They mainly rely on SystemC [39], a discrete event simulation kernel, which can be used to perform rapid system prototyping at several levels of abstraction.

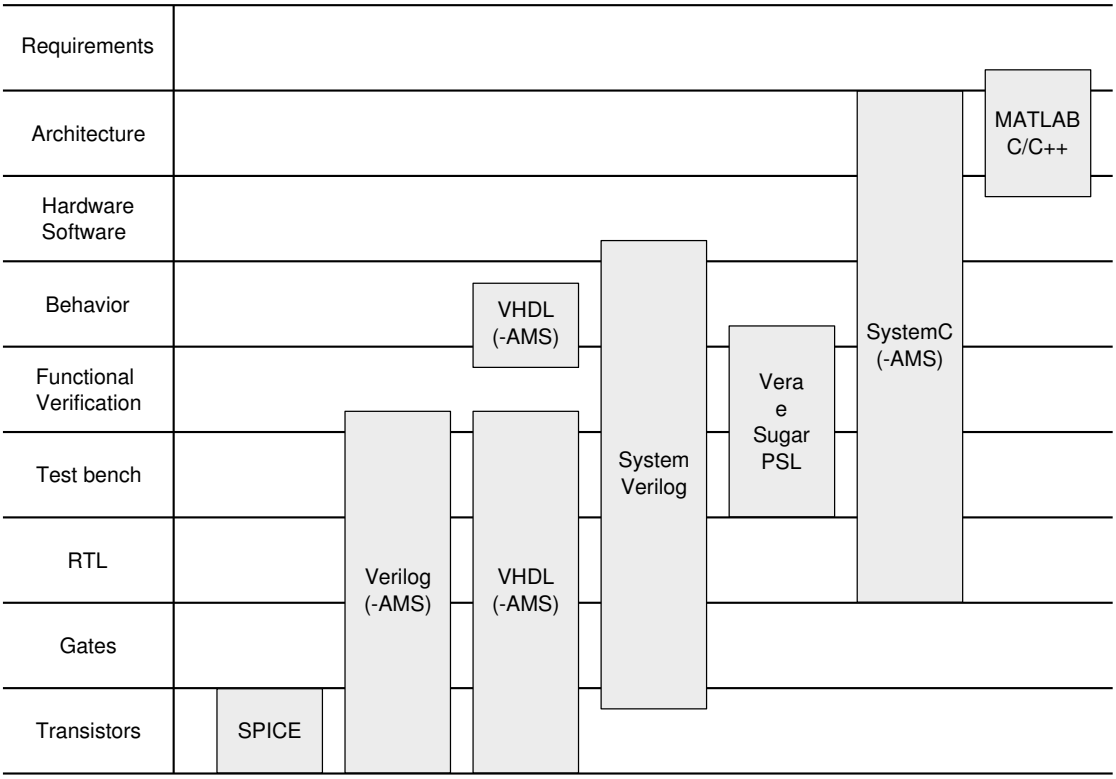

Figure 3.2: Design languages and their main purpose, adapted from [33]

The levels of abstraction covered by SystemC are depicted in Figure 3.2. SystemC goes from the Register Transfer Level (RTL) to the architecture level through several other levels (test bench, functional verification, behavior and hardware/software). Furthermore, this figure illustrates the levels of abstraction cover by other frameworks, notably Matlab.

SystemC [33] is a system design language that allows for the modeling and co-development of hardware and software at a high level of abstraction. It is also a hardware description language (HDL), thanks to its several levels of abstraction which allow it to precisely describe hardware architectures. SystemC comes as a C++ class library and, hence, leverages the powerful and efficient capacities of the C++ language.

The wide range of abstraction levels offered by SystemC allows a designer to describe a system with several levels of accuracy. A designer can describe a subsystem at a lower level of abstraction while the rest of the system remains at a high level of abstraction. For example, the system can be described using the Transaction Level Modeling (TLM) [40] at a high abstraction level and a subsystem can be described using a Cycle Accurate Bit Accurate (CABA) abstraction level. This approach offers the designer a good understanding of the system he is modeling.

The Discrete Event (DE) simulation kernel of SystemC relies on two main tasks - the *Elaboration* and the *Simulation* phases. The SystemC kernel phases are described in Figure 3.3.

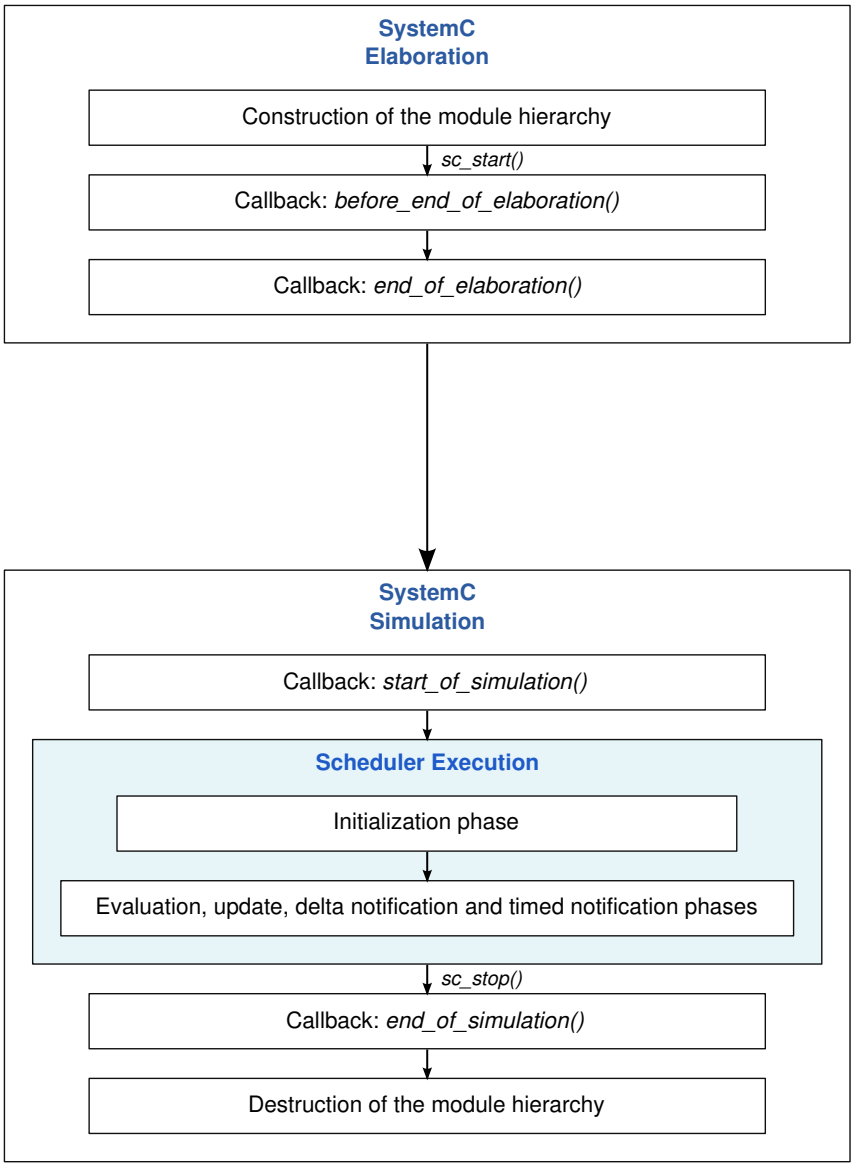

Figure 3.3: SystemC Kernel Phases, adapted from [11].

The Elaboration phase aims to prepare the kernel and building efficient data structures for the simulation. During this phase all the data structures required to support the simulation are created. To this aim, the system modeled is explored.

The objective of the Simulation phase is to execute the model described. To do so, the kernel relies on a scheduler that handles a list of processes to execute. The processes are divided into different lists depending on their status. A process can be runnable, meaning it waits until the scheduler triggers its execution, or a process can be queued in a *pending* list waiting on a notification in order to be runnable.

The kernel catches all the notifications - events that control the execution of the processes. When it receives a notification, the scheduler can move a process from the pending list to the

runnable one if this process is sensitive to this notification. The scheduler repeats this scheme until no more notifications are received, no more processes are runnable, or the simulation is stopped.

Figure 3.3 highlights some relevant callbacks provided by SystemC that are automatically called on each primitive by the SystemC kernel. These callbacks can be overridden in order to execute specific behaviors at different stages of the simulation. Hence, these callbacks constitute entry points into the SystemC simulation kernel that do not alter the kernel itself.

- `before_end_of_elaboration()` is called before the kernel reaches the end of the elaboration phase. During this callback, modifications to the system are still authorized. This callback can be used to program actions to execute during the elaboration phase that may alter the structure of the system modeled
- `end_of_elaboration()` is called when the kernel reaches the end of the elaboration phase. At this moment, the data structures required by SystemC to support the simulation have been created and, consequently, the system can no longer be modified. This callback can be used to program actions, executed during the elaboration phase, that do not alter the structure of the system modeled.
- `start_of_simulation()` is called in the beginning of the simulation. This callback can be used in order to program actions to execute at the beginning of the simulation.
- `end_of_simulation()` is called at the end of the simulation. This callback can be used to program actions to execute at the end of the simulation.

The solutions for the simulation of heterogeneous systems based on SystemC rely on two different approaches. Firstly, we find the solutions which extend and modify the kernel SystemC. Secondly, we find the solutions which extend the functionalities of SystemC without altering its kernel. Solutions following both approaches are introduced in the following.

3.4.1 SystemC-A

SystemC-A [41, 42] is an extended version of SystemC which provides analog, mixed-signal and mixed-domain modeling capabilities. SystemC A enables support for user-defined ordinary differential and algebraic equations which enable the modeling of analog systems. It defines an analog kernel, which provides both linear and nonlinear solvers to solve the analog systems. This kernel is integrated within SystemC through a primitive module but there is a major drawback with the solution proposed - it requires the modification of the SystemC kernel. To make the integration of the analog kernel with the discrete event kernel possible, and to perform the synchronization between them, the authors modified the SystemC kernel which limits portability and standard compliance.

3.4.2 SystemC-H

SystemC-H [43, 44] is an extension of SystemC which enhances the discrete event kernel capacities in order to support heterogeneity. Within this framework, the authors want to provide support for several models of computation. The adopted approach is to provide a simulation kernel dedicated to each MoC. The authors addressed the simulation of heterogeneous systems through the definition of a heterogeneous simulation kernel, each part of the system being simulated by the kernel which fits the MoC used to describe it. They designed a simulation kernel wherein the MoC dedicated kernels are interoperable with the discrete event kernel and thus defined an alternate SystemC kernel. As SystemC-A, they modified the internal structure of the kernel SystemC which limits portability and standard compliance.

3.4.3 HetSC

HetSC [45, 46] is a framework which describes a heterogeneous specification methodology built on top of SystemC kernel. One important thing to notice is that HetSC provides heterogeneous support without modifying SystemC kernel. In the context of this framework, heterogeneity is defined as the ability to specify a set of communicating subsystems described under different Models of Computation (MoCs). This methodology supports untimed MoC and synchronous MoC. We can specify Process Network, Kahn Process Network, Communicating Sequential Processes and Synchronous Data Flow as untimed MoC and distinguish the Synchronous Reactive MoC as synchronous Models of Computation. This framework enables new MoCs to be integrated as long as they can cooperate and be abstracted by the Discrete Event (DE) SystemC simulation kernel. We can say that HetSC is mainly a communication library and that it does not express heterogeneity as we defined it. Within this framework, the Models of Computation (MoCs), which express the heterogeneity, only describe abstracted DE models and cannot express the surrounding physical environment. We should note, for example, that HetSC does not support continuous time.

3.4.4 SystemC AMS

SystemC AMS [47, 48, 49] is a specification methodology developed on top of SystemC by the Accelera Systems Initiative organization[1]. SystemC AMS extensions [50] have been specifically developed in order to improve the modeling capacities of SystemC by allowing the simulation of analog behaviors coupled with digital-centric systems. SystemC AMS comes as a `C++` library that follows the same approach of SystemC. It relies on the same definition of objects to realize the design of a system: modules, ports, interfaces and channels. SystemC AMS is defined following a layered architecture approach [51] described in Figure 3.4.

[1]The development is carried out by the AMSWG: SystemC AMS Working Group

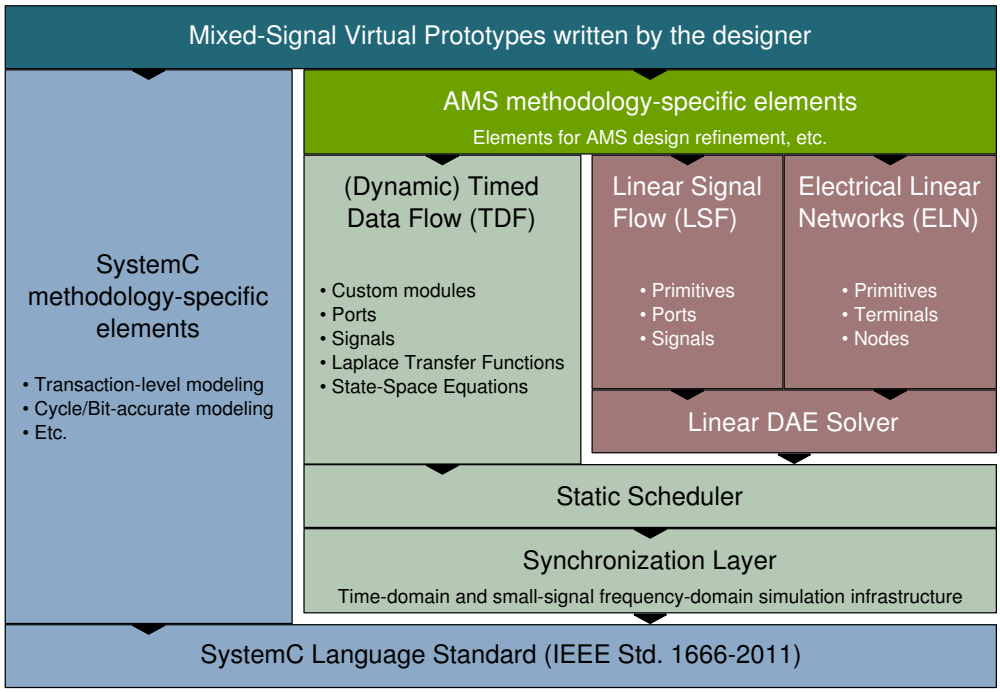

Figure 3.4: SystemC AMS language Standard Architecture, adapted from [52].

SystemC AMS integrates three layers to the existing set of layers defined in the SystemC environment [48]:

- *The view layer*: it defines the different descriptive methods provided to the designer in order to write executable models.
- *The solver layer*: it contains the implementation of different solvers required in order to model specific AMS behaviors.
- *The synchronization layer*: it defines a mechanism in order to organize the simulation of a SystemC AMS model that may include several views.

SystemC AMS also supports the notion of MoCs and several abstraction of time: Discrete Time (DT), Continuous Time (CT). Figure 3.4 shows the three MoCs defined within SystemC AMS: Timed Data Flow (TDF), Linear Signal Flow (LSF) and Electrical Linear Network (ELN).

The TDF MoC allows discrete time modeling, and efficient simulation of signal processing algorithms and communication systems at the functional and architectural level. A dynamic approach to the TDF MoC is defined as Dynamic TDF (DTDF) [53]; it allows modifying some of the simulation parameters during the simulation. One should note the strong implication of the TDF MoC in the definition of SystemC AMS. Indeed, TDF is more than a simple Model of Computation within SystemC AMS; Figure 3.4 highlights the fact that TDF constitutes the only synchronization mechanism available within SystemC AMS. Although other MoCs are defined, they all have to go through the TDF semantics in order to be executed.

The LSF MoC supports the modeling of continuous time behavior through the definition of predefined primitives (such as addition, multiplication, integration, etc.). LSF allows the modeling of non-conservative systems, the connection of several primitives defines a system of linear equations solved by a linear DAE solver. The ELN MoC enables the modeling of electrical networks through the definition of predefined primitives (such as capacitors, resistors, etc.). The connection of several primitives describes the continuous time relation between voltage and current. However, ELN is restricted to the modeling of linear electrical systems.

These extensions were originally implemented in the Fraunhofer SystemC-AMS proof-of-concept simulator [54]. They have been successfully applied in communication [55], automotive [56] and consumer electronics use cases with good simulation performance and accuracy. The commercial software COSIDE [57] is based on this proof-of-concept simulator.

For the moment, though, it is rather difficult for design teams to extend the current SystemC-AMS simulator with other MoCs than those proposed [55]. The SystemC AMS 2.0 standard [50] does not define an Application Programming Interface (API) for this purpose, nor does the proof-of-concept simulator document its internal API. To our knowledge, only two attempts have been published, the first adding an Non-Linear Network (NLN) [58] and the second, a Bond Graph (BG) MoC [59], respectively, by authors with an in-deep knowledge of the SystemC-AMS implementation. These MoCs rely on internal APIs to integrate themselves without modifying SystemC AMS. An analysis of the SystemC-AMS source code shows that each MoC is required to fully handle its elaboration once the SystemC port binding phase has been finished. SystemC-AMS provides only a minimal support for this task by providing a list of all instantiated modules belonging to a certain MoC. It provides one synchronization mechanism used by all the existing MoCs and no API is provided to define new synchronization schemes between MoCs.

All in all, the modeling of heterogeneous systems can be considerably error prone from a physical perspective, since designers from different disciplines use different measurement units and scales. A major improvement towards heterogeneous simulation was made with the integration of dimensional analysis into SystemC AMS [60] through the use of `Boost::Units` library [61]. It allows designers to enhance models with the notion of physical quantities, which avoids compositional errors.

3.5 Multi Disciplinary Monitoring Mechanism

Monitoring is a mechanism which aims to observe and record information about a system. It may be used, for example, to detect threshold crossing, to perform profiling evaluation on a system or as a tracing mechanism. Tracing mechanisms are mainly used for debugging purposes, through the log of information during the simulation.

Monitoring, threshold detection, profiling and tracing functionalities, represents an important feature in a simulator framework. We think that these functionalities should be developed

simultaneously. Performing the detection of threshold crossing, the profiling or the tracing involves the same mechanisms, only the outcome is different. All of these capabilities require a mechanism to probe the system in order to gather relevant, specific information.

In the following sub-sections we will introduce existing solutions that may fit the requirement to provide an efficient monitoring mechanism.

3.5.1 Aspect-Oriented Programming (AOP)

Object-Oriented Programming (OOP) [62] allows us to break up a problem into a set of reusable objects. Although these objects mainly describe a single functionality, they usually share common, secondary behaviors with other objects. These common behaviors are usually scattered throughout the whole system, as shown on Figure 3.5, breaking the encapsulation principle. You can see the same behavior replicated in different components (objects).

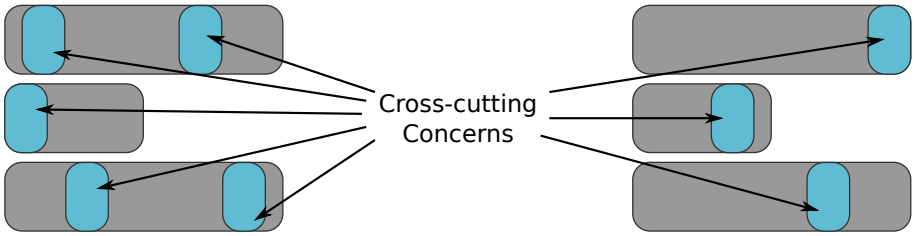

Figure 3.5: Classical Object Oriented Programing.

Think of the tracing of the functions called during the simulation. You would have to insert print function within every function of your system (potentially the exact same code line)! These common behaviors are identified as cross-cutting concerns. Cross-cutting concerns denote behavior that cuts across the boundaries of assigned responsibility for a given modular element. They are often shared, and common. They may describe process synchronization, location control, timing constraints, persistence, failure recovery, tracing, monitoring, verification, etc.

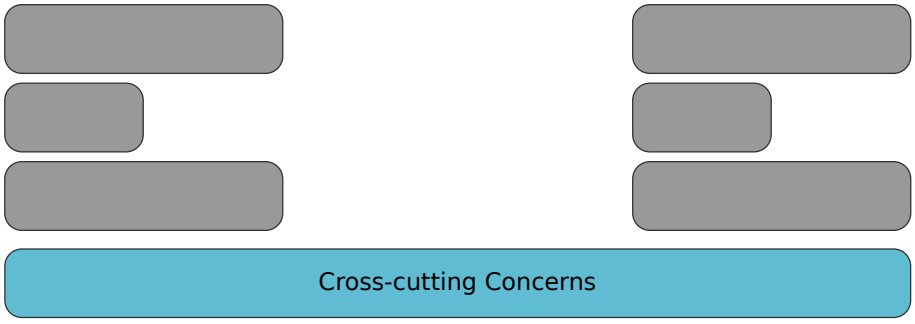

Figure 3.6: Aspect Oriented Programing.

The Aspect-Oriented Programming (AOP) [63, 64, 65] concept was introduced in order to address the issues linked to these cross-cutting concerns. AOP allows defining these common behaviors in a single module, resulting in a better code structuring and increased reusability and code maintenance. It prevents the intermingling of functional code with non-functional code.

The AOP representation of the system is described in Figure 3.6. You can see that the common behaviors previously scattered through the system are now concentrated within a single entity.

Aspect-Oriented Programming relies on a few concepts which, in order to understand how this paradigm works, are described below.

- *Advice*: contains the additional code that you want to apply on your existing model in order to describe a cross-cutting concern.
- *Join Point*: represents a point in the control flow of a program. It defines all the points in execution where it is possible to interact. It may refer to a method, an attribute, a type (class, union or struct), an object, etc.
- *Pointcut*: represents a set of Join Point at which a cross-cutting concern needs to be applied.
- *Aspect*: represents the combination of a Pointcut and an Advice.
- *Weaving*: realizes the insertion of different aspects into the existing software thanks to a weaver. It can be done statically during the compilation or dynamically during the execution.
- *Weaver*: Source-to-Source tool, which produces a new source code, when given an original source code and a set of Aspect codes.

Figure 3.7 illustrates the mechanism of the Aspect-Oriented Programming. The original source code along with the different aspects are given to the weaver, which produces a new source code including the aspects' functionalities. Following that, this new source code is compiled normally using a compile chain (e.g. g++ when coding in C++) to produce the executable file.

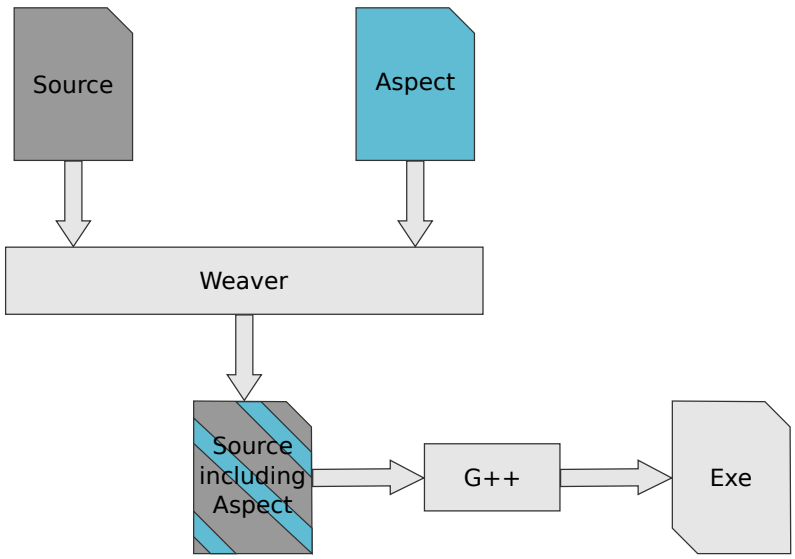

Figure 3.7: Aspect Oriented Programing Mechanism.

Aspect-Oriented Programming is of great interest to us. Not only does it allow us to define common behavior (tracing, monitoring) in a single module, providing a better code structuring and improved reusability and code maintenance, but also it increases the modularity as the additional code brought by the *aspects* can be easily added or removed within the system. The

original source code remains unchanged and independent from the aspect code since it is not aware of the aspect functionalities. Thus it may provide monitoring functionality to existing models with no alteration of these models.

Unfortunately, the *C++* implementation of the Aspect-Oriented Programming, AspectC++ [66, 67], does not support the object declared using template functionality of *C++*. This represents a major drawback since models described with SystemC or SystemC MDVP may use them.

3.5.2 LLVM - Clang

For the purpose of achieving the same goal as witnessed with the AOP we can consider the possibility of developing our own tool to perform source code transformation with a view to enhance the original source code by adding monitoring functionality. LLVM [68, 69] and its frontend Clang [70] represent a powerful tool with the potential to achieve this objective. Clang exhibits a rich API which handles and manipulates the source code through the Abstract Syntax Tree (AST). We can imagine a complete compilation-tool-chain base on Clang to perform the monitoring. We need to automatically identify key points in the design flow where we can insert monitoring source code, relying on specification provided by the end-user. Although Clang represents an interesting and flexible solution that clearly fits our objectives in terms of monitoring, the development of such a tool would require a considerable amount of time and, therefore, cannot be considered in the frame of this thesis.

3.6 Conclusion

For the best of our knowledge, the SoC community still needs a complete design environment that can handle both the digital, and the analog parts, at a higher level of abstraction. With the exception of Ptolemy II, the existing solutions do not clearly express the semantics information needed when it comes to heterogeneous simulation. We take full advantage of the previous work of Ptolemy II, conserving their hierarchical heterogeneity approach, focus on Models of Computation and associated semantics. Conversely, we want to avoid the intermingling of simulation artefacts and models. Accordingly, our approach neither includes domain-polymorphic actors, nor explicit directors or receivers.

The work of SystemC AMS has also inspired us to extend SystemC in a bid to perform analog simulation. The principles and the TDF MoC developed within this framework are of significant interest to us. That being said, we want to steer clear of the strong dependencies on TDF in the existing SystemC AMS as well as providing the possibility of extending the current set of available MoCs. SystemC MDVP can benefit from Boost.Units Library which authorizes the description of quantity data types, the goal being to express physical data/values; compile time checking is possible. However, we aspire to correct the verbose and unreadable error message provided by this

library. A representation of the influence of other frameworks on SystemC MDVP is provided in Figure 3.8. The ensuing chapters of this document are dedicated to the description of our framework - SystemC MDVP.

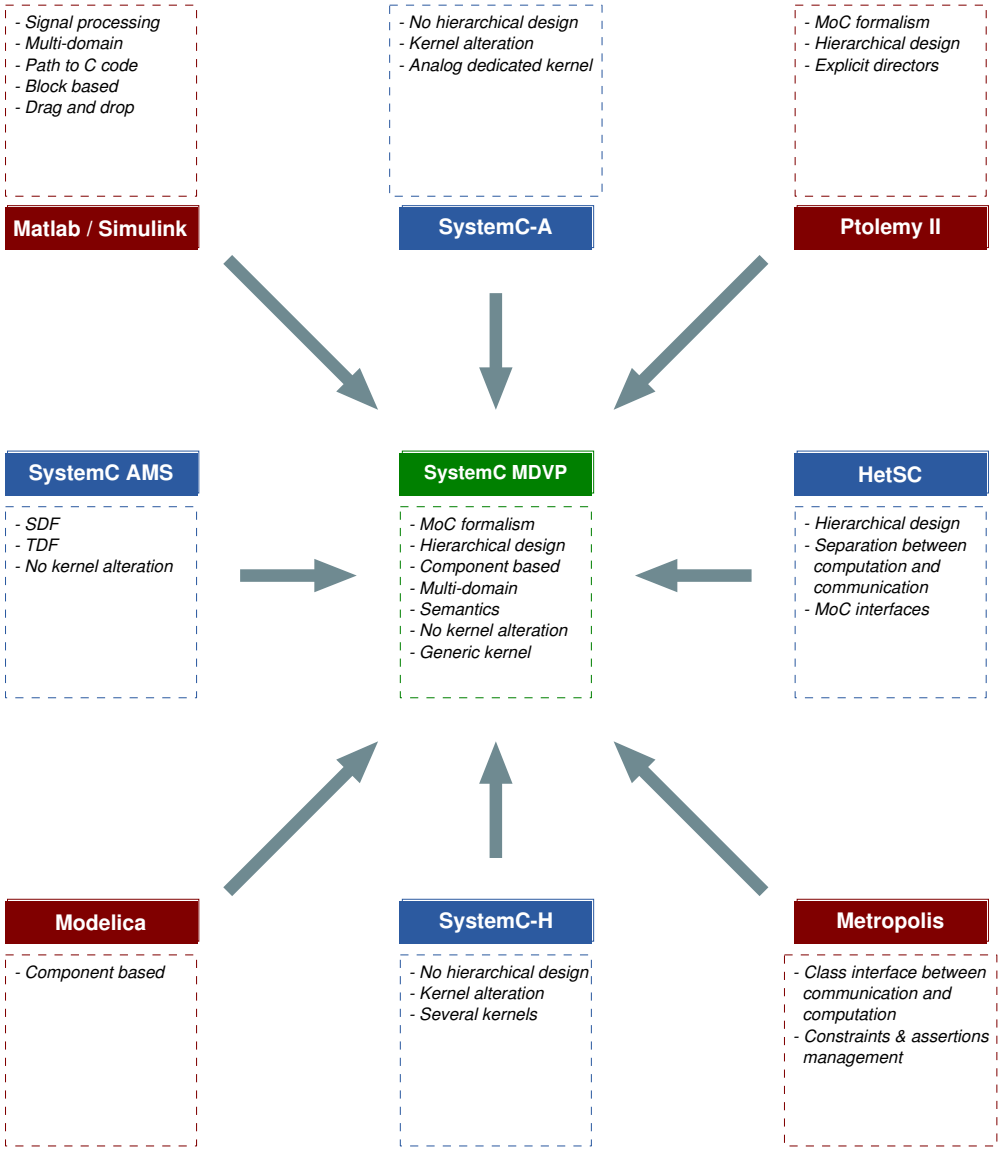

Figure 3.8: Influence of other Frameworks on SystemC MDVP.

The following of this document is dedicated to the description of our framework SystemC MDVP.

Chapter

4 *SystemC MDVP: Principles*

Contents

4.1 Introduction 32
4.2 Models of Computation 33
4.2.1 Discrete Event (DE) 34
4.2.2 Timed Data Flow (TDF) 35
4.2.3 Bond Graph (BG) 36
4.2.4 Electrical Network (EN) 36
4.2.5 Abstract representation of a MoC 37
4.3 Interaction Mechanism 38
4.4 MDVP Hierarchical Approach 43
4.5 User Profiles 45
4.5.1 Simulator Architect 45
4.5.2 MoC Architect 46
4.5.3 SoC Architect 48
4.6 Conclusion 48

4.1 Introduction

Based on previous works in the field of heterogeneous systems, and with an aim to leverage what has already been done, this chapter introduces the underlying principles of SystemC Multi Disciplinary Virtual Prototyping (MDVP), a framework for the simulation of heterogeneous systems.

We believe that digital hardware and software are at the heart of current and future heterogeneous systems. Driven by the desire to enable architectural exploration, early software development, and following a digital centric approach, our framework targets the simulation of high abstraction system-level models, where extreme accuracy is not the most important feature. Therefore, SystemC MDVP is designed as an extension of SystemC which permits to naturally achieve heterogeneous digital centric simulation since SystemC already allows hardware and software co-simulation.

Our objectives with SystemC MDVP are to provide a fast simulator to enable the architectural exploration of heterogeneous systems in the early stages of the development process. We want to allow the simulation of a heterogeneous system as a whole, including both the digital and analog (physical) parts. With a correct-by-construction approach, one of our objectives is to perform the validation of the global system architecture from a functional viewpoint.

Furthermore, one important point is to provide the opportunity to develop the embedded software, associated with the system, in the early stages of the design process. Thanks to virtual prototyping, we are able to develop the software associated with the platform prior to the availability of a physical prototype and, consequently, validate it as soon as possible.

Throughout the course of this chapter, a running example will be used to illustrate the principles of the simulator. This example is shown in Figure 4.1. One can see that the targeted System on Chip embeds a digital component (micro-controller) as well as components from other engineering domains (sensors) (Figure 4.1, ①). These sensors could represent a multitude of engineering domains, such as MEMS, optical, biological, thermal, etc.

The example presented represents a 3-axis vibration sensor (Figure 4.1, ②). Each sensor detects the vibration along a specific axis. The representation of this system will evolve over the course of this chapter in order to illustrate the presented concepts of the simulator. This figure shows our vision of embedded systems which positions the digital part at the center of the system with other domains gravitating around it.

This chapter is organized as follows.

Section 4.2 gives an overview of several Models of Computation allowing us to define the notion of heterogeneity within SystemC MDVP. Characteristics of these MoCs are depicted and an abstract representation of a MoC is introduced.

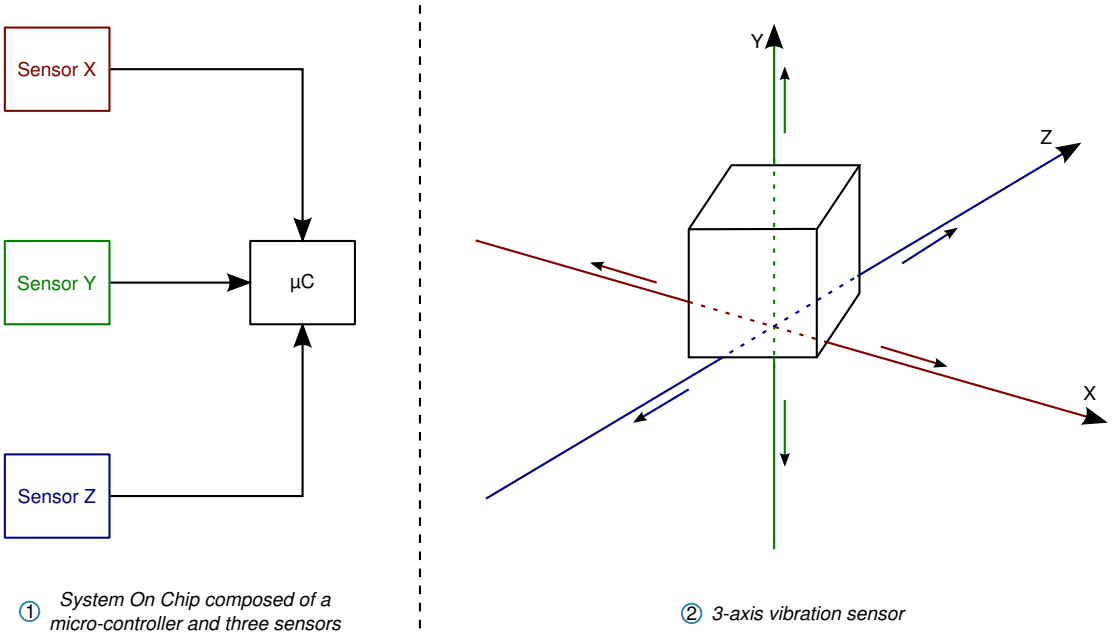

Figure 4.1: Running Example: System on Chip representing a 3-axis vibration sensor.

Section 4.3 describes the interaction mechanism that prevails in our framework. This mechanism is defined by means of master-slave relationships between the MoCs. The semantics associated with this approach are introduced and the resulting opportunities and constraints are discussed.

This interaction mechanism suggests a hierarchical approach to the heterogeneity that is presented in Section 4.4. We describe what it represents and how the architecture of the simulator is impacted by such an approach.

Following the reflection during the design of this framework, we identified the different kinds of users of said framework; these users, as well as their role and position in the design process are discussed in Section 4.5 We illustrate how this approach based on multiple user profiles impacted and modeled the requirements to be met and the principles on which the simulator is built.

Finally, Section 4.6 provides an overview of the principles underlying the SystemC MDVP framework and concludes this chapter.

4.2 Models of Computation

As stated in Chapter 2, a clear definition of heterogeneity must first be established in order to properly perform the simulation of heterogeneous systems.

Within the SystemC MDVP framework, heterogeneity is defined as the association of different Models of Computation (MoCs), where each MoC can represent a specific entity with its own semantics. A MoC constitutes a heterogeneous entity.

A MoC defines how the computation and the communication take place within an assembly of interacting components; it covers the data flow as well as the control flow which allows the MoC to give semantics to this infrastructure [20]. A MoC also defines all the information required to carry out the modeling of a system using its own semantics. As such, in addition to defining how the computation and the communication are handled, a MoC encapsulates the components used to design and model a system (elementary primitive, communication channel and port). Therefore, contrary to Ptolemy's viewpoint, we associate each modeling primitive to a specific MoC, giving it fixed semantics including the used abstraction of time (DE, DT or CT). Finally, a MoC contains a solver which is in charge of the resolution of the physical system described by the MoC using its own abstraction of time. A MoC must provide the functionalities required by the designer to allow him to undertake the design of a system. The association of several MoCs allows for the modeling of different physical entities that belong to separate disciplines.

To illustrate the notion of Model of Computation and support our approach, we now present several examples of MoCs: Discrete Event (DE), Timed Data Flow (TDF), Bond Graph (BG) and Electrical Network (EN). For each MoC we present its principles, its internal mechanisms, its solving algorithm, and how the designer uses it.

4.2.1 Discrete Event (DE)

Discrete Event (DE) Model of Computation is based on a discrete representation of time, where a model is described as a sequence of events occurring in time. Each event occurs at a specific time and has the potential to modify the state of the modeled system. Between two events, the system is stable and no changes in the system can occur, thus the system clock jumps from one discrete timestamp to another following the event rate. For a designer, the use of a DE MoC simply consists in describing the behavior associated with a model and defining when it should be run, i.e. specifying the events which trigger its execution. Several approaches exist to describe a model using the DE MoC, we can use a *Cycle Accurate Bit Accurate* (CABA) approach or a *Transaction Level Modeling* (TLM) approach. While each approach has its advantages and disadvantages, for the sake of simplicity, we will only take the CABA approach into consideration in the following descriptions. Following this approach, the designer also has to handle the communication between DE basic blocks. With this in mind, he uses input and output ports and signals to interconnect the primitive modules all together.

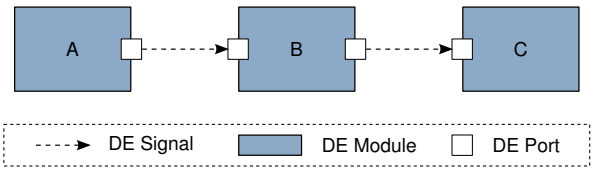

Figure 4.2: Simple model described with DE MoC

Figure 4.2 illustrates an example of a model described with the DE MoC. This example is composed of three modules (A, B and C), four ports and two signals. The behavior associated with

these modules is specified by the designer. Module A writes data to its port, and then the data is carried to Module B by means of a signal. The communication between B and C is achieved in a similar manner.

Within the Discrete Event Model of Computation, the solver is rather a scheduler, its role being to schedule the execution of the processes associated with each module (e.g. in the model described in Figure 4.2: A, B then C). It handles the events and, based on the sensitivity list of each process, chooses which process should be resumed for execution when an event occurs. If no more processes can be run at a specific time and delta, the scheduler advances the time to the next event and goes through the same process again until no more events are generated or available, marking the end of the simulation.

4.2.2 Timed Data Flow (TDF)

Timed Data Flow (TDF) Model of Computation is based on the timeless SDF theory. It uses a discrete time representation and introduces time-stamped sampled data. As with DE Model of Computation, the designer describes the behavior associated with a TDF module and also handles the communication between them through ports and signals.

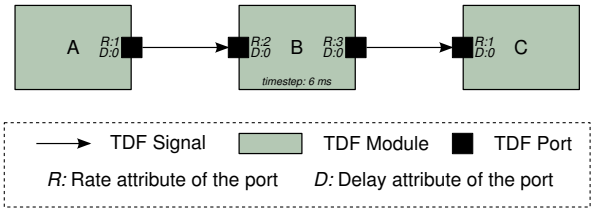

Figure 4.3: Simple model described with TDF MoC

Figure 4.3 illustrates an example of a model described with the TDF MoC. This example is composed of three modules (A, B and C), four ports and two signals connected together as a TDF cluster. Again, the behavior associated with these modules is specified by the designer. Module A writes data to its port then the data is transported to the module B through a signal. As before, the communication between B and C is achieved in a similar fashion.

Within the TDF MoC you have to set out specific attributes in order to correctly represent the system. TDF requires that you define a *timestep* value which represents a time period; the meaning can vary depending on the object to which this attribute is applied. Applied to a module it represents the time period in which the `processing()` function associated to this module should be executed. Applied to a port, it represents the time period in which the samples are read or written by this port. One have to specify a *rate* attribute associated with a port. If applied to an output port, the rate attribute defines the number of sample written by the port in question and when applied to an input port, the number of sample read by this port. One also may specify a *delay* attribute associated with a port (zero by default) which defines the initial number of samples available in a port when the simulation starts.

The Timed Data Flow Model of Computation's solver is, like DE's solver, rather a scheduler. The scheduler of the TDF MoC is able to statically determine the execution order for each of its model components, as well as how many times they need to be executed during each cluster period based on the rate and delay attributes provided by the designer (e.g. in the model described in Figure 4.3: AA, B, CCC).

4.2.3 Bond Graph (BG)

Bond Graph Model of Computation is used to describe an energy-based, physical, dynamic system through a graphic representation. It is particularly well-suited to describe multi-domain systems because it allows for the description of several energy domains (such as mechanical, hydraulic, etc...). Each domain can be represented by two variables - the effort and the flow which, combined, correspond to power. In contrast to DE or TDF, it does not allow for the customization of basic blocks, you can only use pre-defined modules. Thus, for a designer, using BG MoC can be resumed by the assembly of primitive blocks.

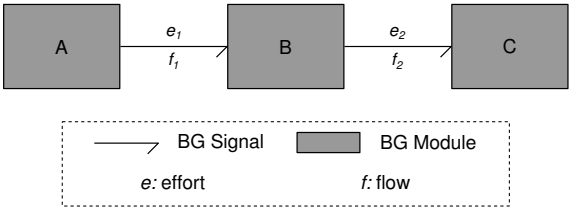

Figure 4.4: Simple model described with BG MoC

Figure 4.4 illustrates an example of a model described with the BG MoC. This example is composed of three modules (A, B and C) and two signals. The behavior associated with these modules is pre-defined. Module A expresses the data to transmit through the combination of a value associated with the effort (e_1) and another associated with the flow (f_1). The communication between B and C is achieved in the same way.

The whole point of using BG is to determine in what order the results should be propagated from one primitive block to another (especially when there are loops of primitives).

4.2.4 Electrical Network (EN)

Electrical Network Model of Computation is used to describe linear and nonlinear electrical systems based on a continuous representation of time. This MoC provides built-in basic blocks, i.e. capacitor, resistor, diode, etc... As with the BG MoC, pre-defined modules must be used as the customization of basic blocks is not supported. Thus, for the designer, the behavior of the associated model is defined by the association of several primitive blocks. With regards to communication, in the same manner as with the other Models of Computation, the designer must manipulate the ports and signals required to connect the primitive modules. Ports are referred to

as terminals and signals as nodes.

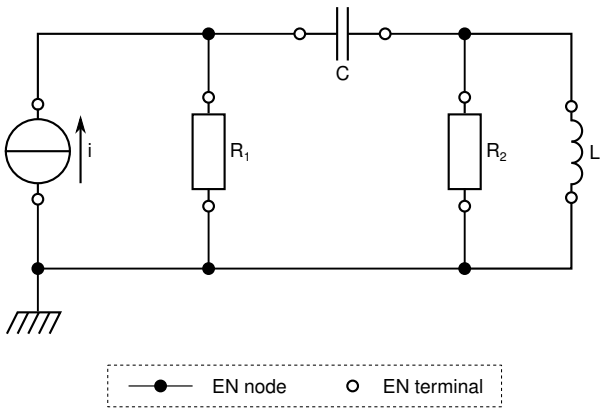

Figure 4.5: Simple electrical network described with EN MoC

Figure 4.5 illustrates an example of a model described with the EN MoC. This example is composed of several built in modules (generator, capacitor, resistor and coil), terminals and nodes.

The Generalized Kirchhoff's Laws (GKL) prevail when modeling electrical systems; they consist of two laws regarding the current (Kirchhoff's Current Law (KCL)) and the voltage (Kirchhoff's Voltage Law(KVL)). KCL states that for any node in the system, the sum of the current flowing into a node is equal to the sum of the current flowing out of this node. KVL states that the sum of the electrical potential differences around any closed network is zero.

The Electrical Network MoC's solver resolves the system of nonlinear equations determined by the global assembly of each primitive block to provide a continuous solution of the system. The solver takes into account the Generalized Kirchhoff's Laws which means that, contrary to the aforementioned MoCs, each primitive block added in the model has an impact on, and may also be impacted by, the electrical system modeled.

4.2.5 Abstract representation of a MoC

Through the presentation of these Models of Computation we are able to draw conclusion concerning different aspects of the MoCs. The way to use a MoC can be simply reduced to interconnect different entities; this approach prevails regardless of the MoC used.

As we have seen, Models of Computation can clearly be different; they may describe different domains and express different ways to specify a behavior. We notice, however, that they share some generic characteristics which our framework is based upon:

- ***Time Representation***: the abstraction of time used by the MoC (continuous time (EN), discrete time (DE), sampled time (TDF), etc...).
- ***Primitive Behavior***: the basic blocks which describe an elementary behavior, or the way to associate a behavior with a basic block when allowed to do so.
- ***Channel Representation***: the communication mechanism used to exchange data between basic blocks.
- ***Composition***: the way to compose a bigger model through ports and sub-model instantiation.
- ***Solving Algorithm***: the algorithm used to resolve the model (solver, scheduler, etc...).
- ***Interaction***: the way to communicate with another Model of Computation.

These six notions provide a functional abstraction of a Model of Computation and can be applied to every MoC. The last point, *Interaction*, represents a key aspect of heterogeneous systems; hence it is discussed in detail hereafter.

4.3 Interaction Mechanism

The presented MoC abstraction allows us to represent every MoC; however, we still need to define an interaction mechanism between them. Indeed, when it comes to heterogeneous modeling, the interaction between different MoCs becomes more challenging. The question remains, how do different Models of Computation interact with each other?

Though aware of the restrictions which accompany such an approach, we stood by our assertion that, in order to make the SystemC MDVP flexible and easily extendible, we should only consider MoC interactions by means of simplified master-slave semantics. In this relationship, one MoC commands and the other obeys. This approach offers a simple definition of the interaction between MoCs, a master MoC imposes its viewpoint upon a slave MoC.

If we look at this from a high abstraction level, we can say that a master imposes its *environment* upon a slave. First and foremost, though, we need to define what the environment represents. The environment contains all the elements needed to guarantee the functional aspects of a MoC; these elements prevail at all points in time and ensure the proper functioning of this MoC. Each MoC can have its own abstraction of time so this must be present in the environment. While these characteristics constitute generic environment components common to all MoCs, there are also master-specific properties, behaviors and characterizations that could be imposed. Additionally, you may come across some MoC-specific simulation context such as input stimuli, floor time stamp, temporal horizon, etc. The characteristics of this environment are unified within a MoC interface that will express the expectations and the requirements desired by a MoC. From

a programming viewpoint, this means that a slave MoC has to implement the complete set of properties/callbacks defined by a master MoC programming interface.

In a bid to perform a seamless interaction between different Models of Computation, the SystemC MDVP framework requires that the slave MoC provide all the interaction mechanisms needed to communicate with the master MoC, i.e. the slave must adapt itself to comply with its master semantics. No matter how complex the sub-models' hierarchy, it must appear as a single model from the master viewpoint, as shown in Figure 4.6.

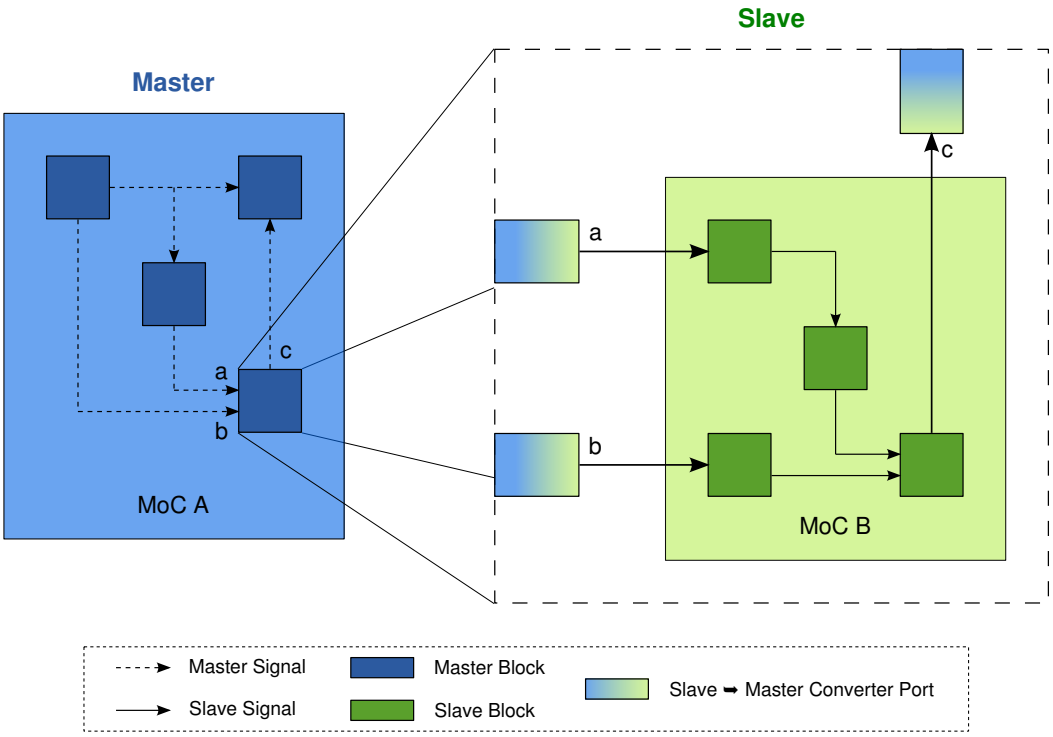

Figure 4.6: Interaction between two MoCs

Interfacing two MoCs also means translating signal values from one MoC to another. For the sake of simplicity, the translation of data values implies the use of converter ports, which offer compatibility bridges between MoCs. For the designer, the modeling process must appear as if one part of the port is in one MoC and the second is in the other MoC. One side is communicating with the Model of Computation it belongs to (the slave MoC since he is the one responsible for providing interaction mechanisms). The other side is communicating with the master MoC, with respect to its semantics, i.e., with regard to its communication interface. As illustrated in Figure 4.6 the translation of data is handled exclusively by the slave MoC while the master remains completely unaware of it. According to the slave viewpoint, a converter port operates as a regular port - data addressed to it respect the slave semantics; from the master perspective, a converter port appears as a regular port and is addressed with data that respect the master semantics.

Since it is the responsibility of the slave to provide the interaction mechanisms, these must be defined within the MoC. As such, the available interactions between MoCs are statically defined. This means that a MoC provides interaction mechanisms for a set of MoCs, thus defining its own

available interactions; a MoC will not be allowed to interact with a MoC for which no mechanisms have been planned. We chose this approach because it allows a Model of Computation to remain unconcerned about the existence of slave MoCs.

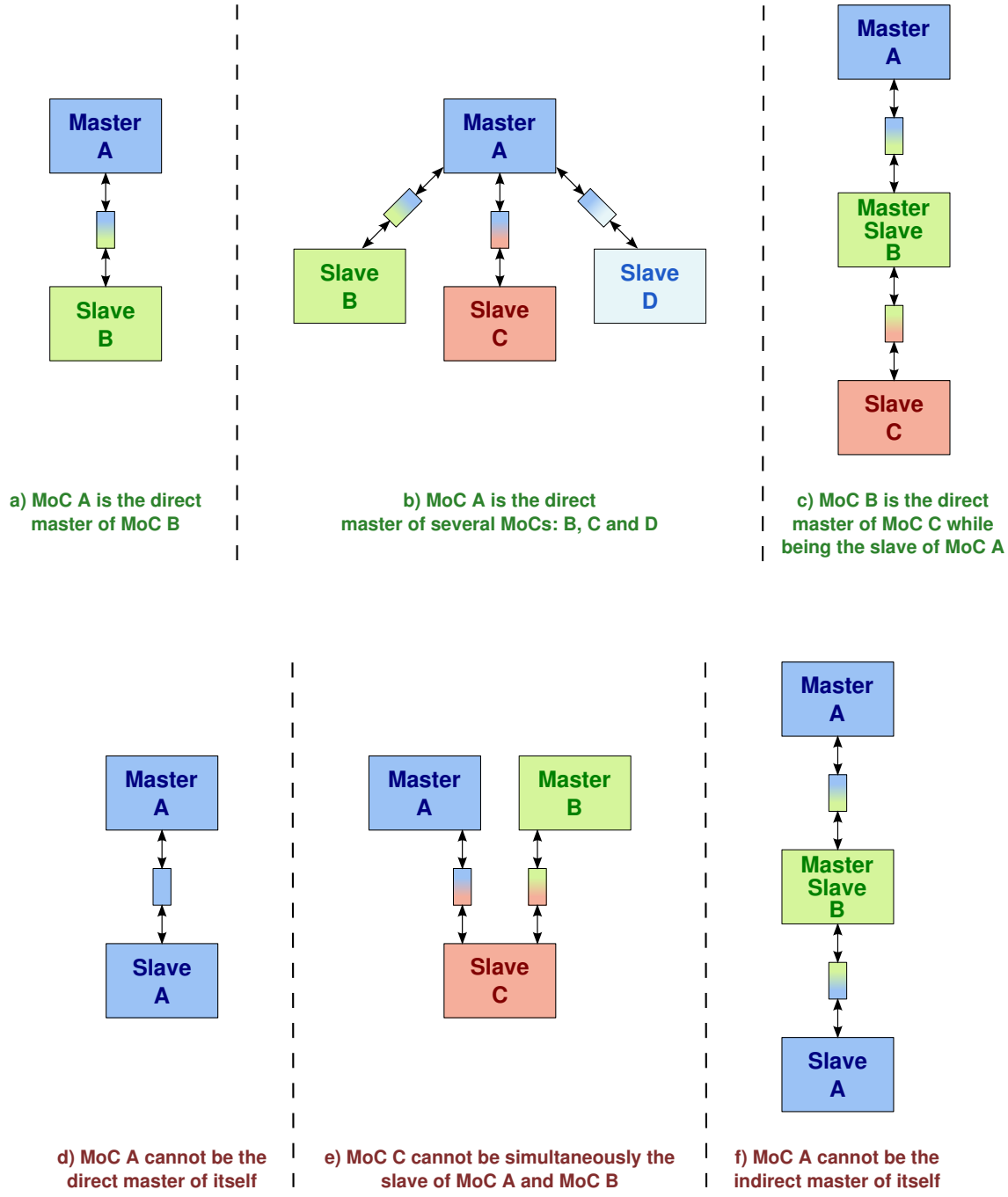

Figure 4.7: Authorized (a, b and c) and Forbidden (d, e and f) master-slave relationships

Although any given Model of Computation could provide available interactions for several masters, SystemC MDVP allows a slave to interact with only one master within a set of interconnected modules. It also implies that, in a separated set of interconnected modules, a given Model of Computation can interact with a master in one set and another master in a different set. Figure 4.7 illustrates the authorized and forbidden master-slave interactions between MoCs inside a set of interconnected modules within SystemC MDVP. One can see that a master could

simultaneously interact with several slaves since it is not aware of their existence (Figure 4.7.b). Being a master or a slave is not an exclusive state; a MoC can simultaneously be the master of another MoC and the slave of a third MoC (as illustrated by the MoC B in Figure 4.7.c). We see, in the above explanation, that a slave cannot simultaneously interact with several masters (Figure 4.7.e). Furthermore, it is important to note that a Model of Computation cannot be its own master or slave (Figure 4.7.d), regardless of the presence of intermediary MoCs (Figure 4.7.f).

While this approach may appear very restrictive (a MoC can be managed by only one master MoC), it allows the right set of interfacing mechanisms to be implicitly chosen during the elaboration of the system being modeled. That is in contrast to Ptolemy II's position, where the designer has to explicitly add the right interfacing mechanism himself (Directors). Another benefit lies in the fact that, as opposed to Ptolemy II, our approach avoids the explicit description of the simulation hierarchy concurrently with the model, cancelling error-prone intermingling of simulation artefacts with the model description.

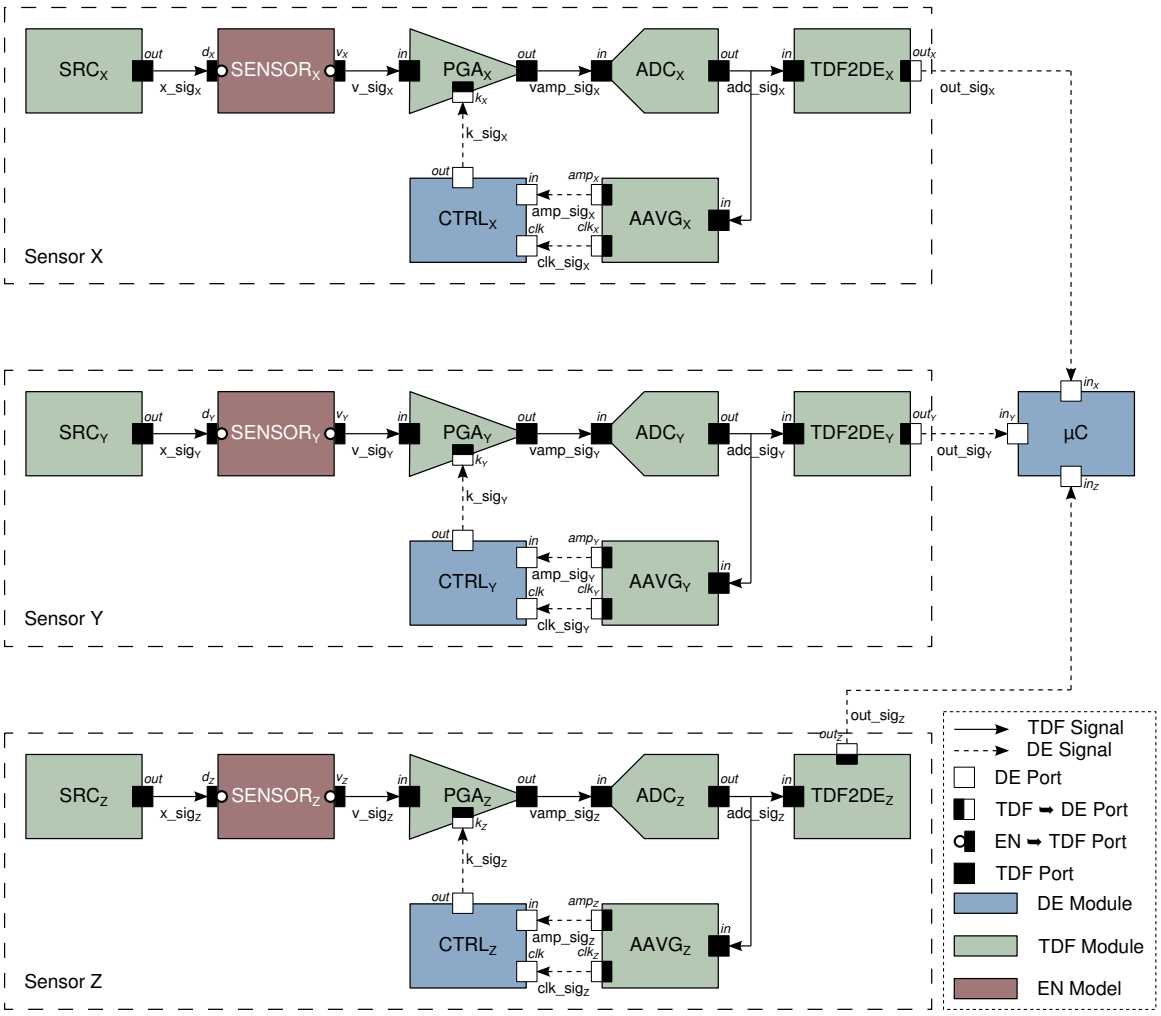

Figure 4.8: System on Chip representing a 3-axis vibration sensor as an assembly of MoCs.

We have seen what a Model of Computation is and how it interacts with other MoCs, therefore we may now provide another representation of Figure 4.1 which describes the running example used throughout this chapter. Figure 4.8 details how a vibration sensor can be modeled using the

EN, TDF and DE MoCs. The micro-controller's description consists only of SystemC components; however, since we are focusing on the heterogeneous aspect of the system, its description is not provided here. Sensors X, Y and Z are constructed following the same model only the axis detection differs from one sensor to another.

For ease of understanding, the different components involved in the design of the sensors are detailed below. The application, shown in Figure 4.8, is actually composed of a vibration source, *SRC*, (modeled as a generic harmonic sine wavelet generator with a TDF module), a vibration sensor, *SENSOR*, which converts mechanical displacement into proportional voltage (modeled by means of EN MoC) and the sensor frontend which outputs a digitized representation of the vibration sensor output voltage. The sensor frontend contains a programmable amplifier, *PGA*, an analog-to-digital converter, *ADC*, a component that computes digital average, *AAVG*, (all three components modeled by means of TDF modules) and a gain controller unit, *CTRL*, (modeled through a DE module) connected in closed loop to the programmable amplifier. Finally the *TDF2DE* component simply converts the TDF value into a DE value.

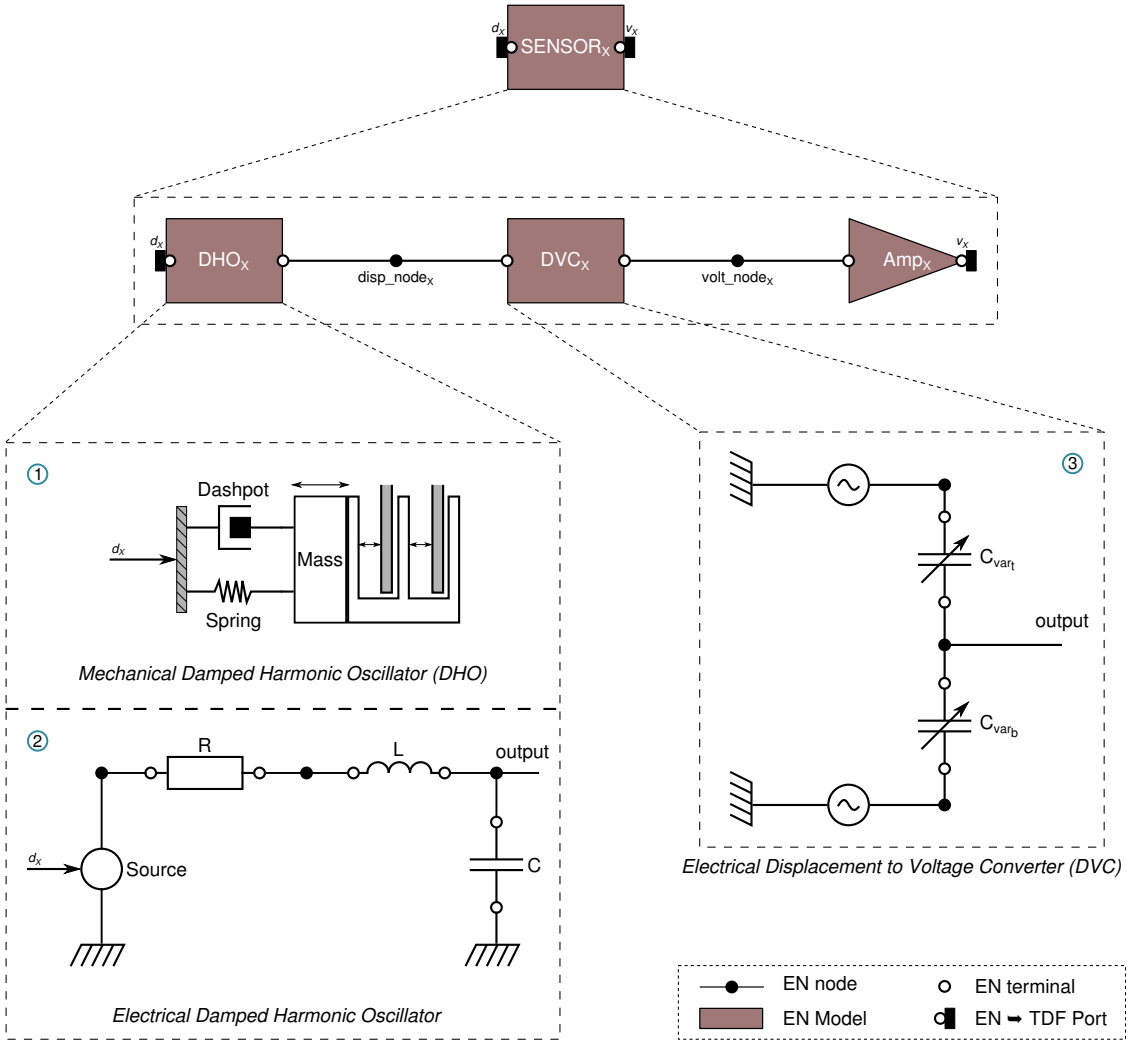

Figure 4.9: Running Example: Sensor description

The detail of the vibration sensor, *SENSOR*, is provided in Figure 4.9. The sensor can be

represented by the association of a *Damper Harmonic Oscillator (DHO)* with a *Displacement to Voltage Converter (DVC)*. Usually a Damper Harmonic Oscillator represents a mechanical system composed of a dashpot (a damper), a spring and a mass linked to a comb (①). When a displacement occurs, the mass moves and the comb is displaced, resulting in the modification of the distance that separates the electrodes from the comb. This modification of the distance allows for the identification of displacements. In order to model this behavior with the EN MoC, we can use an electrical equivalent model (②) represented by an RLC circuit. In this circuit, the coil L represents the mass, the electrical elastance $1/C$ corresponds to the spring constant and the resistance R matches the damping factor of the dashpot.

The information provided by the Damper Harmonic Oscillator is then used to generate a voltage value corresponding to the displacement (③). The displacement information is used to modify the value of the variable capacitors. Finally, the voltage signal is amplified before leaving the sensor.

For the sake of clarity and simplicity, we will not use the detailed representation of the sensor in the remainder of this document. Depending on the requirement, we will represent the sensor through its more abstract representation (just the *sensor box*) or with the intermediate representation (composition of three boxes *DHO*, *DVC* and *Amp*).

We can see that the EN MoC provides converter ports for the TDF MoC. For this reason EN is considered as a slave of, and is, therefore, controlled by the TDF MoC. Similarly, as the TDF MoC provides converter ports for the DE MoC, it is looked upon as a slave of DE and is, in consequence, controlled by it.

The master-slave semantics that we defined within SystemC MDVP as an interaction mechanism between Models of Computation, is a strong and coercive concept in our approach to the simulation of heterogeneous systems. Stating that a master MoC does not need to be aware of the existence of potential slave MoCs (implying that a slave MoC has to comply with its master interface) allows us to represent these relationships/interactions as an encapsulation process. The slave MoC is encapsulated within its master, the slave sub-model is abstracted inside the master model. This approach is especially well-suited to a hierarchical environment.

4.4 MDVP Hierarchical Approach

Since our master-slave semantics naturally fit with hierarchical behavior, it is only rational that SystemC MDVP should rely on a hierarchical heterogeneity approach. This allows us to take full advantage of the interaction mechanism between Models of Computation.

Starting with the flattened representation of the system (provided by SystemC), the framework creates a hierarchical tree with respect to the design and the master-slave interactions. This means that the framework constructs a hierarchical abstraction of the system wherein each

node represents a Model of Computation. Building SystemC MDVP upon SystemC should be interpreted as SystemC being the master of the entire SystemC MDVP environment with respect to our interaction semantics. From a framework point-of-view, we need to respect SystemC semantics and SystemC has to be the root of the hierarchical tree. This hierarchical tree acts as an efficient representation which allows us to easily identify the scope of a Model of Computation and, therefore, to handle automatically the interactions between MoCs since the master-slave semantics are respected at all points in the hierarchy.

The scope of a Model of Computation represents all the interconnected modules belonging to the same MoC; it also includes the modules belonging to a slave MoC. This scope represents a node in the hierarchical tree of the system and we refer to it as a cluster. To detect the scope-limits associated with a Model of Computation, we identify special ports - the converter ports. As previously mentioned these ports provide compatibility among MoCs and represent their borders. Regarding the hierarchical tree, regular ports are inside a hierarchical node whereas converter ports are at the border of a node in order to enable communication with the higher hierarchical node.

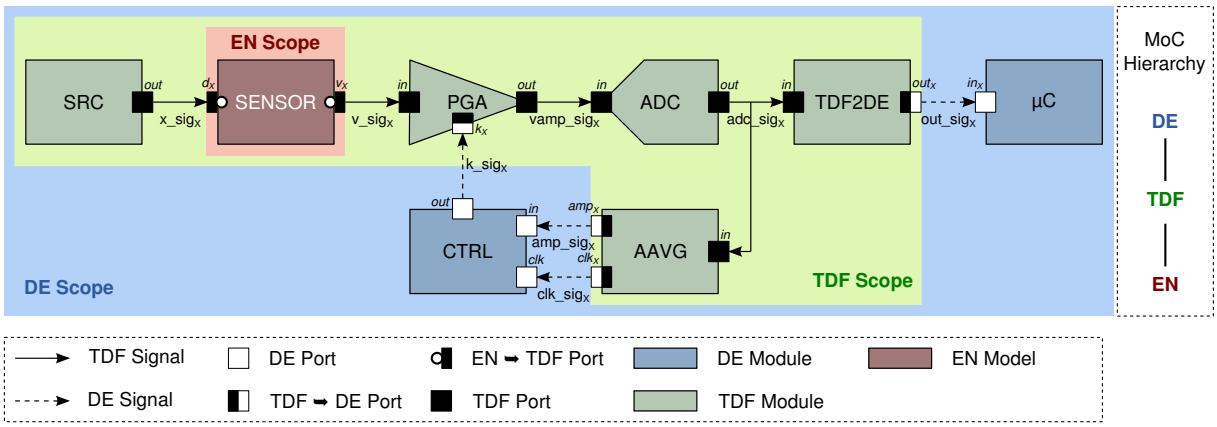

Figure 4.10: Running Example: Scope of EN, TDF and DE MoCs

Figure 4.10 highlights the scope of each MoC used in the running example of the vibration sensor. We can easily identify the relationships between these three MoCs and visualize their respective scopes. The figure also describes the encapsulation principle that follows on from the master-slave interaction mechanism: EN is encapsulated within TDF, as is TDF within DE.

Figure 4.10 clearly illustrates the hierarchical organization of the MoCs; the sensor component is modeled by means of the EN MoC which will behave like a TDF component. The TDF frontend will, in turn, behave like a DE component. This transitive interfacing scheme prevails throughout the whole system producing a clear-cut MoC hierarchy with master-slave relationships. The fundamental purpose of the SystemC MDVP approach is to define all the principles, tools, and API functions which allow a model sub-tree to behave as if it were a standard module of the MoC it is encapsulated in.

4.5 User Profiles

For efficiency, we believe that an environment design needs to be adapted to the user who exploits it. To this aim, we profiled three different kinds of user with different competences, needs and expectations. Indeed, each kind of users corresponds to a layer of the simulator, where the mechanisms, the data structures and the principles should or should not be known by the user depending on its objectives. The different profiles that we identified have specific objectives, and hence, require access to different information.

The first profile identified is the *Simulator Architect* in charge of the definition of the simulator kernel. Second, we identified the *MoC Architect* profile; he is he who defines new Models of Computation. Lastly, we found the *SoC Architect* profile, which corresponds to the designer of a heterogeneous system. This section details the aforementioned profiles.

4.5.1 Simulator Architect

The *Simulator Architect* represents the deepest layer of the SystemC MDVP framework. His objective is to enable the modeling and the simulation of heterogeneous systems while remaining generic. Therefore, he knows the implementation details and the internal mechanisms involved in the core of the framework while staying unaware of any specific details concerning Models of Computation. Indeed, if we want to guarantee the flexibility of our virtual prototyping environment, it is imperative that the Simulator Architect remain completely oblivious to the MoC definition. In effect, he is responsible for providing generic algorithms and mechanisms that will allow for the virtual prototyping of heterogeneous systems, regardless of the MoCs used to describe these systems.

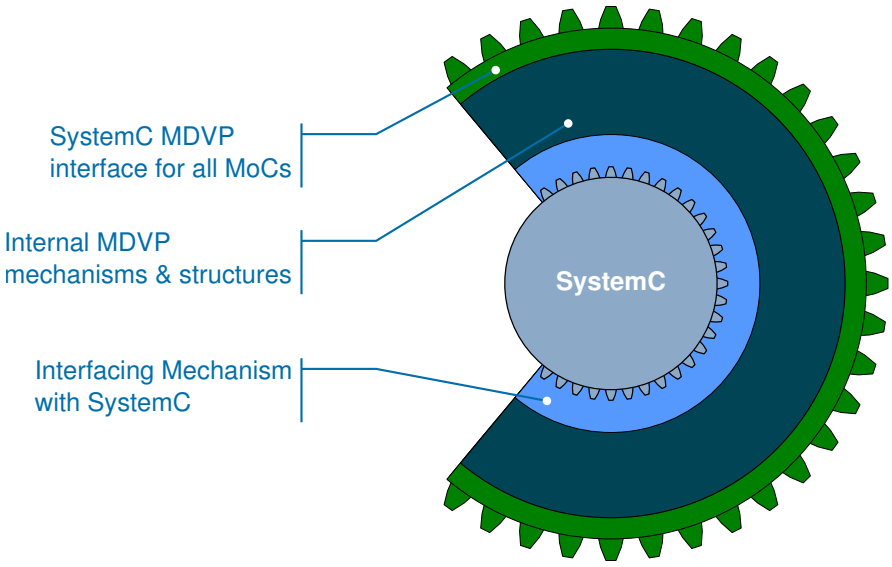

Figure 4.11: SystemC MDVP: Simulator Architect

Figure 4.11 describes the Simulator Architect vision of our framework, and expresses how

SystemC MDVP is positioned in regards of SystemC. We see that the Simulator Architect is familiar with the interface provided by SystemC, and hence, in order to run SystemC MDVP on top of it, he defines a SystemC-compliant interface within SystemC MDVP.

On top of this interface, he defines all the internal mechanisms and data structures required to support the modeling of heterogeneous systems through the use of multiple Models of Computation.

He is, obviously, the one who defines the interface that SystemC MDVP will put forward for the other users. This interface corresponds to the definition of a MoC abstraction. This MoC abstraction allows the MoCs to be plugged with our environment. It also ensures the proper functioning of the internal mechanisms and data structures.

Even if his role seems limited through this figure, the Simulator Architect is the one who makes the heterogeneous modeling possible. Indeed, the definition of an interface in order to be coupled with SystemC and the definition of a common representation for all the MoCs allow for the modeling of such systems. Therefore, he oversees the simulation of all the different components and exploits the SystemC kernel making sure to take full advantage of what this kernel can provide. The Simulator Architect should know nothing more than what is illustrated in Figure 4.11. Our approach makes the internal details of the MoCs definition irrelevant from his perspective.

4.5.2 MoC Architect

The *MoC Architect* represents the intermediate layer of the SystemC MDVP environment. His objective is to design and define new Models of Computation. To this aim, he doesn't need to know the internal mechanisms and details involved in the core framework. Similarly, he doesn't have to know the internal implementation details of other Models of Computation. However, he needs to know how to integrate a new MoC within the SystemC MDVP environment. In addition, he also has to know how to setup interaction mechanisms with other MoCs if he desires to communicate with them. Finally, he obviously knows the implementation details and the internal mechanisms involved within the new MoC he is designing.

Figure 4.12 describes the MoC Architect perception of our framework, and exhibits the positioning of a new MoC within the SystemC MDVP framework. To illustrate the MoC Architect vision, in the following we suppose that a MoC Architect wants to integrate a new Model of Computation, the MoC C, within the SystemC MDVP environment where the MoCs A and B are already defined.

We mentioned that the MoC Architect doesn't have to know the internal details of the core framework or those of other MoCs. Consequently, we see that all the existing MoCs evolving in our framework, and SystemC MDVP itself, behave like black boxes to the MoC Architect except for the MoC interface defined within each MoC and the interface provided by SystemC MDVP. Therefore, these MoC interfaces and the SystemC MDVP interface represent the only resources he

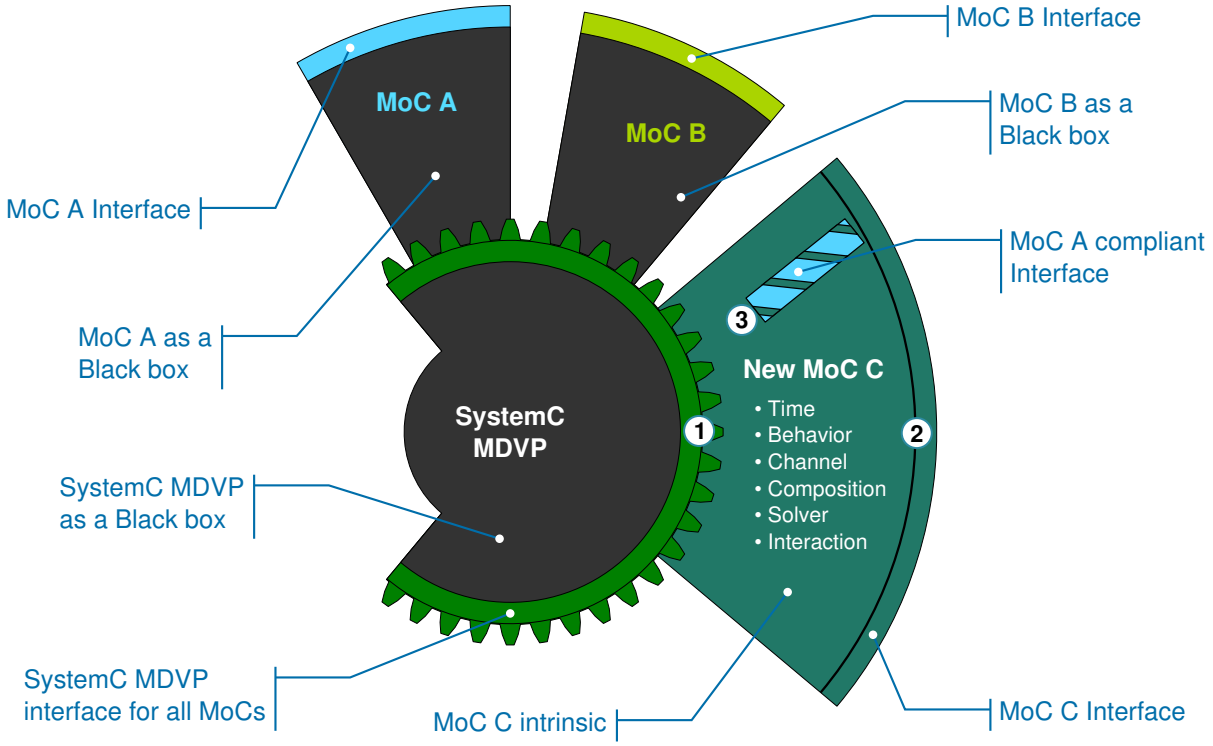

Figure 4.12: SystemC MDVP: MoC Architect

has at its disposal. As well as this collection of interfaces, he naturally has an intrinsic knowledge of his MoC.

In order for the new MoC C to be taken into account within our virtual prototyping environment the MoC Architect defines an interface that complies with the SystemC MDVP interface (Figure 4.12, ①). In addition to defining the intrinsic mechanism involved within the new MoC C, he also exposes a single MoC interface associated with the MoC C to allow other MoCs to communicate with his MoC (Figure 4.12, ②).

Finally, the MoC Architect can setup the interaction with other MoCs he desires, with respect to the master-slave semantics. Among the MoCs evolving in the SystemC MDVP environment he chooses to make his new MoC C communicate with the MoC A. Therefore, he defines a MoC A-compliant interface that will allow the communication between the MoCs C and A (Figure 4.12, ③).

Since the MoC Architect defines, for his new MoC C, the mechanism in order to communicate with other MoCs he knows which MoCs can be master of his MoC. In contrary, since all other MoCs appear as black boxes he cannot know which MoCs can be slave of his MoC. With this approach, as with the Simulator Architect, we ensure that our virtual prototyping environment is flexible by keeping dependencies between different entities to a strict minimum.

4.5.3 SoC Architect

The *SoC Architect* constitutes the shallowest layer of the SystemC MDVP environment. His objective is to design and model heterogeneous systems relying on the set of Models of Computation evolving in our environment. He is the one confronted with the issue of heterogeneity since it is he who builds and models new heterogeneous systems. For the SoC Architect, Models of Computation represent sub-models he wants to nicely assemble in order to describe his complete system. These sub-models need to be clearly defined and readily distinguishable so that he may handle them with ease. Therefore, he knows all Models of Computation available to build his system and all the interactions possible between them (i.e. all the master-slave relations). His main task is to connect basic blocks in order to design a complete heterogeneous system. He defines the behavior of the components assembled in his system and sets up the communication interactions between these basic blocks. He does not have to know the internal specification of the framework or specific details about the MoC, nor is he obliged to apprehend the internal mechanisms involved in the interaction between MoCs. The SoC Architect's perception and use of our framework SystemC MDVP is depicted in Figure 4.13.

Figure 4.13 shows that the SoC Architect has at its disposal a set of Models of Computation available within our environment (Figure 4.13, ①) and that he also disposes of the authorized interaction between these MoCs (Figure 4.13, ②). Using both of these information, the SoC Architect can describe its system; subject to the respect of our master-slave semantics, he couples models from different MoCs belonging to our framework. He can also combine them with models described using SystemC and coupled with an embedded software. Following this approach, the SoC Architect defines an assembly of models from all the environment (Figure 4.13, ③). Finally, this assembly of MoCs allows him to represent and model different physical disciplines which are embedded in the system modeled (Figure 4.13, ④).

With this profile, we highlight the importance of a clear definition of the entities handled and manipulated within a virtual prototyping environment dedicated to heterogeneous systems. We also accord a specific importance to the interaction mechanism involved between these entities which should not be managed by the SoC Architect. To further illustrate the flexibility of our framework, it should be noted that the SoC Architect can carry out the design of a system, based on the master-slave relationships defined within each MoC, without requiring additional information.

4.6 Conclusion

This chapter introduced the principles underlying the SystemC MDVP framework. Starting from the description of several Models of Computation, we extracted the relevant features which constitute the essence of a MoC, i.e. the information required to represent a MoC, and thereupon we provided a way to abstract any MoC. This abstraction, along with the definition of a MoC,

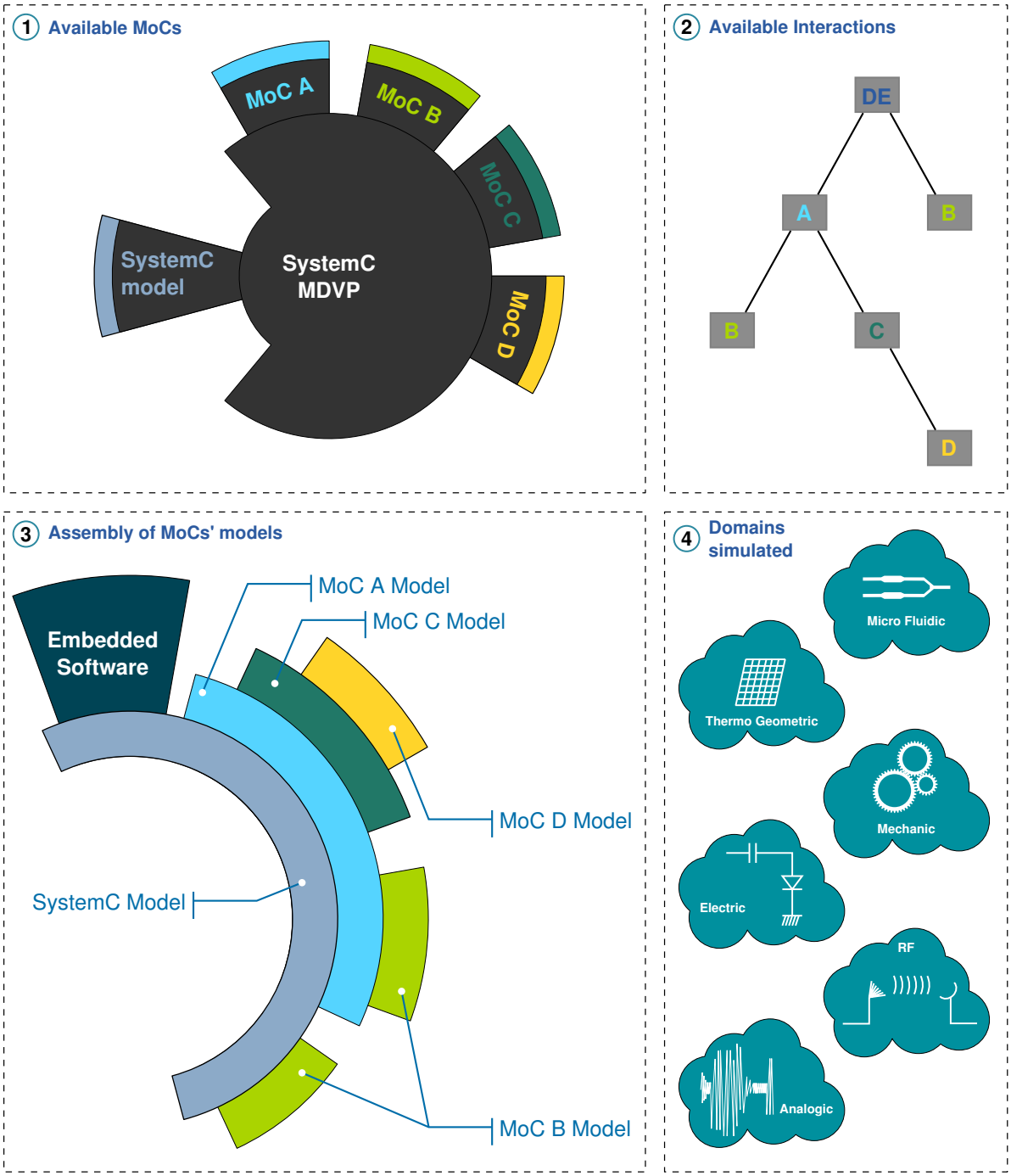

Figure 4.13: SystemC MDVP: SoC Architect

constitutes our definition for what is intended with *heterogeneity*. The Model of Computation represents the entity handled and manipulated by our framework.

In a second step, the notions of master-slave semantics that we defined, within SystemC MDVP, as an interaction mechanism between Models of Computation, represents a strong concept in our approach to the simulation of heterogeneous systems. These semantics allows for a seamless interaction between different MoCs, thus guaranteeing the flexibility of our virtual prototyping environment. Stating that a master MoC does not need to be aware of the existence of potential slave MoCs implies that a slave MoC must comply with its master interface, i.e. fulfill all the

requirements and meet all the expectations imposed by its master. This position makes it possible for these relationships to be represented as an encapsulation process. Our approach specifies that all the available interactions between MoCs are statically defined and that they are provided by a slave MoC. The slave MoC is encapsulated within its master, the slave sub-model is abstracted inside the master model. This approach is especially well-suited to a hierarchical environment and hence explains our choice to follow a hierarchical heterogeneity approach, in order to take full advantage of the interaction mechanism within SystemC MDVP.

When it comes to model parts that are located at the boundary of two MoCs, it is important to note that the designer should not be faced with questions that are deeply linked to the simulation infrastructure. Designers want to connect parts or third party models coming from different sources and not address issues such as the choice of the appropriate director (as in Ptolemy II) or the setting of an obscure simulation parameter. Our virtual prototyping framework allows the user to focus only on the important and relevant things thanks to the identification of different user profiles. Each of the three profiles identified, Simulator Architect, MoC Architect and SoC Architect, has a different representation of the system and, hence, different requirements. We can depict the abstraction of these users as concentric circles going from the center outwards. The inner circle maps the scope of the Simulator Architect, the intermediate circle the scope of the MoC Architect, and the outward circle, the scope of the SoC Architect. For example, the SoC Architect is allowed to experiment without having to become involved with the details of Model of Computation handling.

We bring out our vision of a digital-centric ecosystem, where the digital is at the heart of the system with other engineering domains gravitating around it. These profiles allow us to better cope with the expectations and requirements of a simulation framework for heterogeneous systems. These profiles have been taken into account in the aforementioned principles; they also drive and motivate the way the framework is implemented. The implementation that supports the principles introduced in this chapter is detailed in the following chapter.

Chapter

5 *SystemC MDVP: Implementation*

Contents

5.1 Introduction 52

5.2 SystemC MDVP Kernel Representation 52

5.2.1 SystemC MDVP MoC Abstraction 53

5.2.2 SystemC MDVP core classes 55

5.3 Elaboration Phase 57

5.3.1 Composability / Static Analysis Sub-phase 58

5.3.2 Clustering Sub-phase 62

5.3.3 Solver Instantiation Sub-phase 67

5.3.4 Basic Behavior Block Elaboration Sub-phase 70

5.3.5 Port and Channel Elaboration Sub-phase 72

5.3.6 Elaboration conclusion 73

5.4 Simulation Phase 74

5.4.1 Simulation Mechanism 74

5.4.2 Simulation Opportunities 77

5.5 Conclusion 78

5.1 Introduction

This chapter introduces the implementation details underlying SystemC MDVP that support the principles introduced in Chapter 4. As a reminder, we want to provide a virtual prototyping environment with a smooth management of heterogeneity, a sound management of the interaction between entities and a multi-disciplinary monitoring mechanism all the while, remaining flexible.

In order to provide a high level of flexibility, the algorithms used within the SystemC MDVP kernel must be independent from the Model of Computation definition. They also need to remain generic during the different phases of the life cycle of the simulator. Throughout the course of this chapter, the running example of a 3-axis vibration sensor introduced in the previous chapter is used to illustrate these mechanisms which are implemented within our virtual prototyping environment.

Keeping in mind these notions of independence and genericity, this chapter proposes an overview of the implementation details underlying SystemC MDVP organized as follows.

In Section 5.2 we present the infrastructure of the SystemC MDVP framework. We detail how a Model of Computation is represented within our simulator kernel and the resulting abstraction. We also introduce the core classes that describe the behavior of the kernel and that allow for the integration of SystemC MDVP on top of SystemC.

In Section 5.3 we detail the elaboration phase which constitutes the phase wherein the entire environment for the simulation is set up. The generic algorithms which allow the framework to operate independently of the MoC are introduced.

The next phase in the life cycle of the simulator, the simulation phase, is presented in Section 5.4. We illustrate the simulation mechanisms involved within SystemC MDVP and how the work performed during the elaboration phase is leveraged. Following this, we discuss the opportunities provided by our approach in terms of simulation possibilities.

To conclude, Section 5.5 provides an overview of the implementation details underlying the SystemC MDVP framework and brings this chapter to a close.

5.2 SystemC MDVP Kernel Representation

This section details the intrinsic elements of the SystemC MDVP framework. In Chapter 2, we highlighted the importance of the representation of heterogeneity when dealing with the simulation of heterogeneous systems. In light of this, we first present the underlying representation of a Model of Computation within SystemC MDVP. We model our implementation on the abstract representation of a MoC we previously defined (Chapter 4 Section 4.2). We then introduce the core

classes of our simulator that allow for the integration with SystemC and the proper functioning of our framework.

5.2.1 SystemC MDVP MoC Abstraction

The Simulator Architect requires an abstraction of a Model of Computation if he wants to provide a generic kernel capable of handling both existing and forthcoming MoCs. To this aim, we defined a set of base `C++` classes that allow for the representation of a MoC. We defined these base classes with respect to the functional abstraction of a Model of Computation we highlighted in this work. In order to take advantage of the SystemC framework, we construct our representation of a MoC on top of SystemC-specific components. This is achieved using the inheritance concept in order to abstract our MoC representation from a SystemC viewpoint. Figure 5.1 displays the base classes defined within SystemC MDVP to abstract a MoC and illustrates how they inherit from SystemC. These classes are gathered within a single `C++ namespace: scm_core`. This namespace contains all the classes that define the intrinsic of the SystemC MDVP framework.

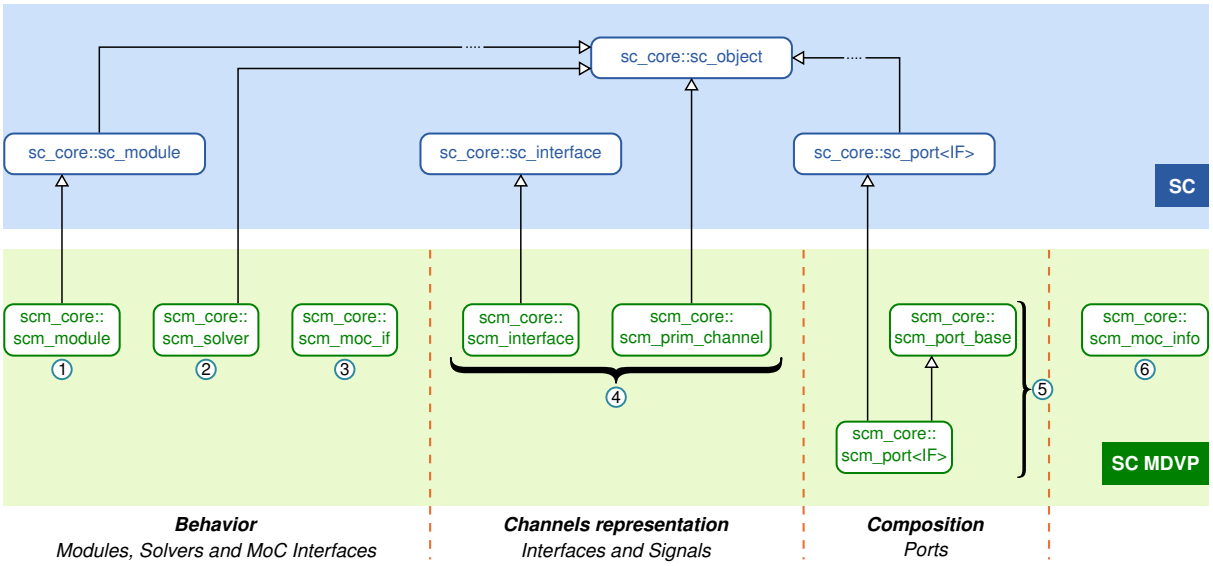

Figure 5.1: SystemC MDVP base classes to abstract a MoC

We define a basic module which represents the primitive elements of a Model of Computation as well as a basic port and a basic channel which address the issue of communication by demonstrating the manner in which to connect different modules together. These classes represent the objects belonging to a Model of Computation that the SoC Architect will manipulated. Concerning the internal mechanism of a MoC, we simply need a base class to represent the solver and a class to express its interface. There is no need for additional information on specific details regarding the MoC's purpose or implementation.

The following section describes the purpose of each class and how they abstract a MoC with respect to the aforementioned functional abstraction.

- Primitive behavior ①.

Within SystemC MDVP, to represent a basic block (a module), we only need one class - `scm_module`. This class allows us to abstract the basic blocks defined within each MoC. In this class there is all the functions that the kernel SystemC MDVP requires in order to handle a module. The functionalities provided within this class are not dependent on the MoC to which the module belongs so they can be generalized in the base class within the kernel. For example, there is a function to perform an automatic registration of a module when it is instantiated in a model description. This class inherits from the SystemC `sc_module` class since it is the easiest way for SystemC to handle our modules and, consequently, allow us to use SystemC functionalities on them.

- Solving Algorithm ②.

In order to represent a solver, we also need a single class `scm_solver`. As with the modules, it allows us to define basic functionalities for all the SystemC MDVP solvers defined within each MoC. This class inherits from the SystemC `sc_object` class which provides a set of basic functionalities useful when handling our solvers such as the mechanism used to register a name and associate it to on object. For example, it provides the mechanism to register a name and to associate it to an object.

- Interaction ③.

The master-slave semantics require that a slave comply with its master semantics and that it be possible to encapsulate the slave within the master. That is to say, the slave must be seen as a single model from the master view point. Accordingly, a slave must express the same functionalities as a master module when encapsulated in the master MoC. Within the SystemC MDVP framework, we established a new class called `scm_moc_if`, which defines the interface of a MoC. All of the functionalities that a MoC expects to find in one of its own modules can be found in this class.

Given that a MoC handles all of its primitive modules through this interface, it is via the same interface that a slave MoC is addressed. A MoC is driven to manipulate its primitive module as well as, potentially, encapsulated slave. Thus, the `scm_moc_if` can represent either a primitive module or an abstract representation of a slave MoC.

- Channel Representation ④.

In order to represent a communication channel in SystemC MDVP, we followed the same approach as with SystemC; we use two classes - an interface and a channel. `scm_interface` defines

a *communication interface* which consists of virtual functions implemented by a channel. This class inherits from the SystemC `sc_interface` class with the aim of benefiting from the functionalities provided by SystemC. However, some of the functionalities performed by `sc_interface` are declared as private in the kernel SystemC and, thus, we cannot access them. This limitation requires that we perform these tasks locally.

`scm_prim_channel` expresses the base functionalities of a channel and, to address the aforementioned limitation, also defines the functionalities that need to be re-implemented. As an example, although `sc_interface` performs the registration of ports associated with a channel, we re-implemented this functionality within SystemC MDVP. `scm_prim_channel` inherits from `sc_object`, which allows our channel to benefit from the basic functionalities of SystemC objects. Communication channels in SystemC MDVP are the result of the inheritance of these two classes.

- Composition ⑤.

The ports within SystemC MDVP are expressed with two base classes. `scm_port<IF>` defines common base functionalities that mainly rely on SystemC, explaining why it inherits from the SystemC `sc_port<IF>` class. It also requires the template parameter to allow for the specification of the communication interface associated with a port.

`scm_port_base` also defines common base functionalities, though these ones do not rely on SystemC. This class is the one used within SystemC MDVP kernel to abstract a port, therefore, contrary to `scm_port<IF>`, it does not express a template parameter and mainly contains handling functionalities. Handling functionalities include, for example, a set of functions that define *getters* for the different attributes of a port such as its name, interface or, indeed, its position regarding to the hierarchy of object instantiated in the model (e.g. the module it belongs to).

Finally, within SystemC MDVP, an additional class is introduced in order to centralize common information about a MoC and access to this information with ease ⑥. `scm_moc_info` plays the role of this class. It stores common data associated with the MoC such as its name and its list of available interactions, i.e. the list of potential masters with regards to the master-slave semantics.

5.2.2 SystemC MDVP core classes

We have seen how SystemC MDVP represents a MoC without the need for MoC-specific implementation details; it remains to demonstrate how the internal framework architecture is constructed. SystemC MDVP only relies on few classes in order to describe its behavior, all defined within the `scm_core` namespace. We have one class to represent its kernel (`scm_core::scm_simcontext`) and three more to handle specific internal mechanisms (`scm_core::scm_cluster_node`, `scm_core::scm_cluster_creator`, `scm_core::scm_moc_interface_creator`). Figure 5.2 illustrates the core classes of SystemC MDVP, and the interaction with SystemC.

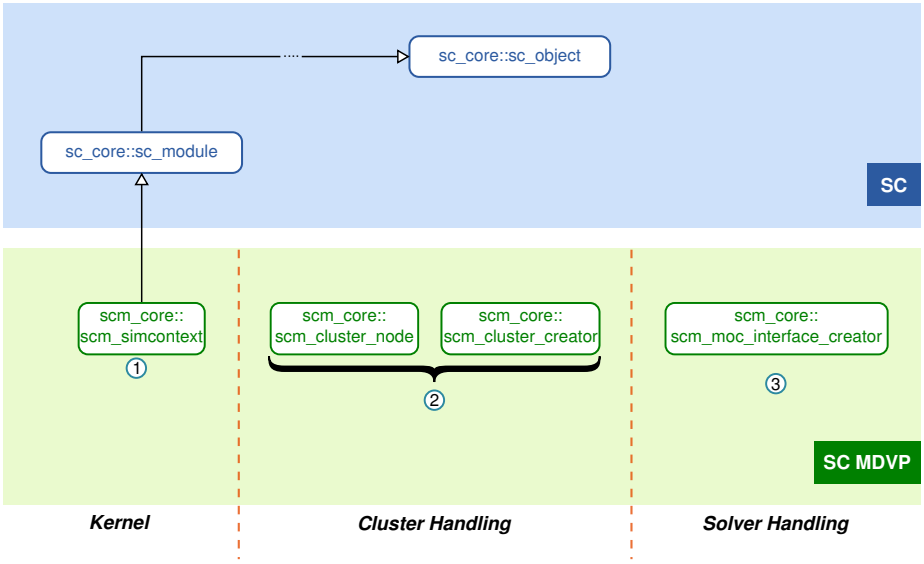

Figure 5.2: SystemC MDVP intrinsic classes

We stated in the introduction of this chapter that our framework is an extension of SystemC; we successfully extended SystemC using the same inheritance principle previously mentioned. We define a *Simulation Context* object through the `scm_core::scm_simcontext` class ①. This class inherits from the `sc_core::sc_module` class of SystemC, allowing our framework to interact with SystemC. This particular class represents the entry-point through which SystemC MDVP passes into SystemC without altering the DE simulation kernel. In addition to handling all of the MoCs, the simulation context contains the integral components of SystemC MDVP along with all of the algorithms.

Extending the `sc_core::sc_module` class of SystemC gives us access to a set of useful callbacks that allow our simulator to perform its own simulation procedures. Two especially helpful callbacks are `end_of_elaboration()` which we use to encapsulate and perform the elaboration phase of our framework, and `start_of_simulation()` which allows us to encapsulate and perform the simulation phase of SystemC MDVP.

The classes `scm_core::scm_cluster_node` and `scm_core::scm_cluster_creator` allow for the handling of the clustering process ②. `scm_core::scm_cluster_node` describes the cluster structure used to represent an homogeneous region within an heterogeneous system. `scm_core::scm_cluster_creator` contains the generic algorithm that performs the clustering and, hence, constructs the hierarchical representation of the system modeled. The class `scm_core::scm_moc_interface_creator` contains the algorithms required to perform the automatic and generic instantiation of solvers within SystemC MDVP ③.

5.3 Elaboration Phase

The work performed in the kernel focusses on the elaboration process since, being responsible for providing all the resources required to perform the simulation, it represents the crucial part of the simulation Any given system model is built from primitive modules (describing an atomic behavior) and instantiated modules, which are connected to each other and prepared for simulation during the elaboration phase. SystemC MDVP framework is built on top of SystemC and, as such, may benefit from the regular elaboration phase of the DE simulator kernel. To perform a generic elaboration phase with multiple unknown Models of Computation, we identify specific sub-phases which are detailed in the Figure 5.3. This figure resumes the Figure 3.3, described in Chapter 3, and enriches it in order to illustrate how the SystemC MDVP elaboration phase extends SystemC to handle heterogeneity while still refraining from altering its kernel. We use the callback `end_of_elaboration()` from SystemC to trigger the execution of our own elaboration mechanism.

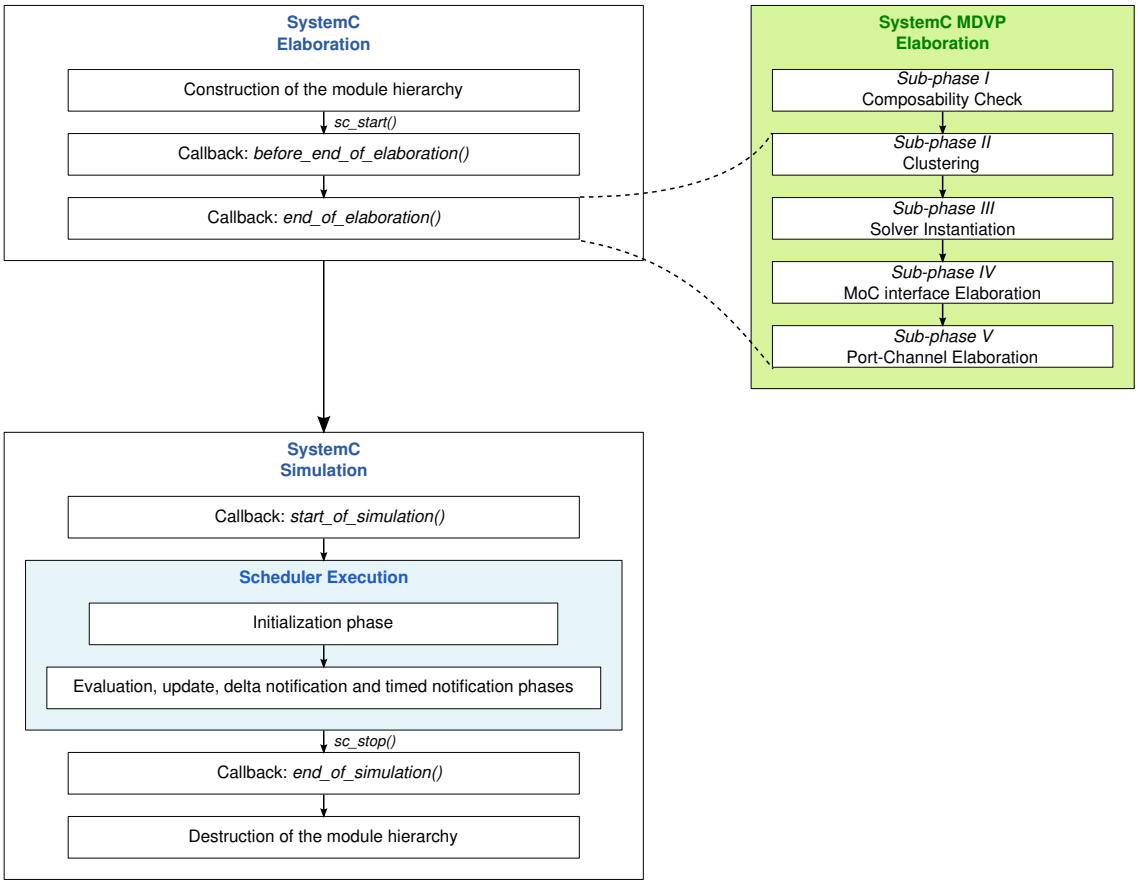

Figure 5.3: SystemC MDVP kernel elaboration phases.

Strictly speaking, *sub-phase I* is not performed during the elaboration, as it has already been realized during the compilation, nonetheless, the composability check perfectly fits with the intent of the elaboration phase. This is why we decided to show this step as part of the elaboration process. Other sub-phases are realized during the `end_of_elaboration()` callback of SystemC, which guarantees that the flattening of the system is complete (each primitive module has been

identified and directly or indirectly linked to a MoC). *Sub-phase II* describes the clustering step, where the system is being explored and analyzed, with relevant information extracted accordingly, to perform the simulation. *Sub-phase III* exploits the results of the previous phase to set up all of the solvers required to simulate the system. Eventually, during *sub-phases IV and V* the elaboration of each component of the system modeled is realized. First of all, MoC interfaces are elaborated; this includes modules and solvers since they are abstracted as MoC interfaces. Thereupon, ports and channels are elaborated.

All sub-phases which constitute the elaboration phase are detailed in the following section.

5.3.1 Composability / Static Analysis Sub-phase

One important aspect within our framework is the development of a "correct-by-construction" approach for the integration of MDVPs with the targeted, multi-domain, electronically-assisted systems. When it comes to simulating microelectronics-assisted systems with different physical domains, the question of composability and compatibility between these domains arises. Thus, to address this question, consistency checking, together with dimensions and units checking, must be performed at the interface between the domains but also inside the description of the modules belonging to each domain. These checks are essential for validating the correctness of the physical interfaces.

Today's model compilers and simulators only perform rudimentary checks, for example on the compatibility of the data types involved in an operation or in respect to the requested interface. Although they are useful, these tests do not fully take into account data semantics (e.g. nature of the data, dimension, units, etc.). Without such information, the interface will be defined using primitive data types (e.g. double) which make the assembly of multi-domain systems error-prone. Thus, with the objective being to perform the simulation of microelectronics-assisted systems with different physical domains, the definition of unequivocal physical data types is needed.

In keeping with our ambition to apply a "correct-by-construction" approach, we aim to make the measurement units, and other semantic meta-information, an intrinsic part of the model implementation in SystemC MDVP. This point of view fits the behavior of physicists and engineers who have a tendency, not to consider isolated values, but rather quantities, tightly linking the value to its measurement unit. Proceeding in such a manner will enable automatic checking at the model interfaces (right quantities assigned to parameters and ports bound to signal with compatible quantities) and promote global consistency in terms of the behavioral description (coherency of mathematical equations).

The composability of an heterogeneous model has to be checked as early as possible in the simulation process so that later phases can rely on a sound structure (Figure 5.3, *sub-phase I*). The composability check consists in verifying the connection between different system components so as to ensure a correct composition of this system from a dimensional analysis and, indeed, a

semantic point of view. The user should be informed early on in the process about problems concerning the mis-connection of ports with channels that do not respect the same semantics. An example of this would be the fact that continuous-time signals representing physical quantities are usually defined as values of type `double` which make them too generic and engender the incorrect binding of physical quantities belonging to different domains, e.g., optical and electrical.

Other problems include: forwarding of parameter values to the wrong parameter (due to the absence of named arguments in C++), and inhomogeneous model equations.

Most of these problems could be detected by the compiler through static type-checking if the variables, ports and channel types were more precisely defined in the model sources. The solution would be to express the data semantics associated with them. Unfortunately, these data structures are often described using a primitive type of the language (a `double` value for example), which does not allow the user to attach or to verify the semantics information carried by these structures.

Dimensional analysis and `C++` share a long history. The idea itself was introduced by Barton and Nackman in 1994. In this work, the key idea is to perform compile-time checking based on template arguments which represent the base units. The last release of `C++` [71], `C++'11`, introduces user-defined literals and *Bjarne Stroustrup* gives an example of dimensional analysis with user-defined literals.

The solution proposed to address the issue of integrating quantity and unit types (having now established their necessity) as well as dimensional analysis into SystemC AMS [60] can be also applied to our SystemC MDVP framework. This solution relies on the more recent open library `Boost.Units` [61] which implements dimensional analysis. `Boost::Units` has been chosen for this project since it already implements all the units from the international system of units and overloads all standard mathematical functions defined in `<cmath>` (standard C++ mathematical library).

`Boost::Units` comes as a `C++` library based on `C++`'s flexible type system to implement quantity and unit data types as template classes which support dimensional analysis at compile-time through template meta-programming. The quantity type defined by `Boost::Units` perfectly fits our needs. It associates a data type (e.g., `double`, `int`, etc.) with a unit (e.g., meter, volt, etc.). This quantity type can be used in mathematical equations or in the definition of an interface of a physical domain. The use of this quantity type can, therefore, prevent the interconnection of ports and channels of different natures and ensure the coherency of model equations. The unique `quantity` type (association of a data type with a unit) overloads only the legal arithmetic and assignment operators. Thus, the compiler issues a "missing overload" error for illegal operations, e.g., the sum of two quantities with different units or the assignment between incompatible quantities.

The `Boost::Units` library provides a mature implementation of dimensional analysis. It makes dimensional analysis possible for arbitrary systems of units at compile-time, as part of the static

type-checking phase, without requiring modifications to the compiler or an additional tool. As the unit is encoded into the quantity only as part of its type, and not as a member variable, modern C++ compilers can optimize this information away after the type checking phase. Thus, no runtime penalty is caused by doing arithmetic with quantities. Only the compile time is increased.

The main drawback to `Boost::Units` lies in the compiler error messages that result from illegal operations involving quantity types. Compiler messages are barely understandable, even unreadable, due to long template type names which are fully expanded by the compiler. While `Boost::Units` takes full advantage of the template programming, providing flexibility and efficiency when expressing meta-data, the downside is that all the template information is inherited in the error's message, thus missing the origin of the error.

Fortunately, in order to address this problem, a filtering application Bufilt has been developed (as it is usually done with generic libraries using a lot of template classes) to parse, simplify and format the error messages output [72]. Bufilt, for `Boost::Units Filter`, parses the error messages output to identify relevant information concerning dimensional aspect and realize a substitution of the template type names by more coherent messages which facilitate the identification, by the end-user, of the incoherency in the simulated system.

The use of quantity type will have a positive impact on the code quality as the designer can be much more precise when specifying the interfaces and computations in his model which involve physical quantities. The compiler can check the coherency of the connected interfaces and implemented equations. Bufilt increases error understanding meaning many problems can be localized and fixed before the first execution of the model. This gives the designer more time to test the important behavioral aspects of his model.

Using "Boost::Units" in one's day-to-day work is a little tricky and tedious because of the "hard" template objects which need to be manipulated. To solve this issue, and to clearly identify exactly which physical quantities are being manipulated, our approach suggests the use of a *common definitions* header file in which all the requested types are defined. The purpose of these definitions is to provide the end-user with a higher level of abstraction for domain types. All of the standard quantity types needed in a specific physical domain are, consequently, hidden behind `typedef`, thus preventing the end-user from manipulating heavily template objects. Using this mechanism, the underlying complexity of the template objects, which defined "Boost::Units", is conveniently hidden.

Listing 5.1, extracted from the common definitions file associated with the running example, defines the quantity types needed to model the vibration sensor. From a composability viewpoint, this example is interesting since there are different kinds of physical quantities being exchanged between these 3 blocks (source, sensor, sensor front-end). The vibration sensor illustrates the power and usefulness of the composability approach based on Boost Units and gives an insight into the design methodology wherein it seems convenient to generate this complex example.

```
1  //! Length data type.
2  typedef boost::units::quantity<boost::units::si::length> length_type;
3  //! Electrical potential data type.
4  typedef boost::units::quantity<boost::units::si::electric_potential>
       electrical_potential_type;
5  //! Time quantity type.
6  typedef boost::units::quantity<boost::units::si::time> time_type;
7  //! Length/Time data type.
8  typedef boost::units::divide_typeof_helper<length_type, time_type>::type
       length_per_time_type;
9  //! Electrical potential * Time/Length.
10 typedef boost::units::divide_typeof_helper<electrical_potential_type,
       length_per_time_type>::type factor_type;
11 //! Frequency data type.
12 typedef boost::units::quantity<boost::units::si::frequency> frequency_type;
13
14 //! Scale unit for micrometer.
15 typedef boost::units::make_scaled_unit< boost::units::si::length,
       boost::units::scale<10, boost::units::static_rational<-6> > >::type micrometer;
16 //! Micrometer data type.
17 typedef boost::units::quantity<micrometer> micrometer_type;
```

Listing 5.1: Definition of physical types

The quantity data types defined in the Listing 5.1 demonstrate how Boost::Units makes it possible to express semantic information regarding the dimensions and units of data handled within a model. We can use these definitions to give a physical meaning to our vibration sensor, as illustrated in Figure 5.4. The system is dynamic (depends on time in seconds), operates at a specific frequency (in Hertz) and involves mechanical displacement (in meters) of a seismic mass at a specific velocity (in meters/second). The transducer translates this displacement into an electrical potential (measured in Volts).

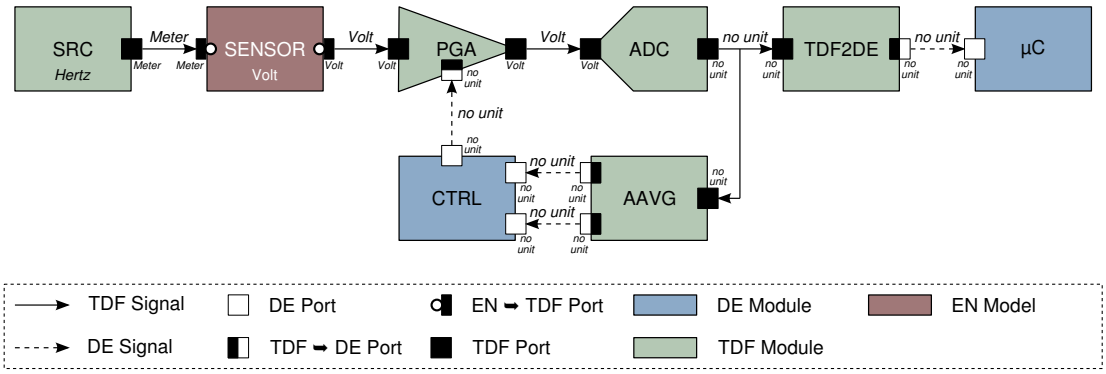

Figure 5.4: Running Example: Improved with units measures

The collaborative definition of this file enforces the use of common grounds to characterize ports, interfaces, physical quantities and scale factors. These quantities are used to define interfaces of components, as well as inner behavior description, with physical equations.

5.3.2 Clustering Sub-phase

The clustering algorithm represents the second sub-phase of the elaboration process (Figure 5.3, *sub-phase II*). Its purpose is to explore the whole system to collect information and generate appropriate simulation data structures by creating a domain-based, hierarchical view of the system. In practice, the clustering sub-phase aims to analyze the top-level system model and its associated sub-models in order to identify a finite set of interacting, homogeneous regions in the design, called *clusters*. Homogeneous regions represent part of the design described using only one MoC. In consequence, a cluster is associated with a single MoC.

Furthermore, the clustering sub-phase aims to organize these cluster as a hierarchy and identify which MoC is associated with each cluster. This hierarchical organization also enables us to highlight the associated master MoC for each MoC. As a result, the dependencies between the underlying MoCs are clearly established, based on the previously detailed master-slave semantics.

Performing the clustering sub-phase with respect to the master-slave semantics allows for the creation of an acyclic graph, representing the cluster hierarchy as a tree of clusters, where each slave cluster may only be encapsulated in one master cluster (only one immediate higher node in the tree). Conversely, a cluster can encapsulate several sub-clusters (a node can have several immediate lower nodes in the tree). The master-slave semantics dictate that, in a single branch of the cluster tree, a MoC associated with a cluster cannot appear twice. As SystemC MDVP is built upon the DE simulation kernel of SystemC, the root of this cluster tree is always a cluster associated with the DE MoC.

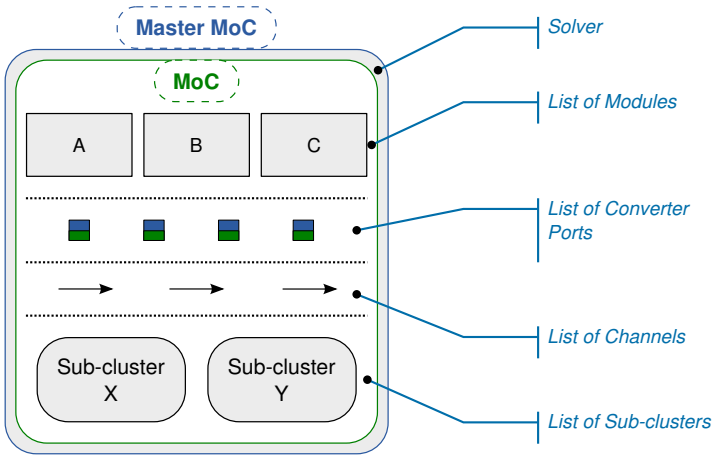

Figure 5.5: Representation of a Cluster

Figure 5.5 represents the structure of a cluster. A cluster stores information to identify the Model of Computation it abstracts and also the MoC which is its master.

Clusters are built with respect to the connectivity of the system provided by the designer, the idea being that components associated with the same MoC and connected together directly or through slave components, are encapsulated into a single object. A cluster organizes these components, storing modules and channels in dedicated lists. Regarding the ports, a cluster only

registers ports at the border of its MoC, i.e. it only stores converter ports. Since a cluster can also encapsulate clusters associated with other MoCs, provided all the master-slave semantics are respected within the whole hierarchy, a cluster keeps a list of the sub-clusters that have been encapsulated. Finally, a cluster also stores the solver that is in charge of computing of the components contained in the cluster.

Encapsulated clusters, which represent the abstraction of a slave MoC, are seen as components of the master MoC and, therefore, are encapsulated into the master's cluster in the same way as primitive modules. A cluster is complete when no more components (modules or sub-clusters) can be added to it by connection or instantiation. This means that all the connected modules and slave modules of the MoC associated with this cluster have been analyzed and organized so as to represent the subtree of the current cluster. Clusters, and their organization as a hierarchy, constitute the backbone of the SystemC MDVP framework.

The clustering mechanism is very similar to the Shift-Reduce process in LR grammars [73]. Since the module (respectively the sub-cluster) encountered during the exploration of the system hierarchy belongs (respectively integrates itself) to the MoC in the process of abstraction, a shift operation is performed. When the cluster is complete, a reduce operation is accomplished with the aim of integrating the subtree as if it were a standard module of the higher hierarchical node.

To facilitate the generation of this tree, all SystemC MDVP modules derive from the aforementioned generic C++ class `scm_moc_if` and a list of all the primitive modules instantiated in the system is built. Based on the content of this list, a set of primitive clusters is created, with each primitive module being encapsulated into its own cluster. This set of primitive clusters represents the initial set of clusters to be analyzed (hereafter referred to as set_cl). Two further structures are needed for port attributes, one indicating the "visited" status of a port and the other defining its type (regular or converter). Algorithm 5.1 details the recursive clustering function.

The `contains_cluster()` algorithm (Algorithm 5.1, line 9) is used to break the recursion, as it avoids re-processing clusters that have already been dealt with. If the analyzed cluster is eligible for processing, it is removed from the set of clusters (Algorithm 5.1, line 12). Each port of the cluster currently being processed is considered as a starting port to find the boundaries of the cluster being built. Each processed port is considered visited to avoid endless loops (Algorithm 5.1, lines 15-18). If the port being processed happens to be a converter port (Algorithm 5.1, line 19), the master cluster to which the port belongs to is compared to the current master of the cluster new_cl (Algorithm 5.1, line 21). A failing comparison means that the cluster new_cl has two different master MoCs, thereby suggesting that a clustering error has occurred as the key-point of the master-slave semantic rules is not being respected. Should such a situation arise, the elaboration phase will be aborted and the user provided with relevant information concerning the malformation of the design he is attempting to imulate, otherwise the valid converter port is added to the cluster new_cl.

If the port is not a converter port, a depth-first traversal is performed, i.e., all the ports

```
1  structure Cluster
2      String moc;
3      String master_moc;
4      List<Module> moc_ifs;
5      List<Cluster> cls;
6      Solver s;
7  end
8  Function process_cluster(set_cl, cl, new_cl)
       Data: set_cl, Set of Cluster to be analyzed.
       Data: cl, Cluster to be analyzed.
       Data: new_cl, Cluster to be built.
9      if not set_cl.contains_cluster(cl) then
10         return;
11     else
12         set_cl.remove_cluster(cl);
13     end if
14     foreach port p in the port list of cl do
15         if is_visited(p) then
16             continue;
17         end if
18         set_visited(p);
19         if is_converter_port(p) then
20             master = get_master(p);
21             check_master(new_cl, master);
22             set_master(new_cl, master);
23             new_cl.add_port(p);
24             continue;
25         end if
26         foreach port p_p connected to p do
27             if is_visited(p_p) then
28                 continue;
29             end if
30             set_visited(p_p);
31             next_cl = get_cluster_from_port(p_p);
32             if is_converter_port(p_p) then
33                 sub_cl = create_cluster(next_cl);
34                 set_cl.add_cluster(sub_cl);
35                 reset_attributes(p_p);
36             else
37                 process_cluster(next_cl, new_cl);
38                 continue;
39             end if
40         end foreach
41     end foreach
42     new_cl.add_sub_cluster(cl);
43 end
```

Algorithm 5.1: Recursive clustering algorithm.

connected to p are analyzed in turn. It should be noted that the clustering algorithm does not involve the communication channel. Since SystemC MDVP ports provide mechanisms allowing the set of ports, connected to the same channel, to be accessed directly, they can be completely bypassed in the algorithm.

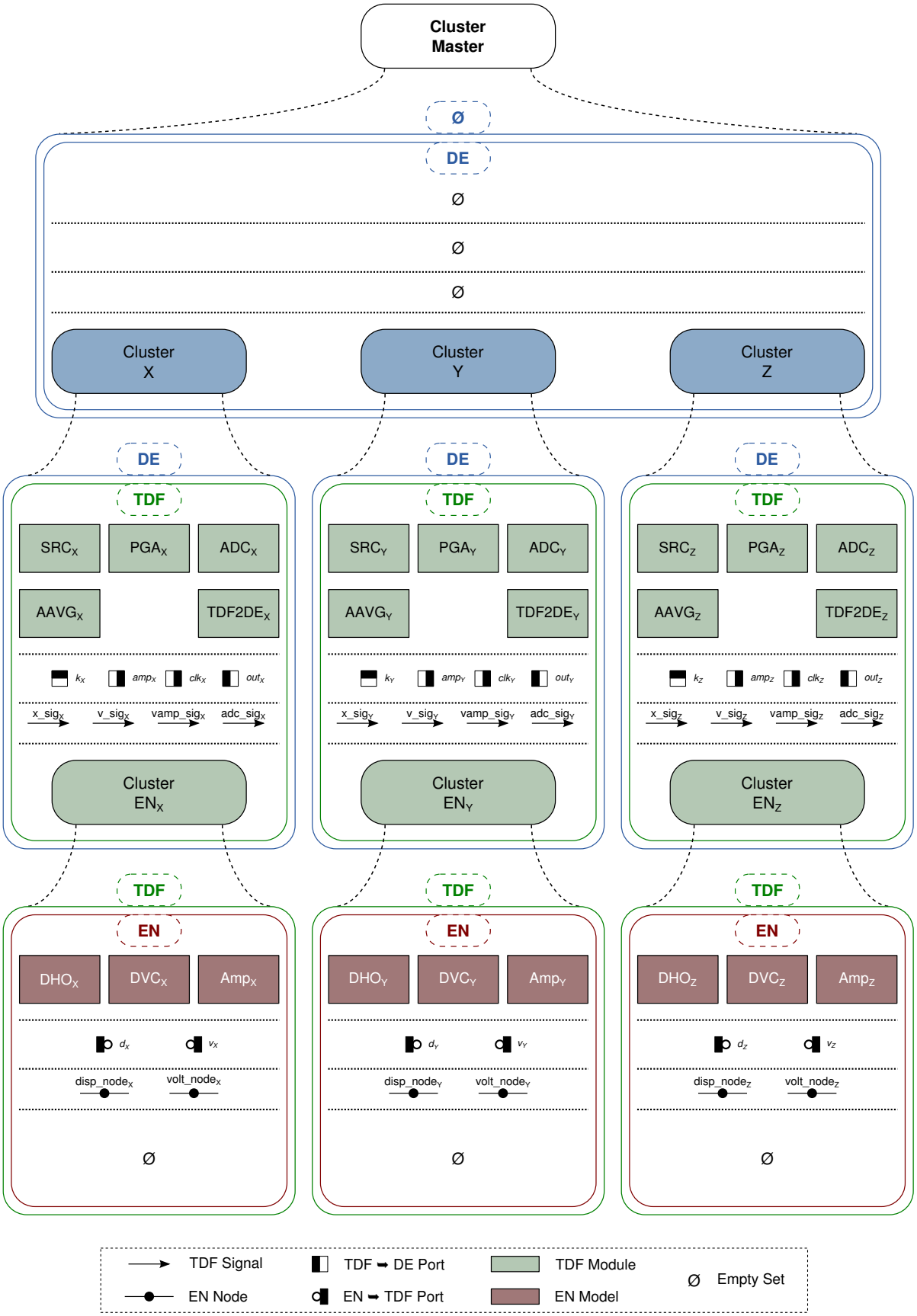

Figure 5.6: Running Example: Cluster Tree associated with the vibration sensor

The process performed on each connected port is similar to the previous one (Algorithm 5.1, lines 27-30). This time, the traversal is used to identify the boundaries (converter port) of another slave MoC. For that purpose, a new empty, cluster is created and the `process_cluster()` algorithm is applied to it. When the sub-cluster built is complete (Algorithm 5.1, line 33), it must be added into *set_cl* (Algorithm 5.1, line 34) to be processed later. To be encapsulated correctly, it must be considered as a component of the current MoC. The purpose of the `reset_attributes()` function (Algorithm 5.1, line 35) is to reset the ports' traversal attributes to false and re-trigger the analysis of the converter port, which has to be considered, this time, as a regular port of the master MoC. In the event that the port does not belong to a boundary, a recursive call is made to analyze the cluster corresponding to this port. When the exploration of a cluster is complete, it is added, as a sub-cluster, to the new cluster being created (Algorithm 5.1, line 42).

Applying the clustering algorithm to our running example, presented in Figure 4.8 as an assembly of MoCs, generates the hierarchical tree of clusters seen in Figure 5.6. In the later illustration, the representation of a cluster corresponds to the one provided in Figure 5.5. The root of the tree represents the encapsulation into SystemC through the DE MoC. It is important to note that the cluster at the root of the tree, referred to as *Cluster Master*, does not contain primitive modules, ports or signals, nor does it define a master. It contains only sub-clusters communicating with the DE MoC.

`Cluster X` represents the MoC TDF and has the DE MoC as a master. It encapsulates five primitive modules (SRC_X, PGA_X, ADC_X, $AAVG_X$ and $TDF2DE_X$), four converter ports (k_X, amp_X, clk_X and out_X) and four signals (x_sig_X, v_sig_X, $vamp\_sig_X$ and adc_sig_X) associated with the TDF MoC. `Cluster X` also contains one sub-cluster (`Cluster` EN_X) associated with EN MoC. From the `Cluster X` point-of-view, `Cluster` EN_X appears as a TDF-compliant primitive in accordance with the master-slave semantics.

`Cluster` EN_X represents the MoC EN and has the TDF MoC as a master. It encapsulates three primitive modules (DHO_X, DVC_X and Amp_X), two converter ports (d_X and v_X) and two nodes ($disp\_node_X$ and $volt\_node_X$ associated with the EN MoC. `Cluster` EN_X does not contain any sub-cluster; it represents a leaf in the hierarchical tree.

The branch of the hierarchy described with the `Clusters X` and EN_X represent the model of the *Sensor X*. The branches beginning with the `Cluster Y` and the `Cluster Z` respectively represent the model associated with the *Sensor Y* and the *Sensor Z*.

This representation infers that each node respects the semantics of the immediate parent node. Finally, this leads to a simulated system, ruled by SystemC semantics, which meets all the requirements for digital-centric systems.

5.3.3 Solver Instantiation Sub-phase

The solver instantiation represents the third sub-phase of the elaboration process. It takes advantage of the cluster hierarchical tree built in the previous sub-phase, especially the information about the *MoC* and the *master MoC* associated with each cluster.

The purpose of this sub-phase is to finalize the creation of internal data structures required to support the simulation semantics, i.e., to instantiate the solvers at the borders of MoCs (Figure 5.3, *sub-phase III*).

The principle behind this sub-phase relies on the use of Creational Design Patterns, a set of Design Patterns which aim to organize and specify the method used to construct object in a system. In particular, we chose to use the Prototype Design Pattern [74]. This pattern defines the way in which an object is created by means of prototypical instance and how to create new objects by copying this prototype through a *clone* process. Applied to our framework, this means that we have an instance of a solver and, when it comes to the instantiation of this solver, this can be achieved simply by cloning the existing one.

This approach requires that all solvers that will be used to perform the simulation of the system be known and already instantiated, which perfectly fits our approach. Indeed, within the master-slave semantics, the available interactions between MoCs are statically defined and the definition of the solvers is based on these available interactions.

In order to store the prototype of the available solvers, we define a *dictionary* containing all the instances of solvers. This dictionary is implemented by means of a `map` structure which represents the data by the association of a *key* and a *value*, usually represented as one <*key, value*>. A specific value is accessed by providing a specific key. In our approach, a value represents an instance of a specific solver. Since our solvers are defined depending on the master-slave semantics, there is a specific solver for each existing pair of <*slave, master*>. The key to address our map of solvers is, therefore, defined by the association of two pieces of information: the *MoC* and the *master MoC*. This Prototype Design Pattern is provided through a factory which stores the dictionary of prototypes and provides the functionalities required to manipulate this dictionary. All that this approach relies on is the class `scm_moc_info` previously described. This class stores information specific to a MoC and provide an easy access to them. As such, we can define a small dictionary, dedicated to a MoC, within each `scm_moc_info` representing a MoC. The global dictionary, handled by the factory, is built through the aggregation of these small dictionaries.

Table 5.1: Dictionary of solvers available for the clusters hierarchy shown in Figure 5.6.

<X, Y>	**Prototype to Clone**
<TDF, DE>	`scm_tdf::scm_de_solver`
<EN, TDF>	`scm_en::scm_tdf_solver`

Table 5.1 represents the dictionary of solvers associated with the cluster hierarchy of the running example of the vibration sensor shown in Figure 5.6. From this dictionary we retrieve the

pairs *MoC, master MoC* brought out during the Clustering sub-phase. The triple represented by *(<MoC x, master MoC y>, solver xy)* defines the *solver xy* which has to be instantiated (cloned) in order to execute the *MoC x* within the context of the *master MoC y*. In our running example, we find the triple *(<TDF, DE>,* `scm_tdf::scm_de_solver`*)* which defines the solver that allows TDF to run under the DE MoC. Similarly, we also find the triple *(<EN, TDF>,* `scm_en::scm_tdf_solver`*)* which defines the solver allowing EN to run under the control of the TDF MoC.

```
1  Function instantiate_solver(cl)
       Data: Cluster cl.
       Result: Solver s instantiated for cl.
2      moc_ifs = cl.get_moc_ifs();
3      foreach sub_cl in the clusters list of cl do
4          m = instantiate_solver(sub_cl);
5          moc_ifs.add_moc_if(m);
6      end foreach
7      moc = cl.get_moc();
8      master_moc = cl.get_master_moc();
9      if master_moc == NULL then
10         return NULL;
11     end if
12     s = find_and_clone(moc,master_moc,moc_ifs);
13     cl.set_solver(s);
14     return s;
15 end
```

Algorithm 5.2: Recursive solver instantiation algorithm.

The Algorithm 5.2 shows the recursive, bottom-up function proposed to perform the automatic instantiation of solvers. The list of modules associated with a cluster is stored in a list *moc_ifs* (Algorithm 5.2, line 2). The function `instantiate_solver()` is recursively called on each sub-cluster. The solver created for each sub-cluster is then added into the list *moc_ifs* (Algorithm 5.2, line 5). Eventually, this list will contain all the modules and sub-solvers contained in a cluster. Once we have gone through all the sub-clusters of a cluster (or when no sub-cluster exists), we obtain the information concerning the MoC and the master MoC of the cluster (Algorithm 5.2, lines 7-8). Thanks to this accumulated information, we possess the specific key to access the specific solver associated with this cluster (Algorithm 5.2, line 12). The function `find_and_clone()` is provided by the factory and searches the dictionary for the value associated with the key *<moc, master_moc>* (provided as parameters). When the prototype solver is identified, it is cloned and the new cloned solver is returned. `find_and_clone()` takes a third argument which represents the list of objects that the solver will have to handle. Given that this list comprises modules and, potentially, sub-solvers, it is abstracted through the `scm_moc_if` class. This information is duly provided to the solver during its cloning process.

As illustrated on Figure 5.7, applying the Algorithm 5.2 to the cluster hierarchy of the running example enriches each cluster structure by providing information about the solver in charge of the computation. Moreover, this sub-phase also allows for a new abstraction of the model relying on the solver point-of-view. On account of this, we can represent the running example from the

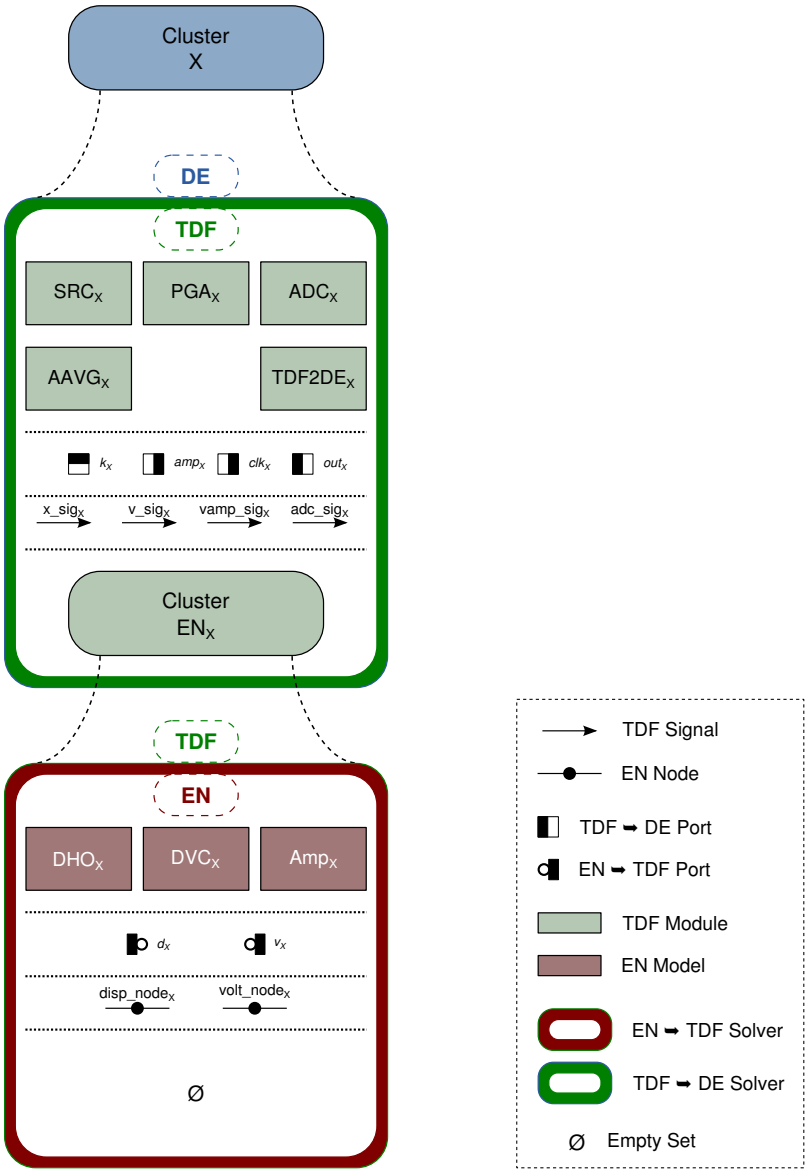

Figure 5.7: Running Example: Branch of the cluster Tree with instantiated solver

solvers point of view as illustrated in Figure 5.8.

We see that, from the point-of-view of SystemC, all the models described with MoCs belonging to SystemC MDVP appear as a classical SystemC component; the sensor X is only seen through its solver associated with DE. Similarly, from the point-of-view of TDF, models described with the EN MoC appear as a classical TDF component thanks to the EN solver associated with TDF. We also note that the components belonging to SystemC are not seen by TDF. The same applies to EN which only perceives its components.

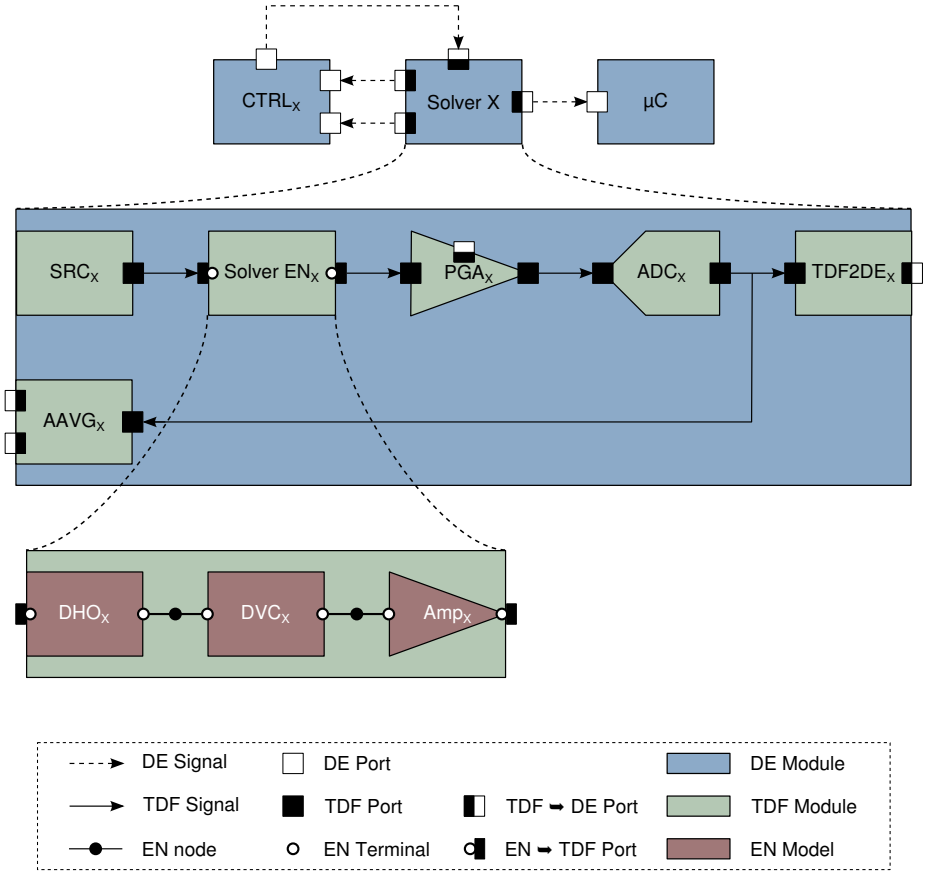

Figure 5.8: Running Example: solver abstraction of the vibration sensor

5.3.4 Basic Behavior Block Elaboration Sub-phase

The previous sub-phases in the elaboration process were dedicated to the exploration of the system, with the aim of collecting information and automatically creating the structures required to perform the simulation. The remaining phases are dedicated to the elaboration of individual components of the model: modules and solver (Figure 5.3, *phase IV*), ports and channels (Figure 5.3, *phase V*).

The generic elaboration of modules and solvers relies on the cluster tree built during the Clustering sub-phase and enriched in the previous sub-phase. The fact that the root of the hierarchical tree represents, by definition, the DE MoC, indicates that all the clusters encapsulated in the root node of the tree represent MoCs that are direct slaves of the DE MoC (SystemC). The DE MoC interface is defined within the SystemC MDVP kernel through the class `scm_de::scm_moc_if`. The master-slave semantics, and the structure previously presented, allow the elaboration of the components to be performed in cascade from the root node of the tree. A bottom-up elaboration is required to ensure that the subtree of a branch of the cluster hierarchy, a slave, is already elaborated before the master begins its elaboration.

Figure 5.9 illustrates the minimal class implementation, in accordance with our approach, based on the running example of the vibration sensor. We see that, in order to communicate with another MoC (a master), a MoC must implement its interface; this mechanism allows the

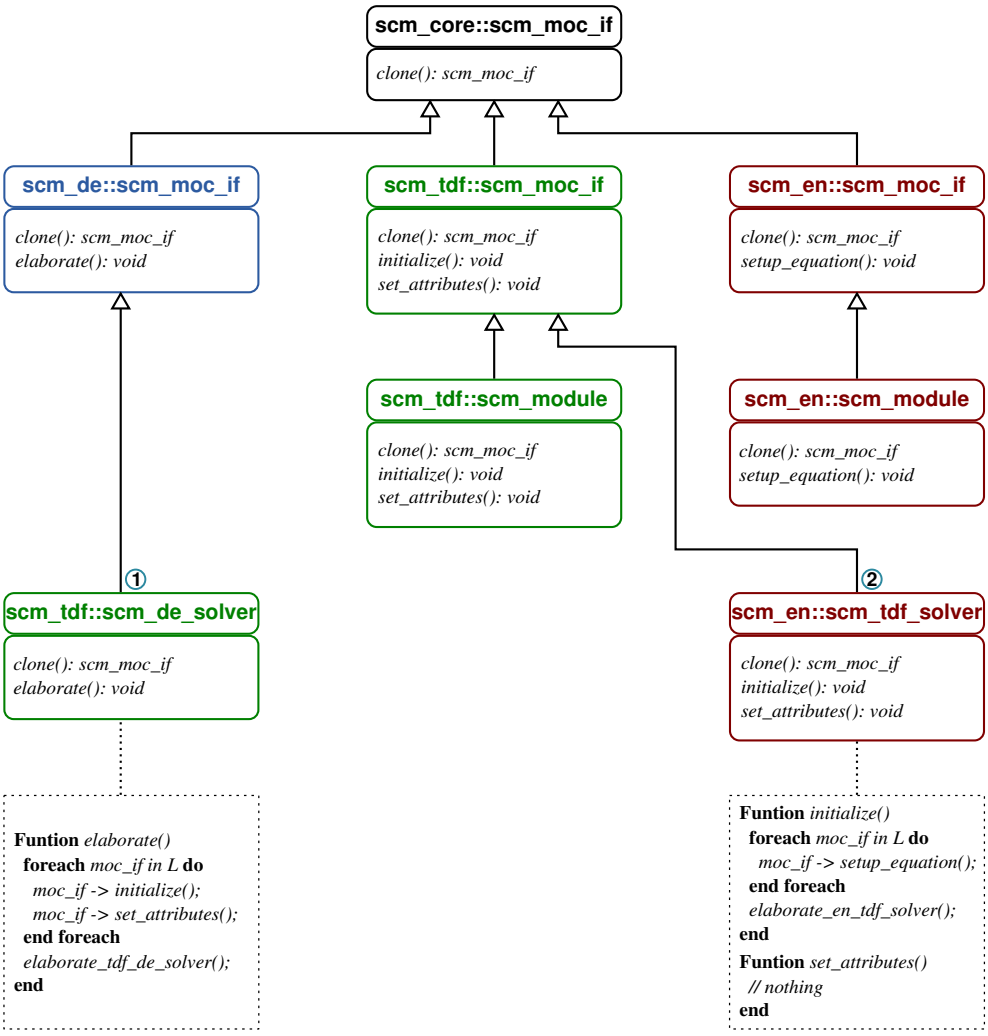

Figure 5.9: Running Example: Minimal class hierarchy to perform the elaboration of all the primitive blocks.

slave MoC to be handled by the master. Undeniably, the TDF MoC seeks to communicate with the DE MoC and, therefore, provides `scm_tdf::scm_de_solver` ① which allows TDF to run under DE. In the same manner, the EN MoC provides `scm_en::scm_tdf_solver` ② in order to communicate with the TDF MoC. These two classes correspond to the solvers available in the system, previously introduced in Table 5.1.

```
1 Function elaborate_moc_interfaces(cluster_root)
      Data: Cluster cluster_root.
      Result: void.
2     foreach sub_cl in the clusters list of cluster_root do
3         cl_moc_if=cluster_root.get_moc_interface();
4         moc_if=static_cast<scm_de::scm_moc_if>(cl_moc_if);
5         moc_if.elaborate();
6     end foreach
7 end
```

Algorithm 5.3: Elaboration of MoC interfaces algorithm.

Algorithm 5.3 illustrates how this organization and representation of data is used to perform a

generic elaboration. In order to elaborate and configure each module and solver for simulation, the Algorithm 5.3 relies on the manipulation of the `scm_moc_if` class.

The master-slave semantics oblige each MoC slave of DE to provide a MoC interface that complies with the DE MoC. This means that the immediate sub-clusters of the cluster root own a MoC interface that inherits from `scm_de::scm_moc_if`, enabling their abstraction by the MoC interface associated with DE (Algorithm 5.3, line 4). The `elaborate()` function (Algorithm 5.3, line 5) can be called on each MoC interface belonging to these sub-clusters and the call to the elaboration functions, specific to each MoC, is automatically performed in cascade thanks to the implementation mechanism detailed in Figure 5.9.

One might observe that the elaboration implementation of the modules and solvers belonging to a MoC is a specific, MoC-defined mechanism. `scm_tdf::scm_de_solver` will automatically call the functions `initialize()` and `set_attributes()` when its function `elaborate()` is called. Similarly, `scm_en::scm_tdf_solver` will automatically call the function `setup_equation()` when its function `initialize()` is called. We also see that, for the EN MoC, the function `set_attributes()` is not used though it still has to be defined in the MoC interface that complies with TDF. This elaboration of modules and solvers is, in fact, a bottom-up elaboration: the modules and sub-solvers handled by a solver are elaborated before the solver performs its own elaboration.

5.3.5 Port and Channel Elaboration Sub-phase

As is the case for the generic elaboration of modules and solvers, the generic elaboration of ports and channels relies on the cluster hierarchy defined during the Clustering sub-phase. Algorithm 5.4 illustrates how the cluster tree is used to perform the generic elaboration of ports and channels within SystemC MDVP. In order to elaborate and configure these components, Algorithm 5.4 depends on the manipulation of the base classes defined with the purpose of abstracting a MoC. The ports and channels are elaborated using the base classes associated to the port (`scm_port_base`), the interface (`scm_interface`) and the channel (`scm_prim_channel`).

The `elaborate_ports_and_channels()` function is recursively called on every cluster in the cluster hierarchy (Algorithm 5.4, line 3). For each cluster, the list of MoC interfaces which contain modules and potentially sub-solvers (Algorithm 5.4, line 5) gives us access to the ports belonging to these components (Algorithm 5.4, line 7). We then go through each port and, if it is not already elaborated, we call its elaboration function (Algorithm 5.4, line 10) and set its elaborated status to *true* (Algorithm 5.4, line 11).

The scope of a cluster ends at its converter ports so the channels associated with these ports are left to be elaborated along with the nodes immediately above in the hierarchy. As such, for ports other than converter ports, access to their associated channels can be gained through their respective interfaces (Algorithm 5.4, line 13). Doing so is possible because each channel must

```
1  Function elaborate_ports_and_channels(cl)
       Data: Cluster cl.
       Result: void.
2      foreach sub_cl in the clusters list of cl do
3          elaborate_ports_and_channels(sub_cl);
4      end foreach
5      list_moc_ifs = cl.get_moc_interfaces();
6      foreach moc_if in list_moc_ifs do
7          list_ports = moc_if.get_mocif_ports();
8          foreach port in list_ports do
9              if !port.is_elaborated() then
10                 port.elaborate();
11                 port.set_elaborated();
12                 if !port.is_converter() then
13                     if=port.get_scm_interface();
14                     ch=dynamic_cast<scm_prim_channel>(if);
15                     if !ch.is_elaborated() then
16                         ch.elaborate();
17                         ch.set_elaborated();
18                     end if
19                 end if
20             end if
21         end foreach
22     end foreach
23 end
```

Algorithm 5.4: Elaboration of ports and channels algorithm.

inherit from an interface and a prim_channel, allowing for the realization of a *dynamic_cast* from an interface to a prim_channel (Algorithm 5.4, line 14). Ultimately, if the channel is not already elaborated, we call its elaboration function (Algorithm 5.4, line 16) and set is elaborated status to *true* (Algorithm 5.4, line 17).

This algorithm executes a deep-first traversal of the hierarchical tree of clusters. To that end, the elaboration is performed cluster by cluster, beginning with clusters representing *leaves* in the hierarchy and moving up the tree. Like the algorithm designed to elaborate modules and solvers, the aforestated algorithm respects a bottom-up elaboration.

5.3.6 Elaboration conclusion

The SystemC MDVP elaboration process presented above is a completely generic mechanism which allows us to abstract any Model of Computation. Our framework is, consequently, independent from the MoCs definition. This solution honors our commitment to designing a flexible virtual prototyping environment, the evolution of which is made entirely possible by its non-reliance on MoCs.

Our elaboration process can be resumed in four steps, as illustrated in Algorithm 5.5. These steps represent all of the sub-phases previously presented, with the exception of the composability

check which is performed at compile-time.

```
1 Function elaboration()
2 |   hierarchy_root = clustering();
3 |   instantiate_moc_interfaces(hierarchy_root);
4 |   elaborate_moc_interfaces(hierarchy_root);
5 |   elaborate_ports_and_channels(hierarchy_root);
6 end
```

Algorithm 5.5: Generic Elaboration Algorithm.

The constituent sub-phases of the elaboration process are executed sequentially. At the end of this process, all components involved in the model are instantiated, elaborated, and ready for the simulation process.

5.4 Simulation Phase

Moving on to the simulation phase, all of the resources required to perform the generic simulation are now available thanks to the work accomplished during the elaboration process. The system modeled is, therefore, ready to be simulated. SystemC MDVP framework is built on top of SystemC and, as such, can benefit from the regular simulation phase of the DE simulator kernel. This means that SystemC is the master of the SystemC MDVP framework and that our framework is obliged to respect SystemC semantics, hence, implying that SystemC governs the whole simulation. As a result, the simulation phase is mostly provided, and supported, by SystemC, meaning that our simulation phase mainly consists in setting up the infrastructure to enable the simulation.

The mechanisms involved in the simulation phase are presented in Figure 5.10. This figure resumes the Figure 3.3, described in Chapter 3, and enriches it in order to illustrate how the SystemC MDVP simulation phase extends SystemC to handle heterogeneity while still refraining from altering its kernel. We use the `start_of_simulation()` and the `end_of_simulation()` callbacks from SystemC to trigger the execution of our own simulation mechanism.

The simulation phase adheres to the hierarchical view of the system provided by the generic elaboration process. The SystemC MDVP simulation mechanism consists of two sub-phases, the first being the hierarchical initialization of the system. The second follows with the settings of the solvers communicating with DE so that they may be handled by SystemC.

5.4.1 Simulation Mechanism

We have already established that the elaboration phase provides a hierarchical view of the system by means of a cluster tree. Modeled on this tree, the elaboration process also affords a solver

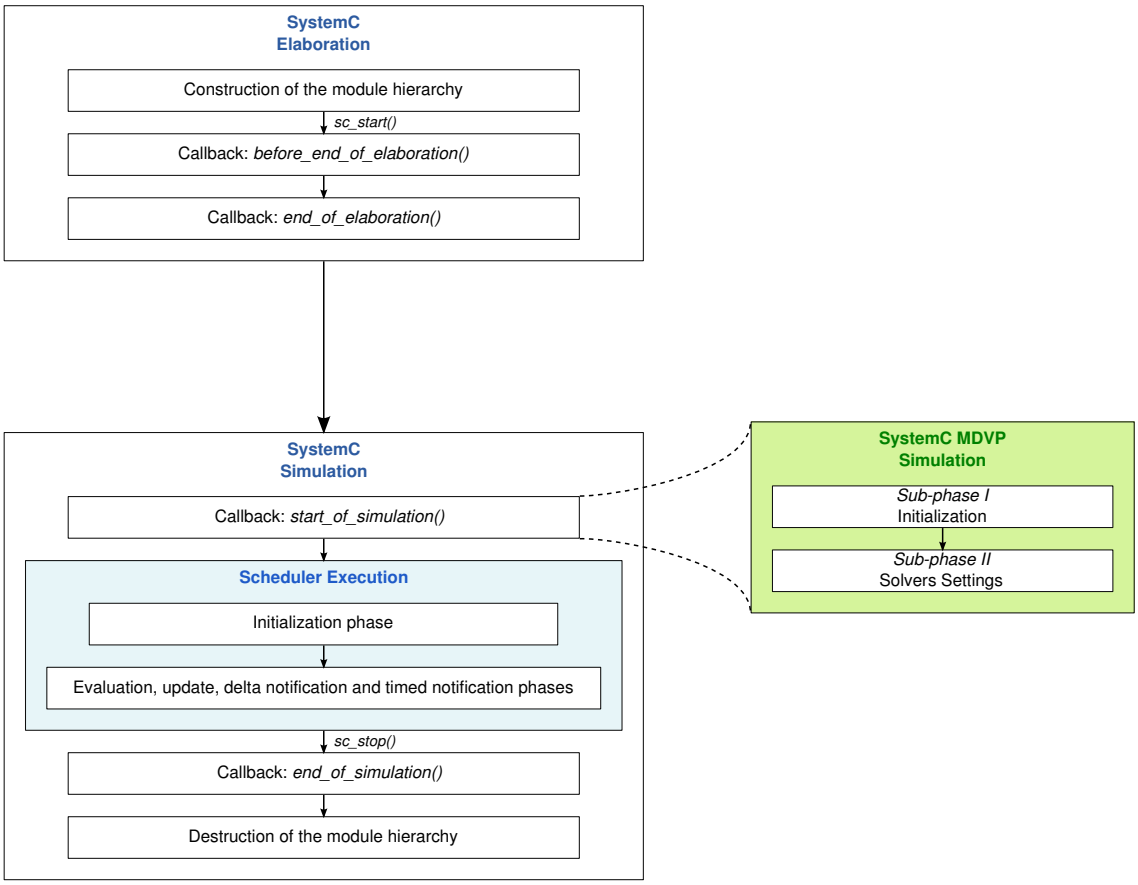

Figure 5.10: SystemC MDVP kernel simulation phases.

hierarchy. The first sub-phase, as noted in the introduction preceding this section, involves the initialisation of the system and is rather straightforward to achieve. This simply consists in exploiting the cluster tree structure to perform the system initialisation in cascade, as explained previously with reference to other elaborations, for instance, that of MoC interfaces.

Returning, again, to the items briefly described in the introduction of this section, it is worth re-iterating the importance of the SystemC DE kernel of simulation which provides and supports the simulation to a great extent. Every cluster encapsulated in the root of this tree communicates with DE and so provides an interface compatible with the DE MoC. The solvers contained in these clusters interact with SystemC by creating a dynamic process which allows a solver to integrate itself into the scheduler of SystemC. This way, the execution control of this solver is handled by the DE simulation kernel. In order to create this dynamic process, we use the functionalities provided by `sc_spawn` from SystemC.

This approach allows us to create a *SystemC thread* which will be integrated into the scheduling list of processes to execute within the SystemC kernel. This thread is in charge of the execution of the components associated with its MoC while respecting the constraints imposed by SystemC. This thread will trigger the execution of the potential sub-solvers associated with its slave MoCs. In the context of our framework the simulation relies on a top-down prefix-order traversal of the hierarchy tree.

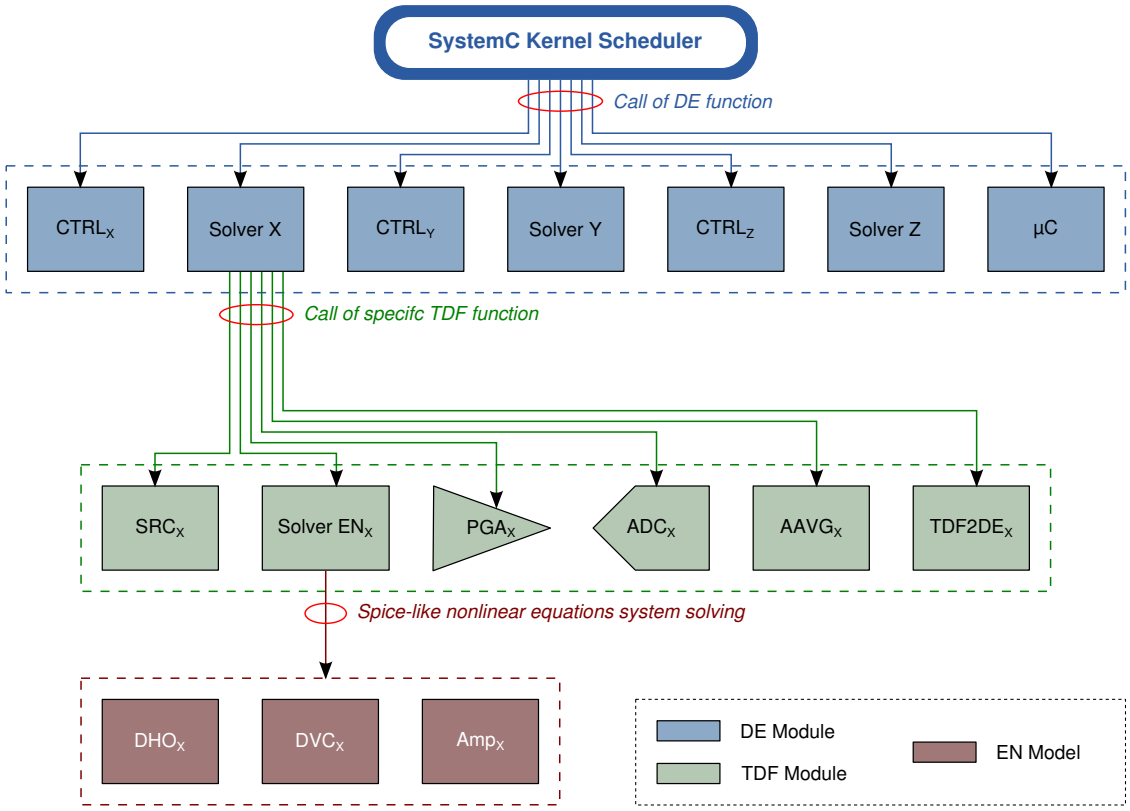

Figure 5.11: Running Example: simulation process.

Figure 5.11 illustrates the simulation process applied to the running example of the vibration sensor. SystemC provides its components, including our TDF solvers (`Solver X`, `Solver Y` and `Solver Z`), with a simulation context. The TDF solvers that are encapsulated at the root of the hierarchical tree are addressed by the simulation kernel of SystemC in the same way as SystemC modules. Based on this context, our solvers are able to perform a local simulation using their own semantics, as illustrated with the `Solver X` in Figure 5.11.

Once it has been called by the SystemC kernel through its interface with DE, the `Solver X` can trigger the execution of the components it is responsible for using the TDF MoC semantics and interface. The `Solver` EN_X, when triggered by the TDF solver through its interface with TDF, can, in turn, trigger the resolution of the equation system, built from its primitives, for the period of time specified by the temporal window offered by TDF.

In a more generic approach, SystemC triggers the execution of each branch of the hierarchical tree representing the system modeled. With respect to the master simulation context, each node is allowed to perform a local simulation using its associated solver. A node can generate a local simulation context and gives its own context to its slave. Each node can manage its own time scale, and advance at its own pace, with respect to the constraints imposed by its master. The results produced by a node follow the reverse path; the local results of a node are converted and integrated into its master node.

5.4.2 Simulation Opportunities

Bearing in mind that we want to provide a generic virtual prototyping environment, the aforementioned approach presents many opportunities regarding the simulation. A generic approach allows for a more flexible simulation mechanism. Since our approach is independent from the MoCs definition, and our simulation mechanism mainly relies on SystemC, the opportunities offered by our framework do not rely on the kernel implementation. Though they are not directly implemented within our kernel, these opportunities are made available by the infrastructure provided by our framework. Our framework allows for the implementation of solvers which take into account roll-back mechanisms or time scale executions. Our simulation process depends on the transmission of a simulation context by a solver to all the components it handles. Among the parameters, floor timestamps as well as horizon timestamps can be transmitted.

Taking this input information into consideration, a solver can perform local roll-back loops in order to provide a convergent solution. The solver needs to store the current state of the sub-system that it handles, prior to the commencement of the sub-system's simulation, for the time-slot provided by the higher hierarchical node. The solver must also be able to restore its previous state. With these two functionalities, the solver can perform local roll-back loops. If the solution diverges, it can start again and tune its internal parameters so as to reach convergence. Such an approach is illustrated in Figure 5.12 where an activity diagram of an algorithm that defines a roll back execution mechanism is described. Where activity diagrams are concerned, the solid black circle (*start*) represents the entry point of the algorithm with the second circle (*end*) indicating the exit point.

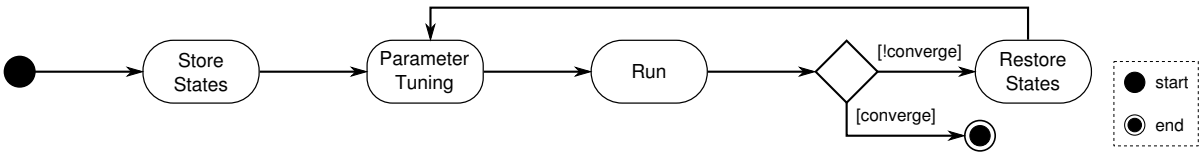

Figure 5.12: Activity Diagram of a roll back algorithm.

Similarly, a solver can store this information in order to postpone its execution. Due to the fact that the time scale between two hierarchical levels may differ, the time-slot provided to a lower hierarchical node might not be long enough for the solver to be able to produce a reliable solution. In such cases, a solver can accumulate several time-slots with the aim of realising a longer execution and, hence, come up with a relevant solution. Figure 5.13 presents an activity diagram describing an algorithm which defines a postponed execution mechanism, thus illustrating the above approach.

Another approach may be considered when the time-slot available for a lower hierarchical node is of a sufficient length to allow the associated solver to perform several executions. The solver will only transmit the expected result to its higher node once its attributed time-slot has been consumed. A demonstration of this approach can be found in Figure 5.14 which shows an activity diagram of an algorithm used to define a mechanism where the execution is performed multiple times until the point is reached where no more time is available.

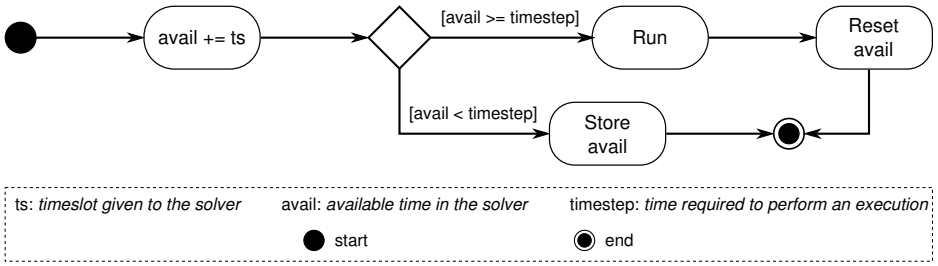

Figure 5.13: Activity Diagram of a postponed execution algorithm.

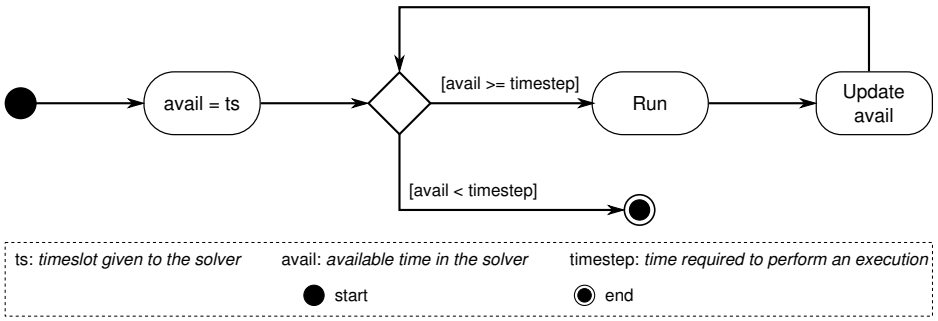

Figure 5.14: Activity Diagram of a multiple execution algorithm.

We presently described some of the opportunities that our virtual prototyping environment has to offer. Thanks to our generic, MoC-independent approach, one can implement a solving algorithm which follows its own semantics.

5.5 Conclusion

In this chapter, we introduced the implementation details of SystemC MDVP, our virtual prototyping environment, which supports the principles outlined in the previous chapter. We explained how SystemC MDVP, developed on top of SystemC, interacts with, and benefits from, this system-level modeling language; accordingly, we pointed out that the kernel phases of SystemC MDVP strictly mimic those of SystemC. The set of basic classes and algorithms which define the kernel, along with those required for sound functioning, were depicted in this chapter and the generic algorithms, which allow the elaboration and simulation phases to be performed in a completely generic way, were discussed.

As has been noted, we designed our framework with the objective being to meet the two principal criteria hereto established: it must be generic and it must not be dependent upon the Models of Computation definitions. We presently expanded on this point and also examined the main classes which guarantee such an independence by implementing an abstract representation of MoCs. This implementation relies on the functional abstraction of a MoC defined in Chapter 4. We abstracted the notion of behavior associated with a MoC by means of separated classes: one to represent the primitive blocks of a MoC (`scm_core::scm_module`), another representing the solving mechanism of a MoC (`scm_core::scm_solver`) and a third class which represents the

interface of a MoC (`scm_core::scm_moc_if`). The interface of a MoC is also used for the purpose of independently abstracting and manipulating primitive blocks or solvers.

Two classes were defined in order for us to abstract the channel representation within a MoC, the first designating a communication interface (`scm_core::scm_interface`, and the second, a communication channel (`scm_core::scm_prim_channel`). The same method was applied for the notion of composition: in this abstraction, a basic port (wherein generic information in relation to ports is re-grouped) is represented by the class `scm_core::scm_port_base` with `scm_core::scm_port<IF>` denoting the port itself. Converter ports, used at the border between two MoCs, are also represented here. A final class, `scm_core::scm_moc_info`, is defined so that generic information characterizing a MoC may be gathered and contained within a single object.

In addition to the classes defined to abstract a MoC, we introduced classes required for the proper functioning of the kernel. A small number of complementary classes are involved in handling parts of the kernel process. Two such classes are required to assure the construction of the cluster tree (which constitutes the backbone of our framework) - `scm_core::scm_cluster_node` and `scm_core::scm_cluster_creator` respectively, and another, `scm_core::scm_moc_interface_creator`, to manage the automatic instantiation of solvers. Having spoken of the aforementioned classes, we moved on to present the idea of a simulation context through the class `scm_core::scm_simcontext` which presides over the entire simulation from the SystemC MDVP viewpoint. It operates the MoCs via the aforementioned abstraction and contains the generic algorithms that allow for the simulation of heterogeneous system.

The work carried out by the kernel focusses on the elaboration process. Responsible for providing all the resources needed for it to be performed, it represents a crucial part of the simulation. During this phase, the generic algorithms responsible for implementing the principles underlying SystemC MDVP are defined. This process is articulated around five sub-phases which aim to prepare the simulator for simulation.

The composability sub-phase is performed at compile-time. Its purpose is to prevent design errors due to incorrect interfacing or incompatibility among the system components. This is achieved through the explicit integration of quantity data types which make it possible to define variables and interfaces enhanced with semantics information. These quantity types allow units and dimensions associated with a component to be expressed. This approach is supported by the use of the `Boost::Units` Library which defines all the units and dimensions indexed in the international system of units. Moreover, it supports all the basic mathematical operations between quantities. At compile-time, we can type-check the quantity and ensure that the interfacing of components is effectuated correctly without any overhead on the simulation; only the compilation time is impacted.

The aim of the clustering sub-phase is to explore the system thoroughly, collecting information and generating an appropriate simulation data structure through the medium of a domain-based, hierarchical view of the system created for this purpose. From the flat representation of the system

provided by SystemC, the clustering sub-phase builds a tree wherein each node represents a cluster, i.e. a homogeneous region in the design portraying a subset of the model which is described using a single MoC. A cluster is, therefore, associated with one individual MoC. This approach capitalises on the master-slave semantics, which define the interactions amongst MoCs, in order to establish the hierarchical organization, demonstrated by the cluster tree, which highlights the dependencies between the underlying MoCs. The root of this hierarchy, by construction, always denotes SystemC and its associated DE MoC. This data structure embodies the essence of the simulator; it constitutes the kernel's internal representation of the system modeled.

The following sub-phase defines the generic, and automatic, instantiation of solvers. By exploiting the cluster representation of the system, wherein, as we know, the emphasis rests on the master-slave relations that link the MoCs, this sub-phase automatically ascertains which particular solver should be instantiated at a given border. The remaining sub-phases of the elaboration process are dedicated to the elaboration of each component of the system. Once again, the cluster hierarchy is used to support these sub-phases. The elaboration of the components follows a bottom-up approach, starting from the leaves of the cluster tree and progressing along the branches.

To summarize, the elaboration process of SystemC MDVP is a completely generic mechanism, uninfluenced by the MoCs' definition. We can conclude, from this, that we have achieved our objective of designing a virtual prototyping environment which is not only flexible, but also capable of evolving since it is not reliant on MoCs.

Predominantly sustained by SystemC, the simulation phase of the SystemC MDVP kernel is quite straightforward. Our framework is, of course, defined on top of SystemC following our own master-slave semantics thus implying that SystemC is the master of our simulator, governing the whole simulation as a consequence. The main idea is to create, associated with SystemC MDVP solvers, SystemC threads that will be integrated within the SystemC kernel scheduler in order to be executed. These threads are created solely for the solvers positioned at the root of the cluster hierarchy, i.e. those communicating with DE. As is the case for prior phases, the cluster hierarchy constitutes a fundamental element of this mechanism. Even after the threads have been created and integrated into SystemC, the cluster tree continues to play a central role in the execution of the simulation. The execution of a branch of the tree is performed in cascade, from the root of the tree (the SystemC threads) to the leaves.

Echoing the defining characteristics of the elaboration phases (and its constituent sub-phases) our simulation phase is generic and brings flexibility to our virtual prototyping environment. We recently outlined the numerous simulation opportunities generated by our implementation of the simulation mechanism. This implementation provides the means to define solvers which can perform roll-back loop or, indeed, time scale execution, where the solver is able to postpone its execution or execute itself several times in a row. All of this is possible thanks to our generic, MoC-independent approach.

Chapter

6 *SystemC MDVP: Monitoring*

Contents

6.1 Introduction 82
6.2 Principles 83
6.2.1 SystemC Monitoring 84
6.2.2 SystemC MDVP Monitoring 86
6.3 Monitoring Mechanism 87
6.3.1 Monitor Handler Instantiation 88
6.3.2 Monitor Slot Instantiation 89
6.3.3 Initialization 91
6.3.4 kernel Routine 94
6.3.5 End of Simulation 96
6.4 Conclusion 96

6.1 Introduction

Several notions spring to mind when the subject of monitoring is evoked. There is, of course, the tracing mechanism (primarily used for de-bugging purposes), which consists in the logging of information during the simulation, not to mention the profiling mechanism which is responsible for analyzing specific values within a system. An approach based on such mechanisms may be adopted to carry out performance tests or detect threshold crossing.

We believe these functionalities should be developed simultaneously. Indeed, the same mechanisms are involved in performing tracing, profiling and other, similar, processes, only the outcome is different. All of the afore-stated functions require a mechanism to probe the simulated system and collect pertinent information. In this thesis, therefore, we consider monitoring as a mechanism which aims at observing and recording information about a system regardless of the eventual outcome.

Let us now consider the matter of developing a monitoring mechanism for the SystemC MDVP framework. Should the approach which consists in using the existing solution for the SystemC models be adopted, it would lead to the development of a solution that employs different mechanisms to access data in the system. In our opinion, however, an approach implicating a unified access to the information necessary for the monitoring of the system would be more appropriate. As such, our monitoring mechanism must be capable of handling SystemC MDVP components as well as those of SystemC. The challenge lies in generating this ability to handle both sets of components while still remaining generic and providing a convenient interface for the end-user.

Section 6.2 introduces the monitoring principles that support a multi-disciplinary approach to monitoring required within SystemC MDVP. We begin by providing an overview of the classes and data structures required to perform the monitoring of heterogeneous systems. We then detail the principles that allow us to perform the monitoring of SystemC components and continue with those involved in the monitoring of components belonging to SystemC MDVP.

The monitoring mechanism is introduced in Section 6.3, as is the way in which it integrates into the SystemC MDVP environment. Herein, we equally reveal the different steps that are added to the simulation process previously described in Chapter 5 and present the generic algorithms which perform the monitoring during the simulation of heterogeneous systems.

The present chapter concludes with Section 6.4 which comprises an overview of the monitoring of multi-disciplinary systems.

6.2 Principles

Its goal being to support the scalability and flexibility of our framework, the monitoring mechanism must remain generic and independent from the MoCs (as well as the disciplines they represent). It is, however, unlikely that it would be able to maintain its independence while simultaneously collecting relevant, and specific, data. We must, therefore, adopt the same approach as we did for the handling of the Models of Computation. We must define a generic abstraction within the SystemC MDVP's kernel and enable the specification of this abstraction within each MoC. The set of classes that support this approach is illustrated in Figure 6.1.

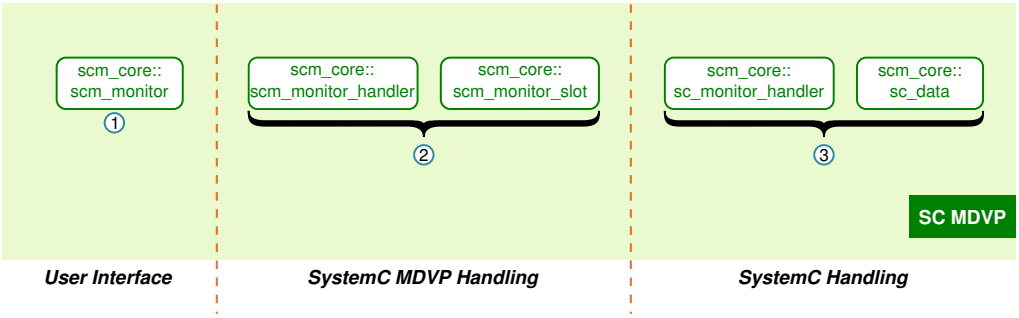

Figure 6.1: SystemC MDVP Monitor classes

The monitoring only requires a couple of classes to function properly. The class `scm_core::scm_monitor` ① represents the user interface and the object in charge of the monitoring mechanism within the simulator. It defines the interface that the user will manipulate to specify the signal he wants to monitor. This is the only interaction that the user will have with the monitoring mechanism. Within the kernel, this class performs the monitoring algorithm that will trigger the probing of the system and aggregates the relevant and specific information retrieved from the different specified sources.

Additionally, we have two classes to handle the SystemC MDVP components: `scm_core::scm_monitor_handler` and `scm_core::scm_monitor_slot` ②. These two classes are in charge of probing the system to gather the relevant information, before formating the collected data so that it respects the format expected by the `scm_core::scm_monitor`. `scm_core::scm_monitor_slot` describes the object that handles the signals to be monitored. For each MoC a specific *monitor slot* must be defined in order to handle the specificities related to the MoC. Taking into account the representation of the system within SystemC MDVP, this approach means that a dedicated monitor slot is created for each cluster in the hierarchy. The SystemC MDVP *monitor handler* (`scm_core::scm_monitor_handler`) defines the object manager that controls the monitoring of a whole branch of the cluster hierarchy. It stores all the monitor slots associated with the clusters belonging to the branch it describes.

Similarly, they are two classes, `scm_core::sc_monitor_handler` and `scm_core::sc_data` ③, to handle the SystemC components. The first class represents the object that manages all the SystemC components which need to be monitored while the other is defined so as to handle the

different data types supported by SystemC.

The presented abstraction of the monitoring within the SystemC MDVP kernel allows for the representation of the monitoring data structures as a monitoring tree which modeled the hierarchy of the clusters built during the clustering phase of the elaboration process.

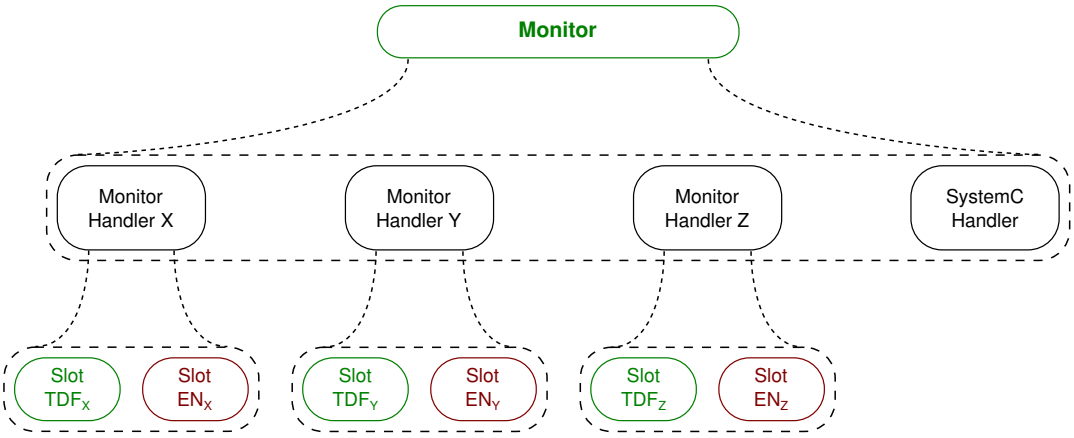

Figure 6.2: Running Example: Monitoring tree associated with the vibration sensor.

Figure 6.2 depicts the monitoring tree associated with the running example of the vibration sensor. We see a monitor handler for each branch of the cluster tree describing this system (a monitor handler for each sensor). Each monitor handler owns a monitor slot to handle the signals associated with the sub-model described with the EN MoC, and another one to handle that of the sub-model described with the TDF MoC. For the Sensor X, we have the `Monitor Handler X` that manages the monitor slots *Slot* TDF_X and *Slot* EN_X which are in charge of the TDF signals and the EN nodes respectively. The SystemC components are handled by the `SystemC handler`. The root of the monitoring tree is the `scm_monitor` which controls and, indeed, triggers the monitoring process from the SystemC MDVP kernel.

This approach allows the kernel to trigger the monitoring functionality within each MoC while remaining independent from them. Moreover, the definition within each MoC of the specificities of the data handled allows for the collection of relevant and specific information.

6.2.1 SystemC Monitoring

We believe that a unified access to the data throughout the whole simulated system constitutes the best approach for a monitoring mechanism. In consequence, we are compelled to manipulate SystemC communication components, i.e. SystemC's channels, though this is not without its difficulties since we do not want to alter the simulation kernel. In order to monitor SystemC components, we have to fully understand the underlying mechanisms within the simulation kernel. Fortunately, SystemC provides some mechanisms that can help us to achieve the monitoring of these components.

When a writing on a SystemC signal occurs, the signal sends an event to notify the writing.

This event is caught by the SystemC simulation kernel and the kernel will then notify a set of *listeners* that have registered themselves to this signal. This mechanism gives us an opportunity to monitor the modification that may occur on a SystemC signal without being intrusive or having to modify the simulation kernel.

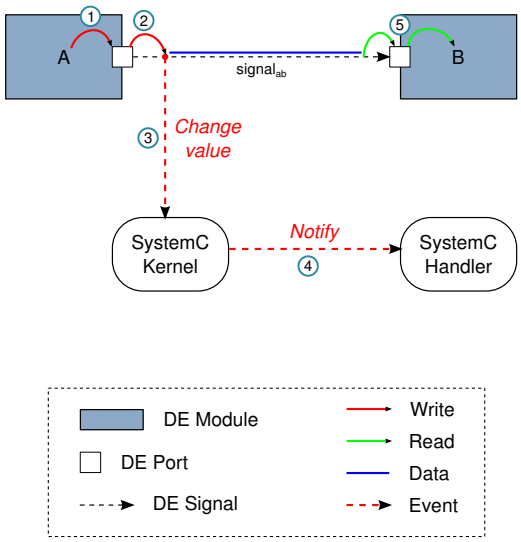

Figure 6.3: SystemC MDVP kernel handling of SystemC monitoring.

Figure 6.3 illustrates the mechanism that we developed in order to handle SystemC components. This figure describes a simple SystemC system composed of two modules, two ports and a signal. The behavior of this system is quite simple, module A writes data which is then read by module B. We see that the module A writes data to its port (Figure 6.3, ①), then the port writes this data to the signal $signal_{ab}$ (Figure 6.3, ②). When this occurs, $signal_{ab}$ triggers an event that is caught by the kernel (Figure 6.3, ③), which then notifies the `sc_monitor_handler` (Figure 6.3, ④). Module B can read the data through its port ⑤; we do not interfere in this process.

We explained that the SystemC kernel notifies the set of listeners that have registered themselves to a specific signal when a writing occurs. In our approach the `sc_monitor_handler` represents a listener that wants to be notified when a writing occurs on a signal that needs to be monitored. Prior to the simulation, the SystemC MDVP kernel, therefore, has to register the `sc_monitor_handler` to each SystemC signal to be monitored in order for the SystemC monitor handler to be notified during the simulation. This is automatically done when the designer of the system specifies a SystemC signal to monitor.

As with the tracing mechanism implemented within SystemC, our monitoring mechanism does not support all the SystemC components. We can monitor channel objects used to interconnect SystemC modules though we are not able to handle the channels described using a `sc_fifo`. Indeed, the SystemC reference manual [39] does not specify a mechanism which would allow us to retrieve values stored in a `sc_fifo` without altering the FIFO. If we attempt to read the value stored in a FIFO, this value will no longer be stored in the FIFO. Hence, in the example shown in Figure 6.3, the module B will fail to receive the value written by the module A. We are, therefore, only able to monitor channels described with `sc_signal`.

6.2.2 SystemC MDVP Monitoring

Within SystemC MDVP, the monitoring mechanism relies on the principle of signal/slot or event/delegate mechanism. [1] This pattern was introduced in Qt in order to handle communication between objects. It is especially used in the implementation of Graphical User Interfaces (GUI). This pattern represents a communication mechanism that allows for the decoupling of a sender from several potential receivers. The principle is to make information from one part of the system available to another without having to hard-wire these two resources.

A signal is emitted when a specific event occurs and a slot represents a function that is called in response of a specific signal. When a signal is emitted it makes specific predefined data available to several slots, provided that each slot had registered itself with the signal.

Within SystemC MDVP, this approach perfectly fits the requirements in order to perform the monitoring process. Applied to our framework, this can be understood as when a writing on a channel occurs a signal is emitted. The signal can carry the information associated with the new value written in the channel, and other information such as the timestamps when the writing occurs. Logically, the slot function called when a signal is emitted is defined within a monitor slot.

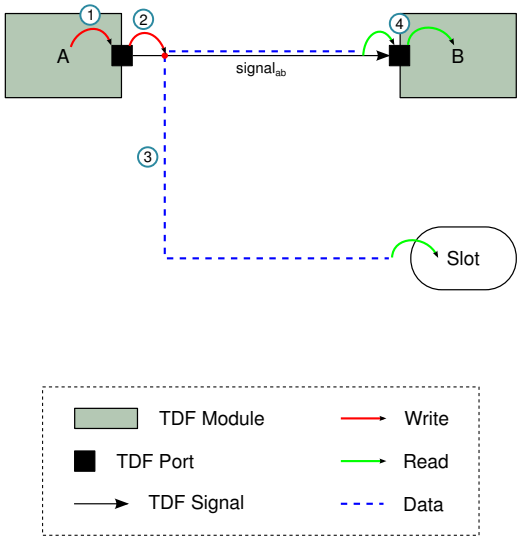

Figure 6.4: SystemC MDVP kernel monitoring principle.

A representation of the mechanism that we developed with the aim of monitoring SystemC MDVP components can be found in Figure 6.4 which denotes a simple SystemC MDVP system comprising two modules, two ports, and a signal from the TDF MoC. As with the example to illustrate the handling of SystemC components, this system behaves in a straightforward manner with module B reading the data written by module A. We note that, when module A writes data to its port (Figure 6.4, ①) and then the port writes this data to the channel $channel_{ab}$ (Figure 6.4, ②), this channel emits a signal which notifies the slot with which it is registered (Figure 6.4, ③). As before, the data is read by module B, through its port (Figure 6.4,

[1] In order to avoid the confusion between signals belonging to SystemC MDVP and signals described in the signal/slot pattern, signals from SystemC MDVP will be referred as channels in this section

④), without any intervention on our part.

Every time a value is written to a SystemC MDVP channel, it is sent through a signal to the slot associated with the channel in question. The slot then formats the data received and stores it until it is collected by the `scm_monitor`.

As previously mentioned, a monitor slot is created for each cluster of the hierarchy of clusters, and hence the registration of a slot with a signal is automatically performed after this creation process. A monitor slot, therefore, may be connected to several different signals in the event that multiple channels belonging to the same cluster need to be monitored.

Developing the monitoring mechanism raises the question of how it could collect relevant specific information while remaining a generic mechanism. We address this issue with the abstraction previously introduced. This means that the monitor slot used to handle the channels of a MoC must be MoC-dependent and specifically implemented for a MoC. The MoC Architect is in the best position to select the relevant information which needs to be monitored. Indeed, he designed the MoC and, as such, can easily identify what information should be taken into account when the MoC is being monitored. This approach guarantees the flexibility of our virtual prototyping environment even for monitoring purpose.

6.3 Monitoring Mechanism

The mechanisms involved in the monitoring process are presented in Figure 6.5. Based on Figure 5.3 and Figure 5.10, this figure shows how the monitoring mechanism is integrated within SystemC MDVP with a view to monitor heterogeneous entity without altering the SystemC kernel. First, the monitoring mechanism is broken down into three sub-phases which are integrated into the elaboration process of SystemC MDVP. These three sub-phases aim at setting up the resources required to execute a generic monitoring mechanism. They consist in creating and initializing a set of data structures. Second, a kernel routine, which is integrated into the scheduler of the SystemC DE kernel simulation, performs the monitoring execution. Thereafter, through the `end_of_simulation()` callback from SystemC the monitoring process can properly terminate its on-going tasks.

This section details the different mechanisms presented in Figure 6.5.

Figure 6.5 details the monitoring mechanism incorporated within SystemC MDVP to promote correct functioning. Our framework is designed in such a way that the end-user need not to be involved with the inner-working (internal elements) of the simulator and it is of the utmost importance that this approach be upheld with the monitoring process. As such, in order to remain generic end keep the involvement of the end-user to an absolute minimum, we hereby introduce several addition sub-phases to the elaboration process of SystemC MDVP. These sub-phases allow us to automatically setup the monitoring infrastructure in a generic manner.

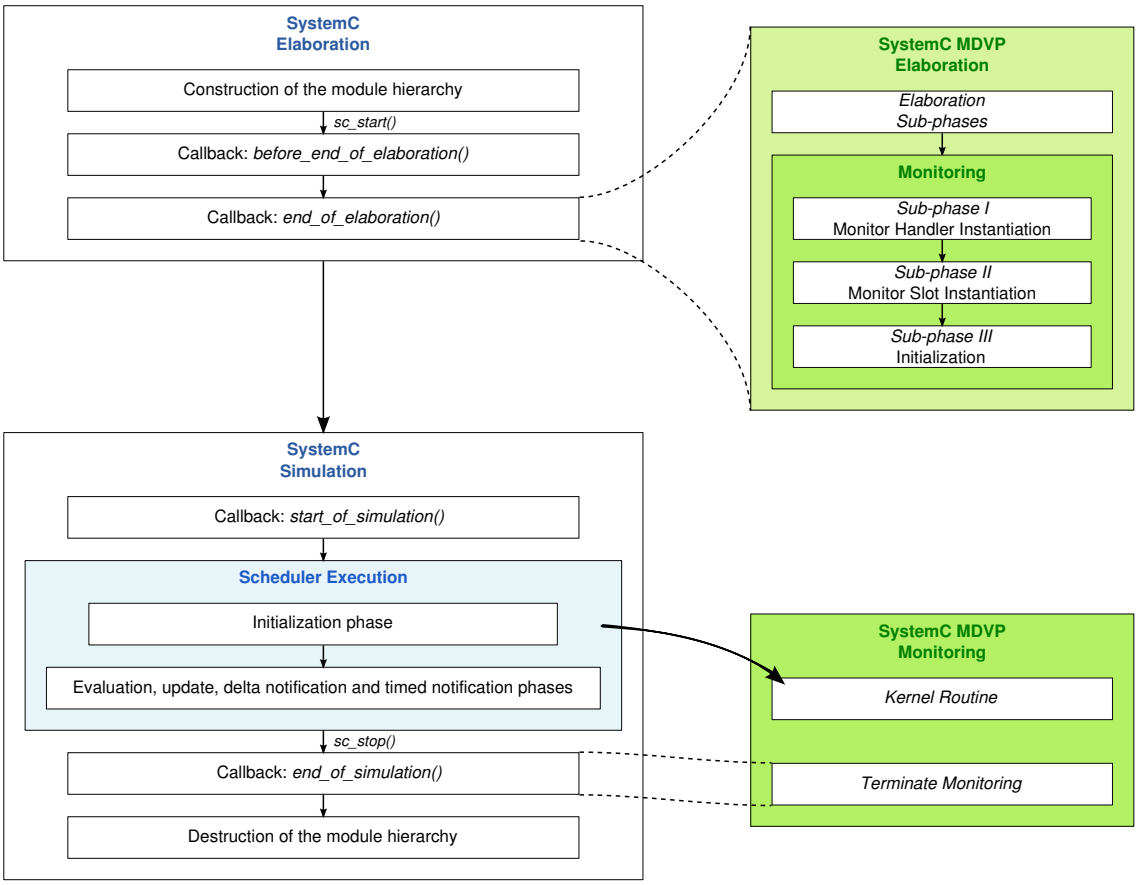

Figure 6.5: SystemC MDVP kernel monitoring phases.

6.3.1 Monitor Handler Instantiation

As previously stated, the monitor handlers represent managers which control the monitoring process during the simulation. When dealing with SystemC MDVP components, they are associated with a whole branch of the cluster hierarchy, and manage its monitoring. When dealing with SystemC components, they handle them directly.

To create these handlers, we take advantage of the work carried out during the elaboration phase. We use the cluster hierarchy built during the clustering sub-phases to support our creation process. It is a straightforward implementation as detailed in Algorithm 6.1.

The principle is to iterate over the top sub-clusters in order to instantiate a monitor handler for each branch of the cluster tree. We systematically examine all the sub-clusters encapsulated within the root of the tree (Algorithm 6.1, line 2), and for each sub-cluster we retrieve its associated solver (Algorithm 6.1, line 3). Then, we can instantiate a handler dedicated to the branch of the cluster tree represented by a solver (Algorithm 6.1, line 4). In order to keep a record of the correspondence between a solver and its dedicated monitor handler, we store this information through a `map` object which associates a key (a solver) to a value (a monitor handler) (Algorithm 6.1, line 5). By proceeding in this manner, we are able to retrieve a monitor handler associated with a solver easily and conveniently.

```
1 Function monitor_instantiation()
    Data: Cluster cl_root.
    Data: Map<solver, monitor_handler> solver_map.
    Result: void.
2   foreach sub_cl in the clusters list of cl_root do
3     solver = sub_cl.get_moc_interface();
4     monitor_handler = new sca_core::sca_monitor_handler();
5     solver_map[solver] = monitor_handler;
6   end foreach
7 end
```

Algorithm 6.1: Generic Algorithm to Instantiate Monitor Handlers.

The instantiation of the SystemC monitor handler is also straightforward. Within SystemC MDVP we do not track the SystemC components instantiated in the model description; these SystemC components are not represented in the data structures of SystemC MDVP. Therefore, the monitor handler dedicated to the handling of SystemC components need not conform to any internal representation, hence, it is simply instantiated as a single entity that will handle all the SystemC components.

6.3.2 Monitor Slot Instantiation

The instantiation of monitor slots is more complex than that of the monitor handlers. Though they must remain generic in order to respect the principles of SystemC MDVP, they must also be MoC-specific in order to retrieve relevant information. We follow the same approach that we applied to the definition of the solvers, which are MoC-specific. We create a dictionary which stores for a MoC its associated monitor slot.

To this aim, we use the class `scm_moc_info` defined earlier in this chapter to store the information that we need. We use the design pattern Prototype to create the monitor slot. Thus, we require a prototype to allow us to apply the clone process. The `scm_moc_info` class of each MoC must contain an entry in a table which defines the associated monitor slot for this MoC. This information is then gathered by the simulation context with a view to construct a table representing all the monitor slots available in the simulated system (i.e. a monitor slot for each MoC used in the design of the system).

We now have all the available monitor slots of the system and their prototype, though we still need to perform the instantiation of the monitor slots for each cluster of the system. The instantiation of the monitor slots is carried out in a generic way thanks to the recursive process described in Algorithm 6.2.

Algorithm 6.2 describes the generic function `create_monitor_slot()` that creates the monitor slots recursively, following a bottom-up process. It takes a monitor handler and a cluster as parameters; the cluster is used to go through the hierarchy of clusters, and the monitor handler to

```
1  Function create_monitor_slot(monitor, node)
       Data: Monitor Handler monitor.
       Data: Cluster node.
       Data: Map<cluster, monitor slot> cluster_slot_map.
       Result: Monitor Slot slot.
2      foreach sub_cl in the clusters list of node do
3          slot = create_monitor_slot(monitor, sub_node);
4          monitor.add_slot(slot);
5      end foreach
6      moc_name = node.get_moc() ;
7      prototype_slot = prototype_table.find_slot(moc_name);
8      if prototype_slot == NULL then
9          print("Error, cannot find a monitor slot associated with moc_name") ;
10         exit() ;
11     end if
12     slot = prototype_slot.clone();
13     cluster_slot_map[node] = slot ;
14     return slot;
15 end
```

Algorithm 6.2: Generic Recursive Algorithm to Create Monitor Slots.

register the monitor slot associated with it.

As a bottom-up process, we are undertaking a deep-first traversal of the hierarchy, which means that the first thing to do is to iterate over the sub-cluster of the current cluster (Algorithm 6.2, line 2). The function `create_monitor_slot()` is called on each sub-cluster and returns the monitor slot associated with the sub-cluster (Algorithm 6.2, line 3). This monitor slot is then added to the monitor handler provided as parameter of the function (Algorithm 6.2, line 4). The first part of the Algorithm 6.2 describes the registering of monitor slots with their corresponding monitor handlers.

The second part of this algorithm describes the creation of the monitor slots. First of all, we access the name of the Model of Computation associated with the cluster under treatment (Algorithm 6.2, line 6). With this name, we can try to access the table that contains the associated prototype of monitor slots (Algorithm 6.2, line 7). This step of the algorithm may fail (Algorithm 6.2, line 8). Failure can happen if the end-user requests our framework to monitor a component associated with a MoC that does not support the monitoring mechanism of SystemC MDVP. In this case, an error message is displayed to the end-user and the process is terminated (Algorithm 6.2, line 10). In contrary, the retrieved prototype is cloned (Algorithm 6.2, line 12) in order to create the monitor slot associated with the cluster *node*. Then, this information is stored in a table for futur use (Algorithm 6.2, line 13). Eventually, the slot which has just been created is returned (Algorithm 6.2, line 14).

The function `create_monitor_slot()` allows for the creation of monitor slots which describe a single branch of the hierarchy of the cluster, and are hence, associated with a single monitor handler. Therefore, the call to the function `create_monitor_slot()` is encapsulated within another function which, collectively, constitutes the instantiation mechanism of monitor slots.

This function is detailed in Algorithm 6.3.

```
1 Function instantiate_monitor_slot()
      Data: Monitor Handler monitor.
      Data: Cluster cl_root.
      Data: Map<solver, monitor handler> solver_map.
      Result: void.
2     foreach sub_cl in the clusters list of cl_root do
3         solver = sub_cl.get_moc_interface();
4         monitor_handler = solver_map[solver];
5         slot = create_monitor_slot(monitor_handler, sub_node);
6         monitor.add_slot(slot);
7     end foreach
8 end
```

Algorithm 6.3: Generic Algorithm to Instantiate Monitor Slots.

`instantiate_monitor_slot()` represents the generic function that triggers the instantiation of the monitor slot for the whole hierarchy of clusters.

The principle is to iterate over the top sub-cluster in order to instantiate a monitor handler for each branch of the cluster tree. We pass through all the sub-clusters encapsulated within the root of the tree (Algorithm 6.3, line 2), and, for each cluster, we retrieve its associated solver (Algorithm 6.3, line 3). Thereafter, we can retrieve the monitor handler dedicated to the branch of the cluster tree represented by a solver (Algorithm 6.3, line 4).

With this information, we can trigger the creation of the monitor slots for the branch of the hierarchy tree represented by *sub_node* and acquire the monitor slot associated with *sub_node* (Algorithm 6.3, line 5). Finally, this monitor slot is integrated into the monitor handler of this branch of the tree (Algorithm 6.3, line 6).

At this stage, we have created all the components needed to perform the monitoring, that is to say the monitor handlers and the monitor slots. What remains to be done is to initialize them for the simulation.

6.3.3 Initialization

Prior explaining how the monitoring system is initialized, we need to introduce how the end-user interacts with the monitoring mechanism. In the beginning of this section we stated that the end-user will interact with the monitoring process through the `scm_monitor` class and only with this object. The end-user uses this object to specify the components he wants to be monitored during the simulation. He can specify SystemC or SystemC MDVP components using the same interface. This is illustrated in Listing 6.1, which represents a sample of the code behind the running example of the vibration sensor.

```
1 #include <sc_mdvp.h>
```

```
2  [...]
3  int main(){
4    [...]
5
6    // Signals
7    scm_tdf::scm_signal<double> x_sig_x("x_sig_x");
8    scm_tdf::scm_signal<double> v_sig_x("v_sig_x");
9    scm_tdf::scm_signal<double> v_amp_sig_x("v_amp_sig_x");
10   scm_tdf::scm_signal<sc_dt::sc_int<NBitsADC> > adc_sig_x("adc_sig_x");
11
12   sc_core::sc_signal<sc_dt::sc_int<NBitsADC> > adc_out_sig_x("out_sig_x");
13   sc_core::sc_signal<int> k_out_sig_x("k_sig_x");
14   sc_core::sc_signal<sc_dt::sc_int<NBitsADC> > amp_sig_x("amp_sig_x");
15   sc_core::sc_signal<bool> clk_sig_x("clk_sig_x");
16
17   // Monitoring
18   scm_core::scm_monitor monitor;
19
20   monitor.monitor_signal(&x_sig_x);
21   monitor.monitor_signal(&v_sig_x);
22   monitor.monitor_signal(&v_amp_sig_x);
23   monitor.monitor_signal(&adc_sig_x);
24   monitor.monitor_signal(&k_out_sig_x);
25   monitor.monitor_signal(&amp_sig_x);
26
27   [...]
28 }
```

Listing 6.1: End-user monitoring specification

This monitor registers the components to be monitored in two separate lists, one for the SystemC components and one for the SystemC MDVP components. Each list is used to initialize the corresponding data structure within SystemC MDVP.

Algorithm 6.4 describes the initialization of the monitor slots. In order to monitor the system, these slots must register themselves with the signals they have to monitor. This is the purpose of this initialization phase, the main idea being to take each cluster, identify the channels that need to be monitored and link it with the corresponding monitor slot.

Since we adopt a bottom-up initialization process, with a deep-first traversal of the cluster hierarchy, and hence, we first iterate over all the sub-clusters (Algorithm 6.4, line 2) to recursively call the initialization function (Algorithm 6.4, line 3). Thanks to the structure created during the instantiation of the monitor slots, we are able to retrieve the monitor slot associated with each cluster (Algorithm 6.4, line 5). Then, as all the channels used to interconnect a set of elements belonging to a cluster are stored in the cluster, we can easily gain access to them (Algorithm 6.4, line 6).

```
1  Function initialize_monitor_slot(node)
       Data: Monitor Handler monitor.
       Data: Cluster node.
       Data: Map<cluster, monitor slot> cluster_slot_map.
       Data: List<Channel> channel_monitored.
       Result: void.
2      foreach sub_cl in the clusters list of node do
3          initialize_monitor_slot(sub_node);
4      end foreach
5      slot = cluster_slot_map[node] ;
6      cl_channels = node->get_channels() ;
7      foreach channel in the channel list of cl_channels do
8          ch_to_monitor = channel_monitored.find_channel(channel);
9          if ch_to_monitor then
10             channel->connect_slot (slot);
11         end if
12     end foreach
13 end
```

Algorithm 6.4: Recursive Algorithm to Initialize the Monitor Slots.

We go through this list of channels (Algorithm 6.4, line 7) and, for each channel, we check if it belongs to the list of channels to be monitored (Algorithm 6.4, line 8). If not, we directly move on to the next channel. If it does belong to the list, we connect this signal with the corresponding monitor slot (Algorithm 6.4, line 10).

`connect_slot()` represents the function that registers a slot with a signal following the signal/slot pattern used to perform the monitoring for SystemC MDVP components. This function is MoC-specific and can then perform the connection with the MoC-specific instance of the monitor slot. This approach allows us to keep our framework generic, and independent from the MoC definition, while enabling the monitoring process to access relevant and specific information.

The SystemC MDVP monitor handlers do not require an initialization process. They handle the monitor slots that correspond to their branch of the cluster and have already been associated with their monitor handler during their instantiation. Thus, no further initialization is needed. As opposed to that of SystemC MDVP, the SystemC monitor handler requires an initialization process. This monitor handler works as a SystemC process which is sensitive to all of the signals that it has to monitor. As with the solver communicating with DE, we have to create a dynamic process that will be registered in the SystemC kernel scheduler. This is achieved during the initialization phase. The SystemC function that allows us to create process (`sc_spawn()`) also allows us to specify a list of sensitivities which will determine when our process should be executed; this is in perfect keeping with our approach. We can then create the process that performs the monitoring of SystemC components sensitive to the entire signals that it has to monitor.

6.3.4 kernel Routine

The monitoring must be performed for the entire duration of the simulation; as such we need to come up with a method which will allow the monitoring mechanism to be executed throughout the simulation process. We do not want the monitoring to result in a huge overhead on the simulation time, therefore we chose an approach where the monitoring process is performed sporadically. To this aim, we encapsulated the monitoring mechanism within a kernel routine, defined as a SystemC process.

With a sporadic execution, a burst treatment of the data is possible. Consequently, the kernel routine should provide a mechanism to indicate when a set of data is available and, thus, the monitoring process should be performed. Therefore, we define a kernel event that controls the execution of the kernel routine. The kernel routine is sensitive to this event. When this kernel event is triggered, the kernel routine may be executed. Figure 6.6 illustrates how the kernel routine in charge of the monitoring operates within SystemC MDVP.

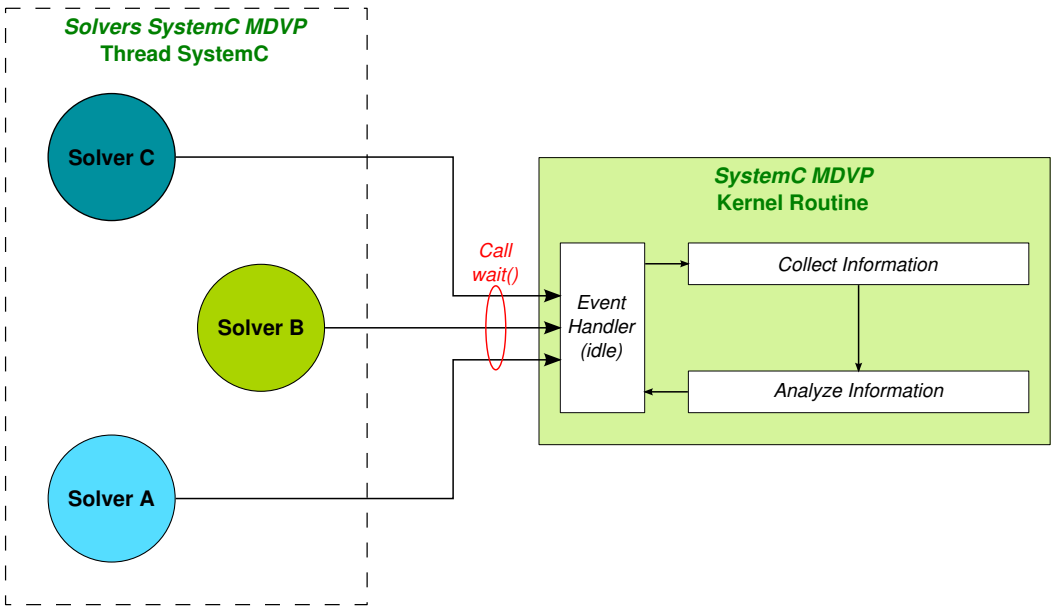

Figure 6.6: SystemC MDVP monitoring kernel routine.

We previously explained how the solvers belonging to SystemC MDVP, which interact with DE, integrate themselves with SystemC. This mechanism relies on the creation of dynamic processes, threads, directly integrated into the scheduler of the SystemC kernel. In order for the kernel to gain control of the execution, the threads must call a *wait()* function. This function interrupts the thread execution in order for the SystemC kernel to execute its own code. Within SystemC MDVP we provide our own `wait()` function so as to allow our own kernel to execute its own code when a solver ends its execution cycle.

The `wait()` function acts as a wrapper for the SystemC `wait()` function. Prior to calling the function provided by SystemC we trigger the kernel event in order to notify our kernel routine of the fact that a solver has ended a simulation cycle and monitored data are, therefore, available.

When the solvers A, B or C end a simulation cycle, they call the `wait()` function provided by SystemC MDVP. This `wait()` function triggers the kernel event dedicated to the kernel routine, then it calls the `wait()` function provided by SystemC.

Several solvers could execute their simulation cycle consecutively before the kernel routine executes its process. Since SystemC does not track multiple triggering of events, we cannot know if the kernel event was triggered several times, and, hence, if several solvers end their simulation cycle. For this reason, in addition to triggering the kernel event to notify the end of a simulation cycle, we also provide a registration mechanism which allows a solver to register itself when it finishes a simulation cycle. This mechanism allows us to retrieve and identify all the solvers that trigger the kernel event.

The kernel routine waits until a kernel event is triggered. When our routine is notified by SystemC of the event being triggered, it can execute the monitoring process. This process consists in two main tasks: the gathering of information and the analysis of this information. When these two tasks have been completed, the kernel routine returns to its initial state, waiting for the kernel event to be re-triggered. Figure 6.7 shows the kernel routine tasks applied to the running example of the vibration sensor.

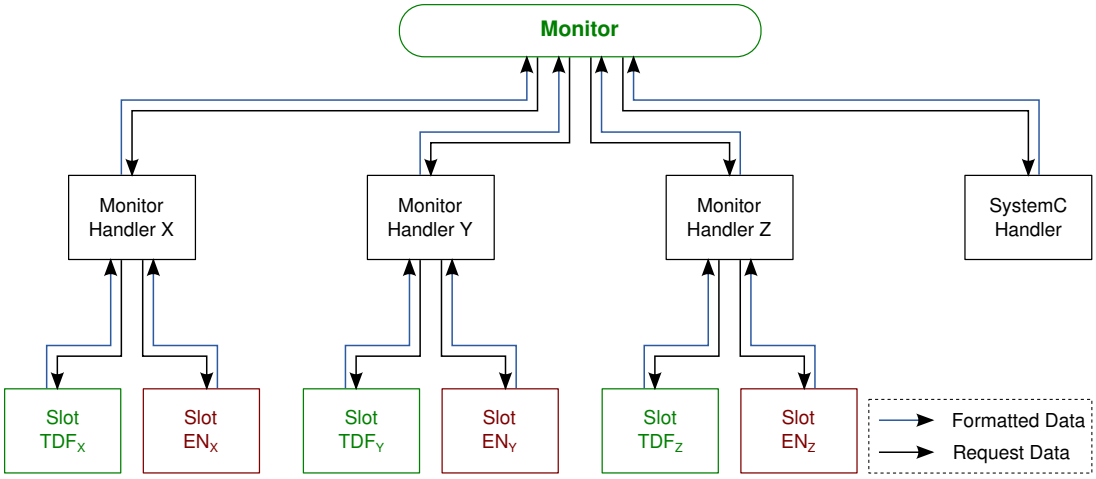

Figure 6.7: Running Example: Monitoring execution for the vibration sensor.

The execution of the kernel routine tasks relies on the monitor tree representation introduced by Figure 6.2. The kernel routine identifies the solvers that have ended a simulation cycle since its last execution. The routine then requests the information, stored during the simulation, from each solvers' associated monitor handler through the `scm_monitor`. This request is propagated along the monitoring tree to the different monitor slots in charge of monitoring SystemC MDVP components. Each monitor handler collects the information provided by the monitor slots under its responsibility and sorts the data in accordance with the simulated time. The same applies for SystemC components except that they are all directly handled by the SystemC handler. In summation, the `scm_monitor` retrieves all the information from SystemC and SystemC MDVP components.

6.3.5 End of Simulation

In addition to the monitoring sub-phases added to the elaboration process, and the Kernel routine added to the SystemC kernel scheduler, we are also obliged to add a phase during another callback of SystemC. For the monitoring process to be performed properly, the monitoring routine must be executed once after the simulation has ended. Fortunately, SystemC provides a mechanism for this purpose - the `end_of_simulation()` callback. As illustrated in Figure 6.5, this callback is performed after the call to `sc_stop()`.

Despite the fact that the data are normally processed by the kernel routine that performs the monitoring, a set of data, which was not handled by the monitoring mechanism, remains to be monitored at the end of the simulation. We must, therefore, process these data manually. Indeed, for this last set of data, we need to perform the work of the kernel routine during the `end_of_simulation()` callback of SystemC.

6.4 Conclusion

The monitoring of multi-disciplinary systems represents a key aspect of virtual prototyping of heterogeneous systems. In the context of the solution we propose, the primary challenge lies in handling both digital (SystemC) and multi-disciplinary (SystemC MDVP) components. In this chapter, we introduced our monitoring mechanism conceived with a view to achieve this goal.

We take advantage of everything SystemC has to offer in order to perform the monitoring of SystemC components without altering the DE simulation kernel. This leads us to the design of a mechanism involving a SystemC monitor handler in charge of all the SystemC components. We used the notification mechanism available within SystemC to be alerted when a value change occurs on a monitored signal. It gives us access to the value thus allowing us to probe SystemC components easily.

To perform the monitoring of SystemC MDVP components we exploit the pattern of signal/slot, commonly used in QT-based applications. This pattern allows us to decouple the sender from the receiver and, hence, does not require that we hard-wire the probing mechanism with the modeled system. MoC-specific monitor slots are created in order to handle MoC-specific values. These monitor slots are abstracted within the kernel the same way MoCs are. As we have shown, this approach allows us to remain completely generic within the kernel while being able to retrieve specific, relevant information from the system. Monitor slots are handled through a monitor handler. The infrastructure of these monitor objects relies on the cluster hierarchy built during the elaboration phase. Eventually, the data structures and mechanisms presented are used within a kernel routine, which is defined as a SystemC process that performs the monitoring of an entire multi-disciplinary system.

Our approach to accomplish the monitoring of multi-disciplinary systems respects the objectives of SystemC MDVP; it does not alter the flexibility and genericity of our virtual prototyping environment. Moreover, we are able to provide the end-user with a single, unique and common interface to monitor components belonging to digital or analog domains.

Figure 6.8 illustrates the results that can be obtained by using the SystemC MDVP monitoring mechanism. This figure denotes the simulation traces of the running example of the vibration sensor for the `Sensor X`. It corresponds to the monitoring specification described in Listing 6.1.

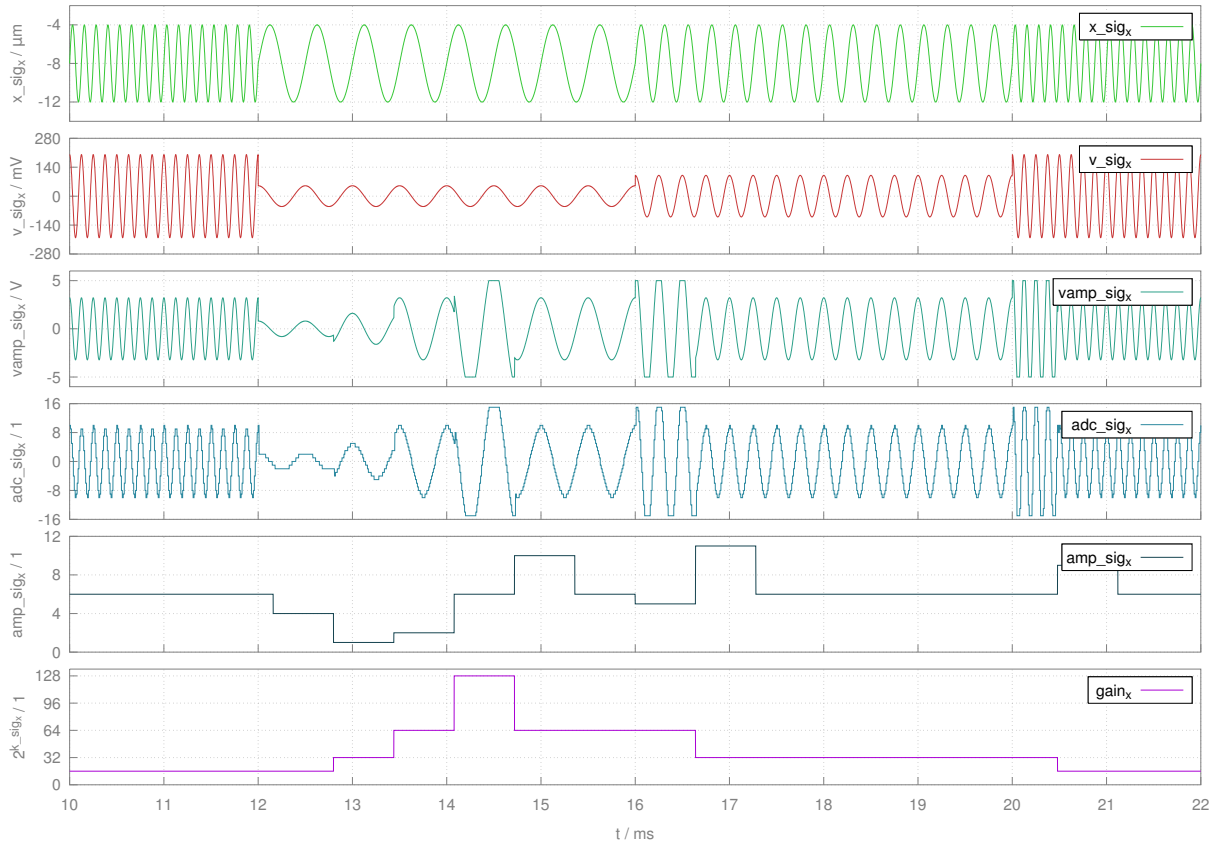

Figure 6.8: Running Example: Simulation Traces for the `Sensor X` of the vibration sensor

Chapter

7 *New MoC Integration Methodology: Application to SPH*

Contents

7.1 Introduction 100
7.2 New MoC integration methodology 100
7.2.1 Interfacing Step 101
7.2.2 Implementation Step 105
7.2.3 Interaction Step 106
7.3 Application to SPH MoC 108
7.3.1 Interfacing Step 109
7.3.2 Implementation Step 110
7.3.3 Interaction Step 114
7.4 Conclusion 114

7.1 Introduction

One of the most important challenge we want to tackle is to provide a flexible virtual prototyping environment, which can increase the set of heterogeneous entities it can handle. Within SystemC MDVP framework, heterogeneous entities are represented through Models of Computation, thus the scalability relies on the possibility to add new MoCs within the simulator. Our framework is designed to be independent from the MoC definition, which allows an easy integration process of new Models of Computation. The design of new Models of Computation is done by a MoC architect, as introduced in Chapter 4. He needs to follow a methodology to perform the flawless integration of its new entity.

This chapter presents the methodology step-by-step that a MoC architect should follow in order to enrich SystemC MDVP with a new MoC and is organized as follows.

First, in Section 7.2 we follow a general approach introducing the main steps constituting the integration methodology. These steps are referenced as the *Interfacing Step*, the *Implementation Step* and the *Interaction Step*. These steps allow for a smooth integration process.

Then, in Section 7.3 we provide an example of integration of a new MoC. We apply our methodology in order to define a fluidic MoC: Smoothed Particle Hydrodynamics (SPH). The integration of this MoC illustrates the easiness of enhancing our virtual prototyping environment with new MoCs.

Finally, Section 7.4 concludes this chapter and provides an overview of the MoC integration methodology within SystemC MDVP.

7.2 New MoC integration methodology

The methodology to design and integrate a new MoC within SystemC MDVP is articulated around three main steps. The methodology defined by these three steps should be respected by a MoC architect in order to easily and conveniently manage the design and the integration of his new MoC within our framework. This approach only defines a pathway that we believe is the best to enrich our framework with new MoC. The MoC integration process and the steps associated with our methodology are described in Figure 7.1.

The first step is called the *Interfacing Step*. During this step the MoC Architect perform the mandatory tasks required in order for his MoC to be handled by SystemC MDVP. Indeed, SystemC MDVP relies on an abstraction of MoCs therefore he has to respect this abstraction while defining his MoC.

The second step is called *Implementation Step*. During this step the MoC Architect performs

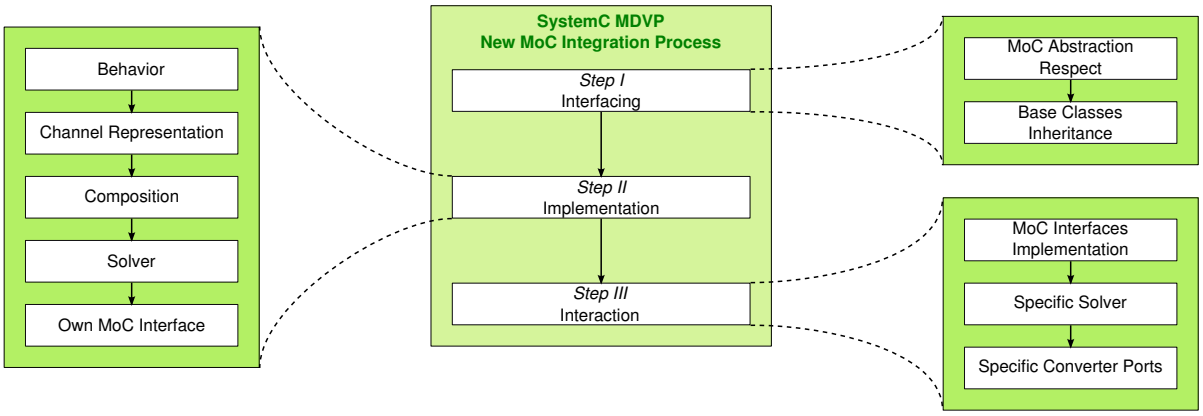

Figure 7.1: MoC Integration Process

all the internal tasks of his MoC without interacting with SystemC MDVP. He implements the internal structure of his MoC. He defines the basic blocks, the behavior, the communication mechanisms, and the solver of his MoC.

The third step is called *Interaction Step*. During this step the MoC Architect performs the tasks required in order for his MoC to interact with others MoCs within the SystemC MDVP environment. He has to define the master-slave semantics that his MoC has to follow.

These steps define the integration methodology, and hence the process a MoC Architect should go through in order to design and integrate a new Model of Computation within the SystemC MDVP framework. In the following of this section we detail these different steps by integrating a new MoC *X*.

7.2.1 Interfacing Step

The *Interfacing step* represents the phase when a new Model of Computation is plugged in within SystemC MDVP. As previously mentioned, SystemC MDVP relies on an abstraction of MoCs to handle the different MoCs provided with the framework. Thanks to this abstraction our framework is able to automatically handle the MoCs and apply generic algorithm to them. In order to respect this abstraction a pretty simple process based on inheritance concept has to be followed. It is required that the MoC Architect extend the set of basic classes used to abstract a MoC by the kernel (presented in Figure 5.1).

Figure 7.2 illustrates the inheritance pattern required to define the behavior interface of a new MoC. First, the MoC Architect has to provide a class `scm_mocX::scm_moc_info` to encompass all the generic information regarding his MoC (①). This class inherits from the kernel class `scm_core::scm_moc_info`. Second, he has to provide a class `scm_mocX::scm_moc_if` to define his MoC's interface (②). This class inherits from the class `scm_core::scm_moc_if`.

Now based on the `scm_mocX::scm_moc_if` and the `scm_mocX::scm_moc_info` classes he can

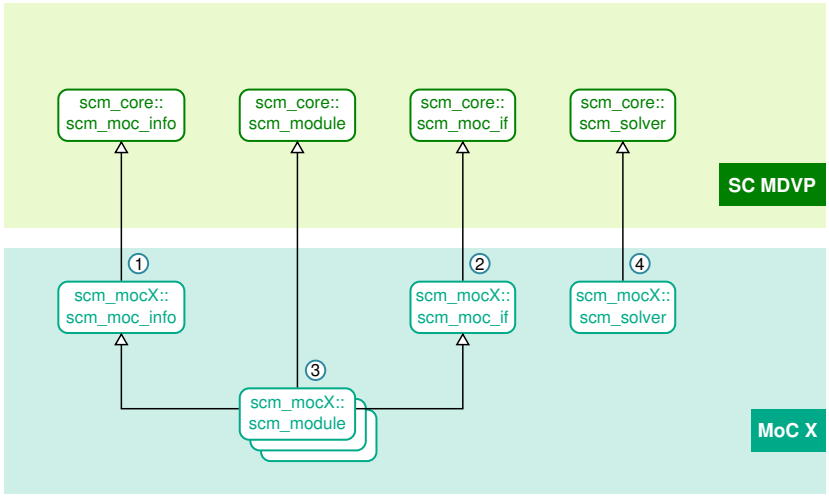

Figure 7.2: Inheritance pattern to define behavior of a new MoC.

define a basic elementary behavior. The MoC Architec has to define a class `scm_mocX::scm_module` to represent an elementary behavior of his MoC (③). This class inherits from the kernel class `scm_core::scm_module` in order to respect the abstraction. It also inherits from the class `scm_mocX::scm_moc_if` since it has to respect the interface of its own MoC. The kernel of simulation needs that this class also inherits from the class `scm_mocX::scm_moc_info`. This last inheritance is required in order to automatically register the MoC within SystemC MDVP when an elementary block is instantiated. The implementation details of the elementary behavior are not required, one can define several basic blocks; we required at least one module.

Finally, he has to define a class `scm_mocX::scm_solver` to represent the solver of his MoC (④). This class inherits from the kernel class `scm_core::scm_solver`.

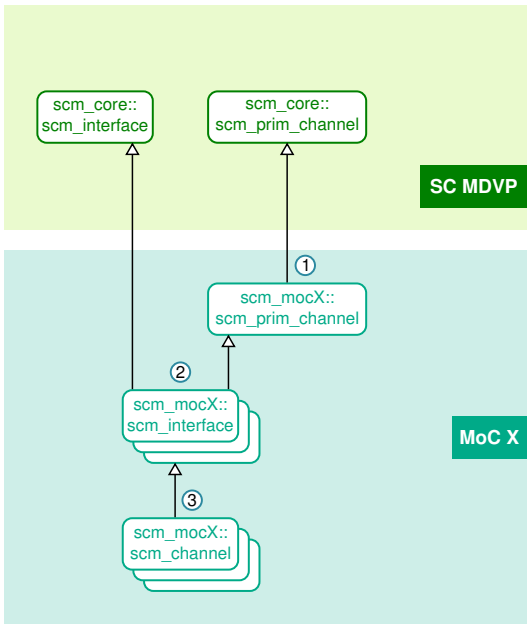

Figure 7.3: Inheritance pattern to define communication channel of a new MoC.

The same process is applied to specify the signals defined within a new MoC and is illustrated

in Figure 7.3.

First the MoC Architect has to provide a class `scm_mocX::scm_prim_channel` to represent the basic behavior of his channels (①). This class inherits from the kernel core class `scm_core::scm_prim_channel`. Usually, the signals transport data where their type is represented by a template parameter. One cannot manipulate such an object without knowing the value of the template parameter. Thus, this basic class allows for the manipulation of the signals of the new MoC by the SystemC MDVP kernel and by the MoC Architect without requiring the specification of a template parameter. Hence, it maintains the flexibility of our virtual prototyping environment.

Then, he has to provide a communication interface to define the way to access to a signal. The purpose of these communication interfaces is to be used by a port in order to communicate with a signal. This is achieved through the definition of the class `scm_mocX::scm_interface` (②). One can define as many communication interfaces as he wants, we require at least one communication interface. These interfaces must inherit from the kernel classes' `scm_core::scm_interface` and `scm_core::scm_prim_channel`.

Eventually, he defines the signals used as communication channel within his MoC. Thus, he provides a class `scm_mocX::scm_channel` (③). This channel inherits from the above mentioned communication interface. Like the communication interfaces, one can define as many channels as he wants, we do not define restriction for the channels.

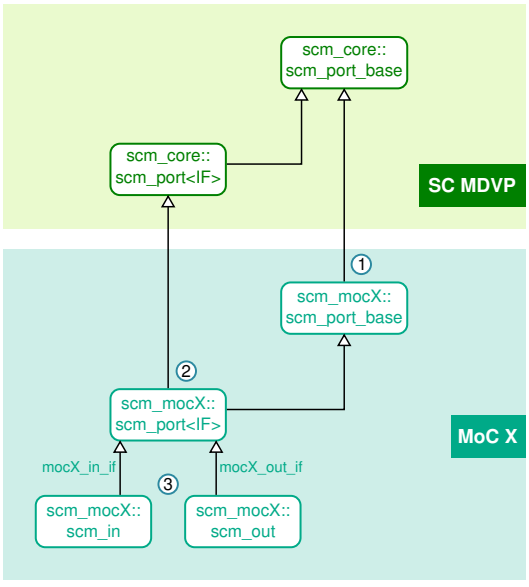

Figure 7.4: Inheritance pattern to define composition mechanism of a new MoC.

Once again, the same principles apply for the ports defined within a new MoC as illustrated in Figure 7.4.

First the MoC Architect has to provide a class `scm_mocX::scm_port_base` to represent the basic behavior of his ports (①). This class inherits from the kernel core class `scm_core::scm_port_base`.

Usually, the ports, as the signals, handle data where their type is represented by a template parameter. As aforementioned, one cannot directly manipulate such objects without knowing the template parameter value. Thus, this basic class allows for the manipulation of the ports of the new MoC by the SystemC MDVP kernel and by the MoC Architect without requiring the specification of a template parameter. Hence, it maintains the flexibility of our virtual prototyping environment.

Now he has to define the basic class of all the ports of his MoC. This is done through the class `scm_mocX::scm_port<IF,T>` (②) that inherits from the kernel class `scm_core::scm_port<IF>`. It also inherits from the class `scm_mocX::scm_port_base` in order for the ports to be manipulated by the kernel using the basic class defined.

Eventually, he defines the ports provided to the SoC architect (the end-user). They are represented by the classes `scm_mocX::scm_in` and `scm_mocX::scm_out` (③). One can define as many ports as he wants as long as they inherit from the class `scm_mocX::scm_port<IF>`. In addition of this inheritance, they also have to implement a specific communication interface (presented in Figure 7.3) in order to interact with the channels defined within the new MoC (`mocX_in_if` and `mocX_out_if` as an example).

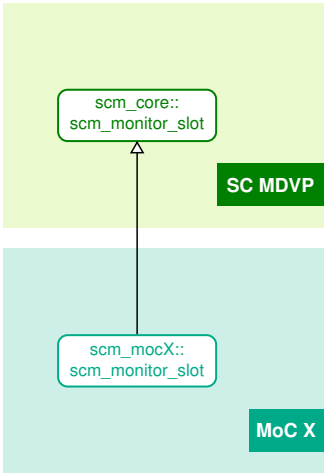

Figure 7.5: Inheritance pattern to define monitoring of a new MoC.

Following the same inheritance principle the MoC Architect can define the necessary class required in order to perform the monitoring of channels associated with his MoC. This is shown in Figure 7.5. He has to provide a single class `scm_mocX::scm_monitor_slot` that inherits from `scm_core::scm_monitor_slot` to respect the monitoring abstraction required by the kernel SystemC MDVP.

The classes presented during this integration step constitute the basic infrastructure of a new MoC. Now one can go to the next step and specify the implementation details of a new MoC.

7.2.2 Implementation Step

The *Implementation step* represents the phase when the internal details of the new Model of Computation are specified. Since SystemC MDVP is designed in order to be independent from the definition of MoCs, the MoC Architect is free to implement his MoC the way he wants. SystemC MDVP only imposes that he implements the functions required by the abstraction of MoCs it used.

He has to specify the behavior of his MoC, starting by the behavior associated with the basic blocks of his MoC. It is up to him to provide a single object to represent the elementary behavior of his MoC, leading to a basic block where the behavior may have to be defined by the SoC architect. This is done, for example, in the TDF MoC implemented within SystemC AMS or SystemC MDVP. The SoC architect has to specify and define himself the behavior associated with each TDF module he exploits in his design.

Another approach is to provide to the SoC architect a set of predefined built-in modules, where no characterization of the behavior is allowed. This is the case with the EN MoC for example. With this MoC comes a set of primitive modules such as capacitor, resistor or voltage source. The SoC architect is not allowed to specify the behavior associated with these primitives, he just instantiates them in his design.

As with the module, the development of the signal MoC-specific implementation is not restricted. The MoC architect can specify numerous communication interfaces, and signals if he needs. In regards of the communication interfaces, he can specify an input interface to handle the input from a port and an output interface to handle the output from a port. He can also adopt another approach and choose to define a single *inout* interface for both input and output from a port. In regards of communication channel, he can specify channels that behave like a pipe, a buffer, a FIFO. It is completely up to the MoC architect, depending on the purpose of his MoC.

Once again, the same applies with the ports where the MoC architect is left free to define all the ports he wants. However, logically he should provide ports for all the communication interfaces he defined if he intends to communicate with them. He can define specific input and output ports, or a single *inout* port for both input and output.

The implementation of the solving algorithm within the solver of the MoC is also completely left up to the MoC architect. One understand that the implementation details are irrelevant from the SystemC MDVP kernel viewpoint. Therefore, we provide the MoC architect with a good flexibility in order to conceive its MoC.

7.2.3 Interaction Step

The *Interaction step* represents the phase when the available interactions with the other MoCs, i.e. the master-slave semantics, are specified. This step can be divided into two sub-processes: the definition of interfaces to respect the master MoCs interface semantics, and the definition of converter ports to exchange data with the master MoCs. We follow the approach presented earlier that consists in inheritance principle in order to set up the interaction mechanism. In this section we assume the definition of the TDF MoC that follows the previous steps and the new MoC X is positioned as a slave of the TDF MoC.

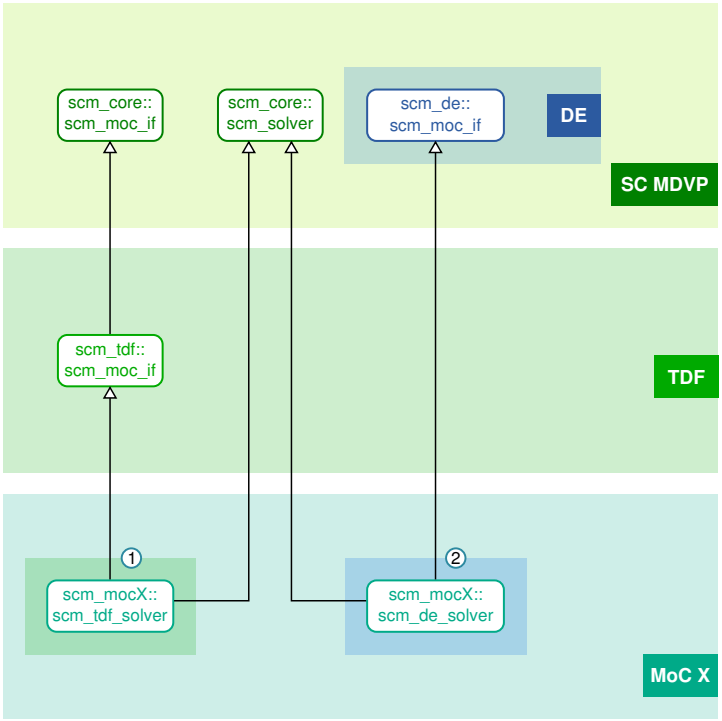

Figure 7.6: Inheritance pattern to define an interface solver within a new MoC.

Figure 7.6 represents the inheritance pattern that needs to be fulfilled in order to define a specific interface dedicated to each specific master MoC. In order to communicate with the TDF MoC the MoC X has to provide an interface that respects the TDF semantics. These semantics are defined within the MoC interface of TDF in the class `scm_tdf::scm_moc_if`. Therefore, a class `scm_mocX::scm_tdf_solver` which inherits from the TDF MoC interface (①) is defined. This approach guarantees that the master will be able to apprehend the slave MoC as one of its own components since all the functionalities it expects to find are provided. This interface dedicated to the master MoC represents a solver which respects the semantics of the master. As a solver, it has to inherit from the basic class `scm_core::scm_solver`.

In order to communicate with the DE MoC the MoC Architect should follow the same approach as for the communication with TDF. The notion of MoC interface is only defined within SystemC MDVP, and hence, SystemC does not define a MoC interface for its DE MoC. This is why the SystemC MDVP environment provides within its kernel a MoC interface for the MoC

DE. Therefore, a class `scm_mocX::scm_de_solver` which inherits from the DE MoC interface (②) can be defined. Exactly like the definition of the interface for TDF, this interface dedicated to DE represents a solver and, as such, has to inherit from the basic solver class.

More generally, a new slave MoC which desires to communicate with a master MoC must provide a `scm_slave::scm_master_moc_if` class that inherits from the class `scm_master::scm_moc_if` that defines the MoC interface of the master MoC . The MoC architect has to provide an interface for each master MoC he wants to communicate with.

Within these interfaces, the synchronization between MoCs is accomplished. It is left up to the MoC Architect to decide if each interface represents a specific solver, or if it represents an interface to manipulate an internal solver (which can be defined during the implementation phase). He can define a dedicated solver for each master MoC he wants to communicate with. While this approach may be necessary when interacting with MoCs that require a specific synchronization mechanism, we believe that an approach based on an internal solver represents a better solution when possible. Indeed, such approach centralizes the solving algorithm and eases the maintenance of the solver. It also allows for the decoupling of the synchronization from the resolution algorithm which is less error-prone.

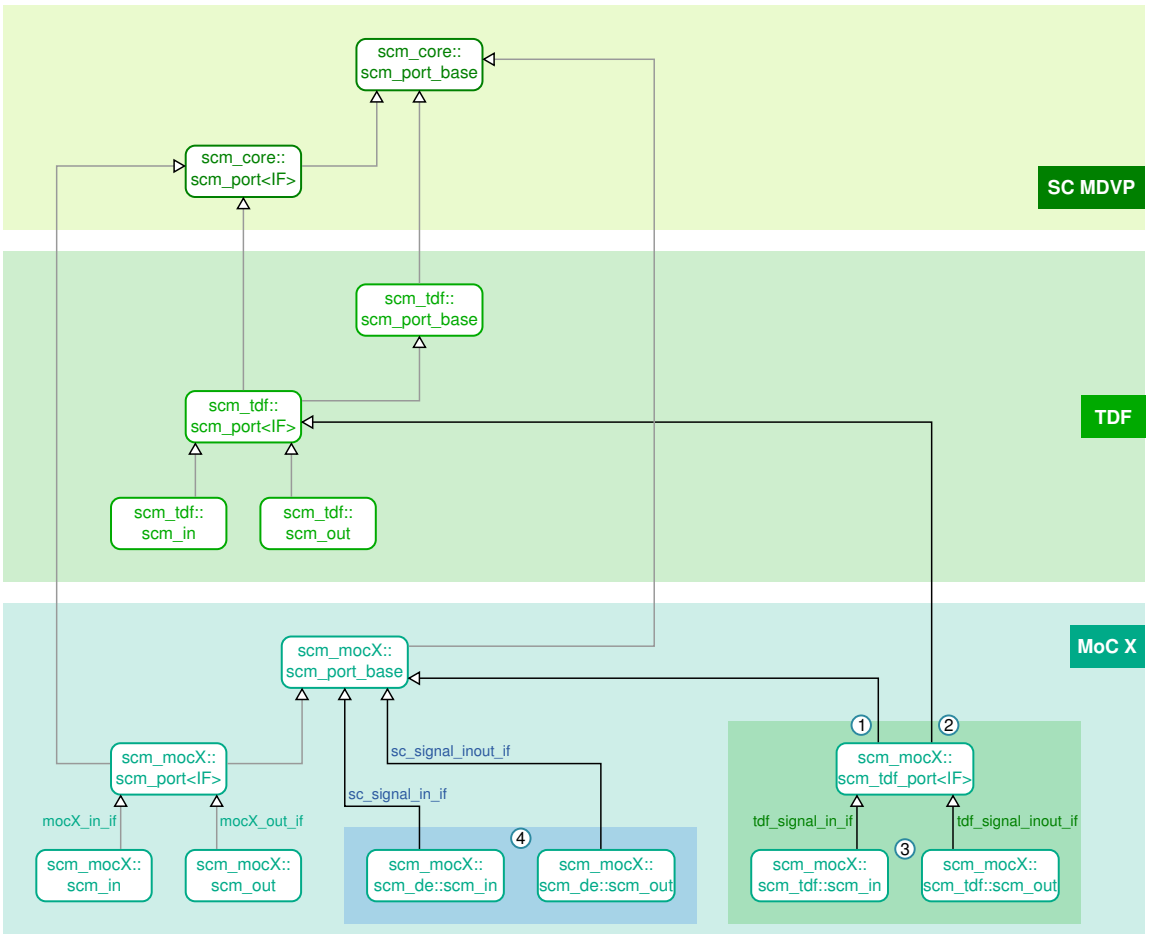

Figure 7.7: Inheritance pattern to define converter ports within a new MoC.

The second sub-process consists in defining the converter ports dedicated to each master MoC

the MoC Architect wants to communicate with and is illustrated in Figure 7.7. A converter port represents a port located at the border between two MoCs. Each MoC situated at this border should see the converter port as one of its regular port. Therefore, the converter port must express functionalities and semantics from both MoCs.

To this aim, the MoC Architect has to provide a class `scm_mocX::scm_tdf_port<IF>` that inherits from the class `scm_mocX::scm_port_base` (①). This guarantees that the functionalities and semantics present in his regular MoC's port are present in the converter port. In addition, he also has to inherit from the class `scm_tdf::scm_port<IF>` (②). This guarantees that the functionalities and semantics present in the regular master MoC's ports are present in the slave converter ports. The class `scm_mocX::scm_tdf_port<IF>` represents the basic class for all the converter ports dedicated to the communication with the master MoC TDF.

From now on, the MoC Architect can apply the same approach as for the definition of regular ports. The converter ports are represented by the classes `scm_mocX::scm_tdf::scm_in` and `scm_mocX::scm_tdf::scm_out` (③). He can define as many converter ports dedicated to the communication with TDF as he wants as long as they inherit from the class `scm_mocX::scm_tdf_port<IF>`. In addition of this inheritance, like regular ports, they also have to implement a specific communication interface in order to interact with communication channels. Thus, in order to communicate with a master communication channel, a converter port has to implement a master communication interface defined in the master MoC.

The methodology to follow in order to define converter ports dedicated to the communication with DE is quite similar to the one described but the new MoC X does not have to inherit from a *port base* class from the DE MoC. Indeed, the MoC Architect simply has to follow the same process that for the definition of regular ports except that he has to respect a communication interface belonging to SystemC. Therefore he defines two classes `scm_mocX::scm_de::scm_in` and `scm_mocX::scm_de::scm_out` (④) that represent input and output converter ports to DE. They respectively implement the interfaces `scm_signal_in_if` and `scm_signal_inout_if` provided by SystemC.

This approach allows the master MoC to see a converter port as one of its regular port. This way, the MoC X can exchange data seamlessly with a master MoC.

7.3 Application to SPH MoC

In order to support and verify the principles and mechanisms proposed within our framework, SystemC MDVP, we need a set of several Models of Computation to extend the simulation environment. Within this thesis, the development of a MoC that allows the description of fluidic networks has been achieved using the Smoothed Particle Hydrodynamics (SPH) theory [75, 76, 77, 78]. The choice to develop a fluidic MoC based on the SPH theory relies on the needs of the European Project Heterogeneous Inception in which this thesis is performed.

SPH is of substantial interest since it can simply take geometry, spatial derivatives and convincing fluidic behaviors into account. Hence it can provide a methodology which allows a user to compose and simulate a fluidic system without paying the price for a thorough finite elements description.

7.3.1 Interfacing Step

In essence, the SPH Model of Computation which is integrated into SystemC MDVP essentially mimics the primitive principles of the LSF (Linear Signal Flow) MoC from SystemC AMS. The aim of this approach is to provide predefined fluidic components to the end-user.

Following the previously described methodology, we create all the classes required by the SystemC MDVP framework. Since we want to provide to the SoC architect a set of predefined primitives, we represent elementary behaviors through the definition of multiple classes. Figure 7.8 lists the current primitives available within the SPH MoC.

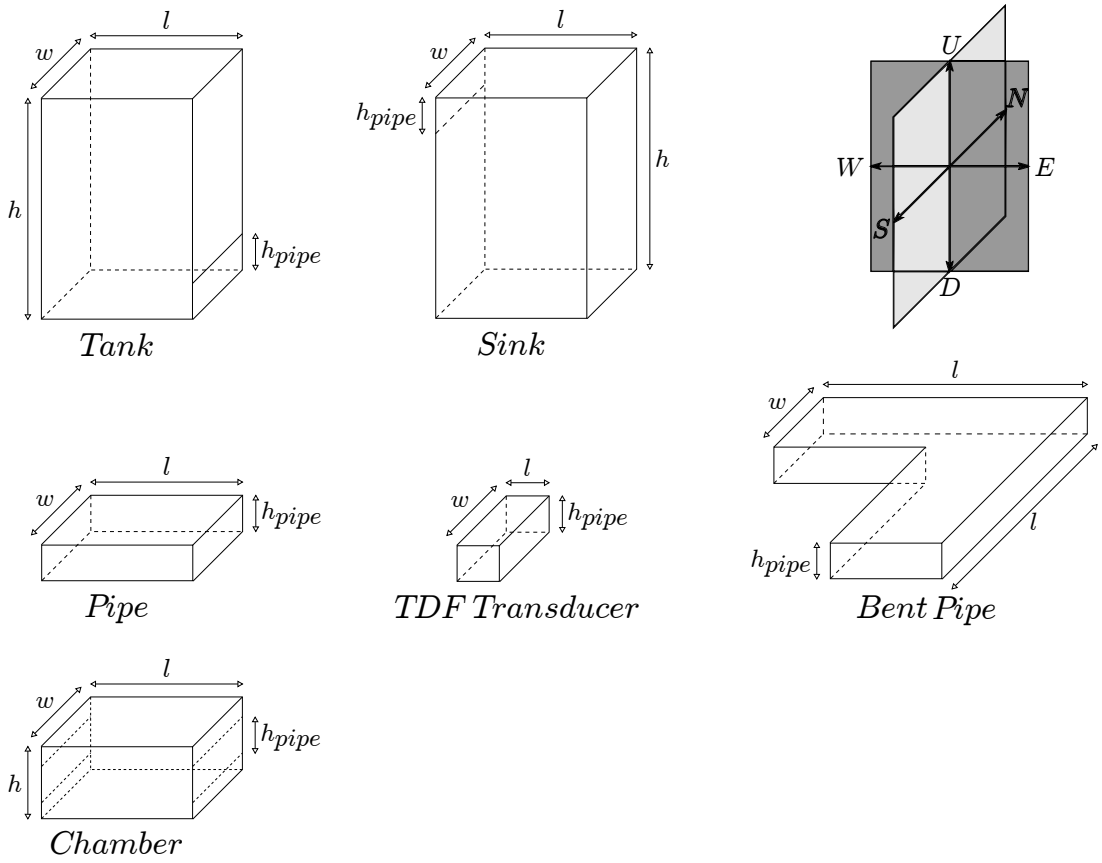

Figure 7.8: SPH modeling primitives.

The definition of communication channels within the MoC SPH is not really relevant. Indeed, our MoC does not use them in its intrinsic mechanism. Therefore, we only provide the basic requirements to meet the framework expectations. Communication channel will only be used by the SoC architect in order to link primitives altogether during the conception of the fluidic network. The same approach is followed regarding the composition mechanism and the definition

of SPH ports.

7.3.2 Implementation Step

In the SPH theory, the continuous fluid is replaced by a set of particles whose individual motion is approximated and which, therefore, possess individual properties such as density, pressure, velocity, etc. The particles move according to the governing conservation equations, i.e. a simplified version of the Navier-Stokes equation, in which the convective acceleration term is not considered.

$$A_i(\vec{r_i}) = \sum_j A_j \frac{m_j}{\rho_j} W(\vec{r_i} - \vec{r_j}, h) \tag{7.1}$$

The right hand side of Equation 7.1 represents the forces applied on a specific fluid particle. These forces can be divided into two categories: Internal Forces (such as Pressure, Viscosity and Surface Tension) and External Forces (Gravity, Magnetic Fields, etc.). All the forces involved in the SPH algorithm are expressed as density forces. The solving algorithm at the very heart of SPH aims at computing the forces applied on a particle i to find the acceleration and by integration the position and velocity of this particle. Any SPH quantity (density, force) for a particle i can be determined using Equation 7.2.

$$A_i(\vec{r_i}) = \sum_j A_j \frac{m_j}{\rho_j} W(\vec{r_i} - \vec{r_j}, h) \tag{7.2}$$

In Equation 7.2 h represents the support radius (the distance of interaction of a particle with others) and W represents the Smoothing Kernel, which is used to weight the approximated implication of a particle j in the calculation of a quantity for particle i according to their Euclidean distance. $\vec{r}$ represents the position of a particle. The support radius and the Smoothing Kernel are illustrated in Figure 7.9.

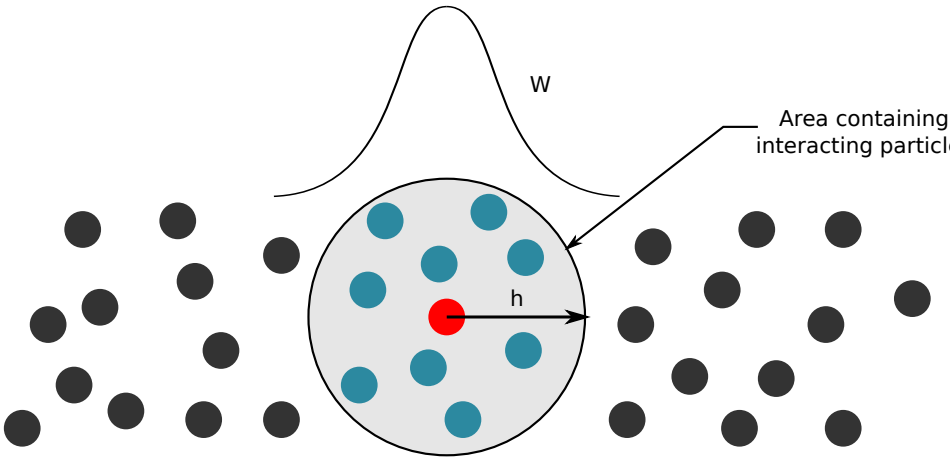

Figure 7.9: Smoothing Kernel and support radius in SPH.

Applying Equation 7.2 to the density computation leads to Equation 7.3. The density of particle i represents the density of the area covered by the smoothing length.

$$\rho_i(\vec{r_i}) = \sum_j m_j W(\vec{r_i} - \vec{r_j}, h) \tag{7.3}$$

The generic Equation 7.2 is also used to compute forces, but it has to be slightly modified. For instance, the modification needed to compute the pressure force while respecting the reciprocity principle of Newton Law implies Equation 7.4.

$$\vec{F_{P_i}}(\vec{r_i}) = -\rho_i \sum_{j \neq i} m_j (\frac{P_i}{{\rho_i}^2} + \frac{P_j}{{\rho_j}^2}) \nabla W(\vec{r_i} - \vec{r_j}, h) \tag{7.4}$$

To compute the pressure P, a modification of the perfect gas equation "$P_i = k(\rho_i - \rho_0)$" is used, where ρ_0 represents the rest density of the fluid described by the set of particles and ρ_i the density computes at the position of the particle i. The integration of the rest density allows the representation of repulsion and attraction processes. A low density leads to a negative pressure, which induces on neighboring particles an attractive force. On the contrary, a high density causes a positive pressure leading to a repulsive force.

Along with the viscosity force, we take into account the relative velocity between particle i and its neighbors in Equation 7.5, where η represents the viscosity coefficient.

$$\vec{F_{V_i}}(\vec{r_i}) = \sum_{j \neq i} \eta \frac{m_j}{\rho_j} (\vec{v_j} - \vec{v_i}) \nabla^2 W(\vec{r_i} - \vec{r_j}, h) \tag{7.5}$$

The force associated to the surface tension needs to meet some requirements. This force acts at the surface of the fluid and at the interface between two fluids, e.g. the interface between water and air. Since this force makes sense only at the surface, only particles which limit this surface should be affected. Several equations are used to achieve this purpose: Equation 7.6, Equation 7.7, Equation 7.8 and Equation 7.9.

$$\vec{C_{s_i}}(\vec{r_i}) = \sum_{j \neq i} \frac{m_j}{\rho_j} W(\vec{r_i} - \vec{r_j}, h) \tag{7.6}$$

$$\vec{n_i} = \nabla C_{s_i} \tag{7.7}$$

$$\vec{K_i} = -\frac{\nabla^2 C_{s_i}}{|\vec{n_i}|} \tag{7.8}$$

$$\vec{F_{s_i}}(\vec{r_i}) = \sigma \vec{K_i} \vec{n_i} = \sigma(\nabla^2 C_{s_i}) \frac{\vec{n_i}}{|\vec{n_i}|} \tag{7.9}$$

C_s identifies the particles at the surface of the fluid. The gradient of this quantity for a particle allows us to determine if it should be affected by the force. If the length of this gradient is greater than some threshold (defined according to the fluid simulated) the particle should be affected by the surface tension. The last force implied in the computation is the gravity, which is simply defined as $\vec{F_{g_i}} = \rho_i \vec{g_i}$.

With all the forces defined, we can compute the acceleration, in a straightforward way as $\vec{a_i} = \sum \frac{\vec{F_i}}{\rho_i}$. Since we are using density forces, the division is performed on density and not mass.

From the solver viewpoint, SPH simulation is performed in four straightforward steps. The first step does the computation of density (other quantities rely on the density value of each particle). The second calculates the internal and external forces applied to each particle, and the third uses Newton's law to determine the acceleration. Finally, with an Euler, Leapfrog or Verlet scheme [79], acceleration is integrated twice to give the new position of each particle. This simulation loop is illustrated in Algorithm 7.1.

```
1 Function Simulation_loop()
2 |   Update_density_pressure();
3 |   Update_forces();
4 |   Update_acceleration();
5 |   Update_position_velocity();
6 end
```

Algorithm 7.1: SPH Simulation Loop.

To describe any micro-fluidic network appropriately, the SPH algorithm must also consider the geometrical aspects. Every particle has to interact with the environment (pipe, tank, etc.), and hence we need to take into account collisions of particles with these solid elements. Therefore, we introduce an algorithm to detect when a particle is colliding with its container. In the current implementation, we used an "a posteriori" detection mechanism, which means that the collision is actually detected after it occurred. In the previously described Algorithm 7.1, after updating the particle's position, we check against the containers if a collision occurred and if so the particle is moved and a correction on its velocity is applied according to the angle of the collision and the restitution coefficient of the container.

To reduce the overall computational cost some optimization can be done. As all the quantities related to a particle i depend on the quantities of the neighboring particles, a dynamically managed grid can be used to contain sets of particles (all particles are stored into cells whose width is twice the smoothing length). Thanks to this structure, the search for neighboring particles of a particle i consists in looking only at the cells adjacent to the cell which contains the particle i (maximum 9 cells in 2D and 27 cells in 3D). Second, referring to the third law of Newton, one can directly take into account the implication of particle i to particle j when we evaluate the

implication of particle j to particle i (reciprocity principle). This optimization is considered in the Algorithm 7.1, a step before computing the density is added; it aims to distribute all the particles inside the grid according to their specific positions.

To each geometric primitive matches a C++ constructor method that takes three sets of parameters. The first set of parameters represents the geometric bounding box of the SPH component (expressed as length, width and height). The second set of parameters defines the amount of fluid that is already present in the geometric element when the simulation starts. Finally, the last parameter allows for the specification of the orientation of a specific output port of the primitive, opening the way to the building of any 3D topology of the fluidic network. This solution greatly simplifies the connections of upstream and downstream elements and the global design. Listing 7.1 shows a simple fluidic netlist composed of a tank, a pipe, two bent pipes instances and a sink.

```
#include <sc_mdvp.h>
[...]
#include <sph.h>
int main(){
  [...]
  scm_sph::sph_tank t1(l=3000, w=3000, h=5000, hpipe=1000, Fluid(), OUT_EAST) ;
  scm_sph::sph_pipe p1(l=10000, w=3000, h=1000, NULL) ;
  scm_sph::sph_bentpipe bp1(l=2000, w=3000, h=1000, NULL, OUT_SOUTH) ;
  scm_sph::sph_bentpipe bp2(l=2000, w=3000, h=1000, NULL, OUT_EAST) ;
  scm_sph::sph_sink s1(l=3000, w=3000, h=5000, hpipe=1000, NULL) ;

  scm_sph::sph_signal t1_p1;
  scm_sph::sph_signal p1_bp1;
  scm_sph::sph_signal bp1_bp2;
  scm_sph::sph_signal bp2_s1;

  t1.out(t1_p1);
  p1.in(t1_p1);
  p1.out(p1_bp1);
  bp1.in(p1_bp1);
  bp1.out(bp1_bp2);
  bp2.in(bp1_bp2);
  bp2.out(bp2_s1);
  s1.in(bp2_s1)
  [...]
}
```

Listing 7.1: Code snippet of SPH primitives composition

After the SPH netlist has been elaborated, the simulator elaboration phase converts the individual geometric volumes associated to each SPH primitive into a global complex 3D volume that will act as the SPH simulation envelope. All the SPH simulation process is performed in this 3D envelope.

7.3.3 Interaction Step

Concerning the SPH MoC the interaction step is quite straightforward. We simply follow the methodology in order to define a converter port to the TDF MoC as well as a MoC interface compliant with the TDF MoC interface. Since the SPH MoC relies on pre-defined primitive modules, we also define a primitive module which implements this port - the SPH-TDF transducer.

The SPH-TDF transducer can be used to detect the presence of specific particles in a given volume, as illustrated in Figure 7.10. These particles are referenced, and a TDF-compatible scalar value is generated according to the ratio of counted particles over the volume.

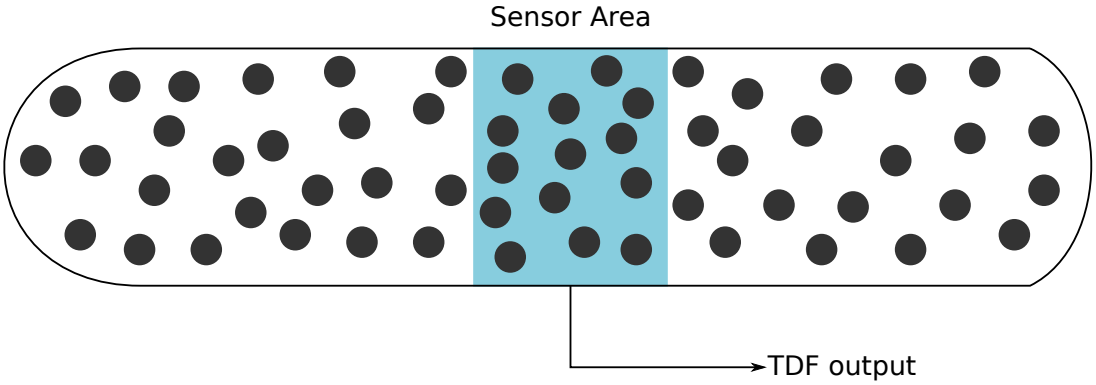

Figure 7.10: SPH – TDF transducer.

7.4 Conclusion

Multi-physical systems represent non homogeneous applications that involve several different entities. In our approach to enable the virtual prototyping of multi-disciplinary systems we chose to represent these different entities, involved in the systems' conception, through the notion of Model of Computation. Consequently, such virtual prototyping environment has to be flexible in order to handle the possible MoCs. In this chapter we introduced the methodology a MoC architect has to follow in order to enrich our framework with new Models of Computation. A clear definition of a MoC by means of our MoC abstraction results in an easy, straightforward integration process within our environment.

The integration methodology consists in three steps, leading to the definition and the integration of a new MoC within SystemC MDVP. Based on inheritance mechanism, the first step called *Interfacing Step* allows a MoC architect to define the basic of its MoC while respecting the requirements of SystemC MDVP. Meeting the expectation of SystemC MDVP through the respect of its MoC abstraction allows for the automatic handling of the new upcoming MoC by the framework. This approach ensures the flexibility of our virtual prototyping environment.

The second step called *Implementation Step* allows a MoC architect to define the intrinsic of its MoC. We illustrated that a simulation kernel that does not rely on the definition of MoCs allows

a MoC architect to freely develop its MoC with as few constraints as possible. The definition of elementary behavior, communication channel, composition mechanism and solving algorithm are completely left up to the MoC architect without dependencies on SystemC MDVP(except for the MoC abstraction). Again, this enhances the flexibility of our framework.

Eventually, the last step, called *Interaction Step*, allows a MoC architect to define the interaction with other MoCs he considers. He defines the master-slave semantics that its MoC will follow, i.e. all the authorized interactions. The definition of these interactions also relies on inheritance mechanism and leverages the MoC abstraction defined by SystemC MDVP. This approach allows for the automatic handling of interaction between MoCs, and hence contributes to the flexibility of SystemC MDVP.

To support this methodology we presented a new Model of Computation - Smoothed Particle Hydrodynamics (SPH). SPH allows for the description of fluidic network. The principle is to replace a continuous fluid by a set of interacting particles. We followed the three steps in order to integrate the SPH MoC within SystemC MDVP. The singularities of this MoC illustrate the flexibility of our framework that we claimed. Indeed, this MoC does not make use of ports or channels in its intrinsic mechanism and it represents a 3D oriented-systems based on predefined primitives and yet the integration process within SystemC MDVP is straightforward. The complete inheritance diagram of the MoC SPH is described in Figure 7.11.

The integration methodology that we presented tackles the challenge of providing a flexible virtual prototyping environment. It easily allows for the increase of the set of heterogeneous entities, and hence opens the way to the integration of MoCs associated with different physical domains.

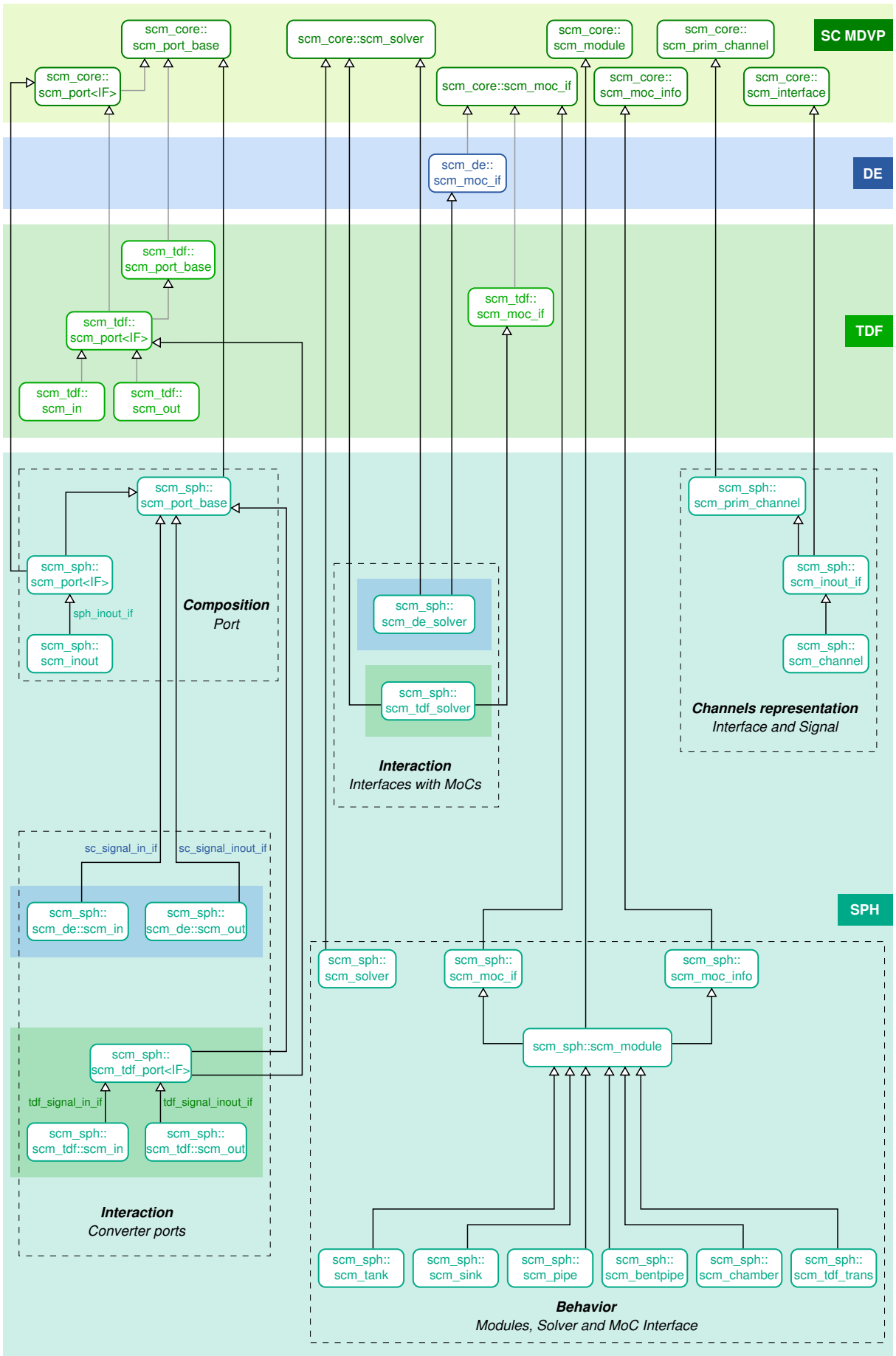

Figure 7.11: SPH MoC inheritance diagram.

Chapter

8 *Validation Case Studies*

Contents

8.1 Introduction 118

8.2 Application Based on SPH 118

8.2.1 Case Study Description 118

8.2.2 Results 120

8.2.2.1 Constant rate 121

8.2.2.2 Different pressures 122

8.2.2.3 Different viscosities 123

8.2.2.4 Performance comparison with FEM 124

8.3 Application based on RFID 124

8.3.1 Case Study Description 124

8.3.1.1 A Passive RFID reading system 125

8.3.1.2 Detailed Principles 126

8.3.1.3 Modeling of the RFID system 126

8.3.2 Results 129

8.4 Conclusion 135

8.1 Introduction

In the previous chapters we introduced our virtual prototyping environment SystemC MDVP: the principles on which it relies and the implementation details that support these principles. This chapter, in turn, introduces case studies which aim to illustrate the possibilities offered by our framework. We present application examples supported by our virtual prototyping environment.

In Section 8.2, the first case study dedicated to the Model of Computation (MoC) Smoothed Particle Hydrodynamics (SPH) is presented. It involves the conception of a fluidic network that aims to represent the fluidic component of a point-of-care blood analysis system. The simulation and the results are compared between several solutions.

In Section 8.3, a second case study is introduced. This case study describes a Radio Frequency Identification (RFID) device - more precisely a passive RFID reading system. This case study aims to illustrate the interactions between several MoCs within SystemC MDVP.

Eventually, Section 8.4 closes this chapter and brings an overview of the presented case studies.

8.2 Application Based on SPH

8.2.1 Case Study Description

One of the proof-of-concept multi-domain applications that are modeled in the European project H-Inception is a prototype of a point-of-care blood analysis system which includes a micro-fluidic subsystem. The analysis procedure is controlled by a micro-controller which activates pistons and valves in order to move and mix the blood samples with several reagents that imply several biochemical processes. The final biochemical reaction is electrically monitored with an AMS device, digitally converted and sent back for characterization. In order to evaluate the proposed modeling schemes, a real prototype is tested and its timing behavior is compared against several approaches: Poiseuille Fluidic Networks (PFN) and Smoothed Particle Hydrodynamics (SPH).

The Hagen-Poiseuille law allows for the modeling of the behavior of a fluid within a container. To do so, the system modeled has to meet few requirements: it must describe a steady laminar, incompressible, Newtonian (constant viscosity) and quasi-unidirectional flow. The Hagen-Poiseuille equation Equation 8.1 gives the pressure drop of such fluid through a container traversal.

$$Q = \frac{\Delta p}{R_h} \tag{8.1}$$

This equation describes the volumetric flow rate Q (in microliter/second for instance) as a

linear variation with the gradient of pressure Δp (pressure drop). R_h represents the hydraulic resistance, its definition varies depending on the shape of the container (rectangular, cylindrical, etc.). It takes into account the length, the radius and the diameter of the container. It also relies on the velocity and the viscosity of the fluid.

Equation 8.1 is a simple linear equation that fits quite well the purpose of modeling micro-fluidic behavior. Indeed, if this equation if applied on system described with container's diameter above a threshold the fluid becomes turbulent and the pressure drop is no longer accurate.

Equation 8.1 represents the hydraulic equivalent of the Ohm's law in electrical circuit with the pressure acting as the voltage and the volumetric flow rates acting as the current. Therefore, the behavior of a fluid described with the Hagen-Poiseuille law can be described using the Ohm's law using the same algebraic equations and solving methods that are used in electrical linear networks (which apply the Kirchhoff equations). This is the approach followed in the PFN MoC. It is implemented using the TDF and ELN MoCs from SystemC AMS.

Moreover, in addition of the PFN and SPH simulation results we also do a comparison with results provided by a typical Finite Element Method (FEM) simulation framework. This work was the subject of a journal paper [80] and the following of this section is mainly derived from this paper.

The real prototype was designed to include five different ports which can be used as inlets or outlets, and two micro-chambers. This configuration is frequently used for diagnostic applications, using the first micro-chamber for sample concentration, mixing and purification. The second chamber usually includes electrodes or micro-sensors to finally detect molecules of interest.

The micro-fluidic real prototype was made of COP by lamination techniques. First, two 188 mm thick sheets were first structured by a cutting-blade to obtain the desired configuration of micro-channels and micro-chambers. Another two COP layers were then structured, one to be used as bottom layer and the other with included holes at the port locations to be used as inlets and outlets. Layers were aligned and bonded together using *Pressure Sensitive Adhesives* (PSA).

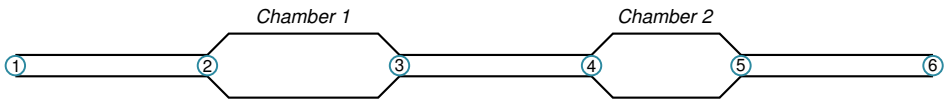

Figure 8.1: Micro-fluidic Network.

Figure 8.1 represents the fluidic network modeled in this case study. In order to perform timing behavior analysis on the simulated model we choose six position of interest (represented by ①, ②, ③, ④, ⑤ and ⑥). The fluid is injected in the left entry point (①) and collected in the right exit point (⑥). The time required by the fluid to reach each point of the model is registered, and then compared between the different approaches under different physical conditions.

Next, different experiments and comparison results are broken down.

8.2.2 Results

The results presented in this section represent comparison of the different approaches (PFN and SPH) against the experimental results obtained from the real prototype. These comparisons are performed under different physical conditions. First we experiment at a constant flow, second we change the pressure and finally we modify the viscosity. The idea is to inject a fluid within the system and to observe when the injected fluid completely replaced the fluid initially present in the network. A final comparison is provided against a FEM model.

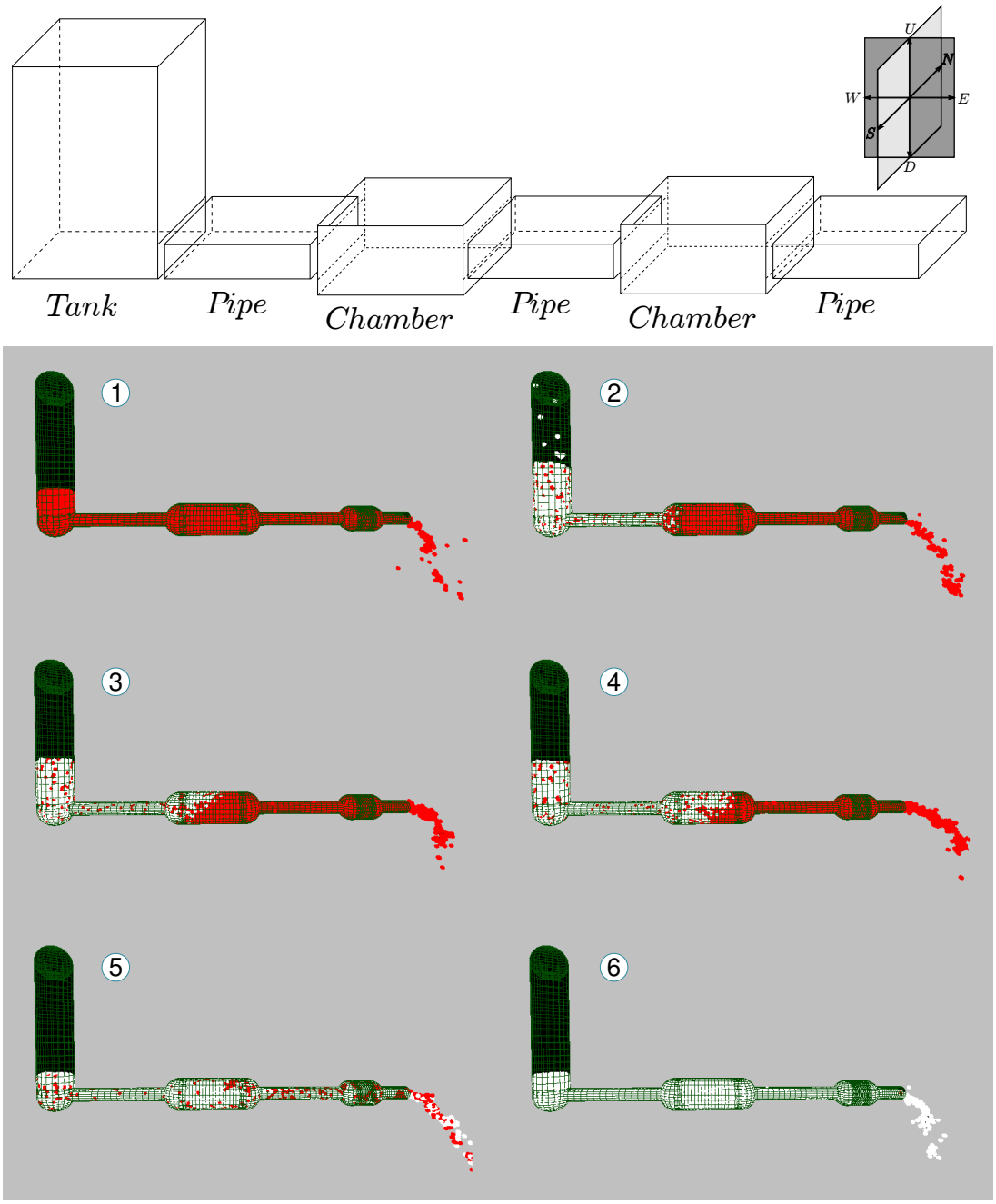

Figure 8.2: SPH micro-fluidic chip.

Figure 8.2 shows the schematic representation of the fluidic network presented in Figure 8.1 using the predefined primitives from the SPH MoC.

Figure 8.2 presents the SPH representation of the fluidic network of Figure 8.2, at various simulation stages. A complementary tank has been connected to the entry point 1 in order to realize the different experiences presented here. The system is initially filled with red particles, then white particles are injected in the left entry point. We can observe the progression of the white particles along the fluidic network until they completely replaced the red particles.

SPH is intrinsically very sensitive to simulation parameters. Therefore, a strong process of tuning was carried out in order to match the simulation results with real experiments.

3D visualization is obtained by means of a classic `C++ OpenGL` rendering engine which represents fluidic primitives either as parallelepiped-shape structures (Cartesian coordinates) or spheres and cylinders (Spherical coordinates). The camera and look at 3D points can be modified by the end-user to identify in the 3D fluidic network a region of interest and to be able to zoom on it.

8.2.2.1 Constant rate

In order to impose a constant liquid flow to the real prototype micro-fluidic network, we pushed the plunger of the syringe placed at the inlet (Figure 8.1, ①) at a constant speed. The experiment was carried with water at a flow rate of 0.014 ml/min. To visualize the progression of the fluid within the fluidic network we inserted a mixture of Rhodamine B with water in the system which is characterized by a specific blue color. First, the fluidic network was completely filled with water, then the Rhodamine B was injected, and the advance of the characteristic blue color was optically observed by a microscope.

The times at which the Rhodamine B reached the different fluidic ports are shown in Figure 8.3, and they are compared with the results obtained by the simulation.

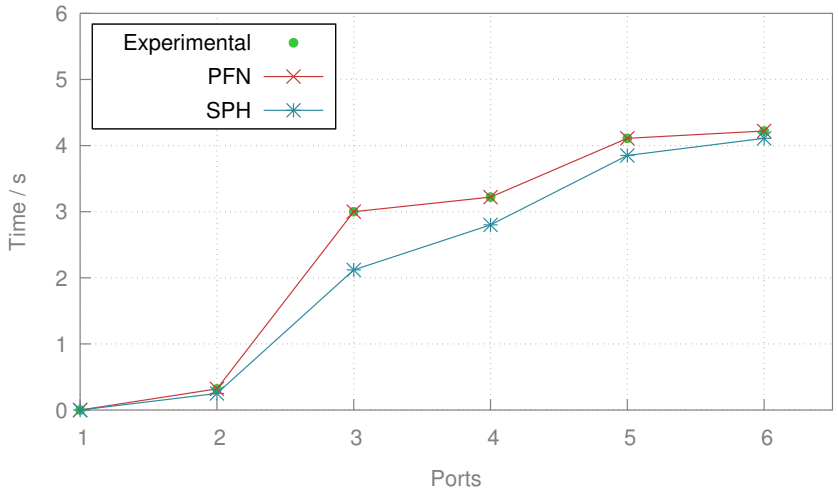

Figure 8.3: Experimental, PFN and SPH results for constant flow.

The PFN simulation matches the experimental results with a lot of accuracy. In the SPH simulation, maintaining a constant flow of particles with a constant pressure at the input of the

fluidic network is hard to obtain in practice. Even, with a timing difference between SPH and the experimental results, we can see a global behavior of the fluid within the SPH simulation that follows the experimental fluid behavior. Nevertheless, the PFN and SPH results coincide in an acceptable way.

8.2.2.2 Different pressures

We are interested in observing the behavior of a fluid when it is subject to different pressure. In order to impose and modify the pressure within the real prototype we used an external pressure source, represented by a tank. The difference in height between the tank level and the outlet allows for the specification of a pressure at the input of the fluidic network. Two different heights were used to check the simulation results, resulting on an experience with a pressure of $200Pa$ and another one with a pressure of $400Pa$.

As for the first experience at constant flow, we used a tank filled with a mix of Rhodamine B and water to visualize the flow within the network. Once the micro-fluidic network was filled with water, the tank filled with a mixture of water and Rhodamine B was connected, and the flow of the colored liquid was observed by a microscope.

As a result, the elapsed times obtained to reach the different ports were experimentally obtained. Results are shown in Figure 8.4 and Figure 8.5, and compared with simulation results.

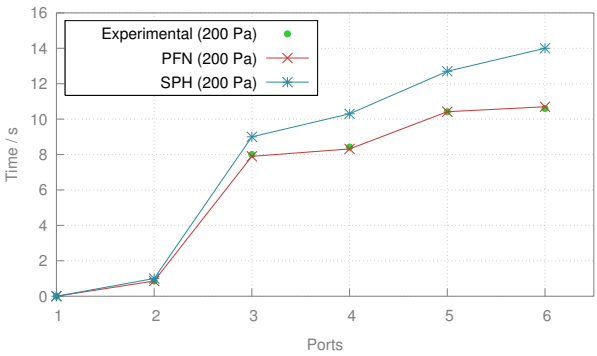

Figure 8.4: Experimental, PFN and SPH results for pressure at 200 Pa.

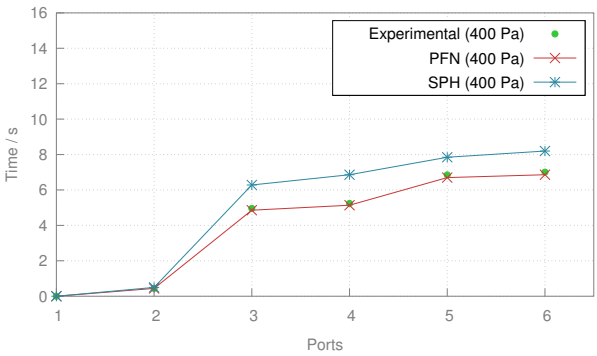

Figure 8.5: Experimental, PFN and SPH results for pressure at 400 Pa.

Again, the PFN simulation is very accurate, matching with the experimental results both at $200Pa$ and at $400Pa$. The SPH simulation is clearly less accurate and the difficulties highlighted during the first experience are still present during these experiences. However, we can nevertheless notice that the SPH simulation behaves qualitatively. The modification of pressure from $200Pa$ to $400Pa$ leads to increase the speed of the fluid within the network for both experimental and SPH. The experimental fluid is approximatively 33% faster when the pressure is augmented, and the SPH fluid is approximatively 40% faster. It keeps in the order of magnitude of experiments and it keeps the global behavior following the experimental one.

8.2.2.3 Different viscosities

We are now interested in observing the behavior of a fluid when it is subject to different viscosity. We used methanol instead of water within the real prototype in order to perform an experience with a different viscosity (590 Pa s). We used the same experiment condition from the previous experience except that we do not want to vary the pressure this time. Consequently, the experience was done in $400Pa$ only.

In order to visualize the fluid we used Rhodamine B in the previous experiences, this time we switch to Erythrosin B. We change the coloring solution since the Erythrosin B dissolves much better in methanol than Rhodamine B. Results can be observed in Figure 8.6, and compared with the results obtained by simulation.

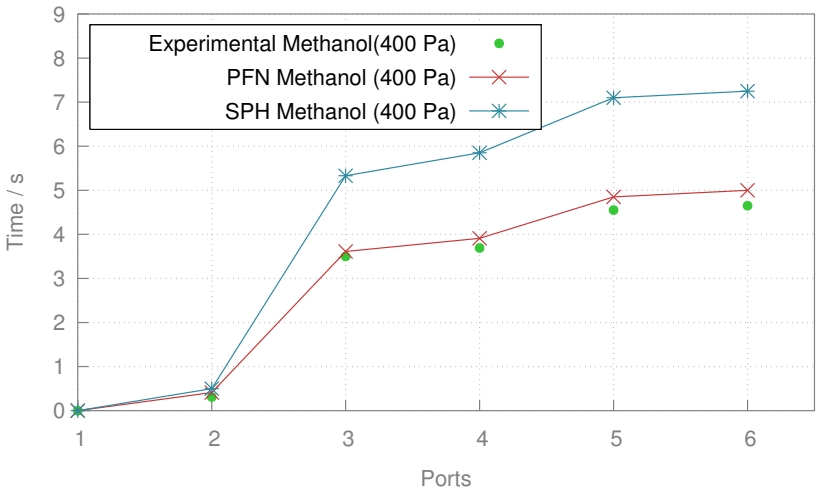

Figure 8.6: Experimental, PFN and SPH results for Methanol at 400 Pa.

PFN results were very accurate although they were not as accurate as in previous experiences. A maximum discrepancy of 9% was observed for methanol at the fifth micro fluidic observing point. The SPH results still express the same difficulties to be extremely accurate, but once again the global behavior of the fluid follows the behavior of the experimental fluid.

The SPH MoC is very sensitive to the simulation parameters and can be tricky to correctly tune. The results obtained during these experiences can be explained by the difficulties to perfectly tune all the parameters of the SPH MoC. The model itself can justify in part the results, the particle model suffers from a small compression issue that can have an incidence on the results. Finally, in SPH the environment is taken into account, and hence there are interactions with the containers defining the fluidic network. The collision handling defined within the SPH MoC can also explain in part the results obtained.

8.2.2.4 Performance comparison with FEM

We have performed comparison between models simulated against experimental results obtained on a real prototype in order to highlight the acceptable accuracy we are able to reach with a virtual prototyping environment. We now, want to compare the performance of our tools against an existing solution.

Therefore, we compare the performance between the approach that uses SystemC AMS (the PFN), the approach that uses SystemC MDVP (SPH) and another approach that relies on FEM implementation (ANSYS CFX [81]).

Table 8.1: Modeling and Simulation times.

Tool	Modeling Time (s)	Simulation Time (s)
ANCYS CFX	2700	1800
PFN	600	10
SPH	650	30

Table 8.1 contains the modeling and simulation times for each approach studied here. The modeling time represents an estimation of the time required by a SoC Architect to design the presented micro-fluidic network following each of the approach. The simulation time represents the time the tools take to perform the simulation (once the system described, this is pure runtime). The presented data are rounded since they only express an order of magnitude.

In regards of the modeling time, both approaches relying on SystemC AMS and SystemC MDVP are clearly faster than the approach based on FEM (five times faster). The difference between the SystemC AMS and SystemC MDVP approach is not relevant, and hence are considered similar.

In regards of the simulation time, the difference between the FEM approach and those based on SystemC AMS and SystemC MDVP is even bigger. The approach that relies on SystemC AMS is one hundred and eighty times faster than the FEM approach, while the approach based on SystemC MDVP is sixty times faster. Such a difference between the SystemC AMS and SystemC MDVP approaches can be explain in part by the fact that the SPH simulation provides a 3D rendering of the system during the whole simulation.

8.3 Application based on RFID

8.3.1 Case Study Description

We present a complete case study that aims to validate the modeling and simulation principles developed previously. The interest of this case study lies in the fact that it involves three

distinct Models of Computation: Discrete Event (DE), Timed Data Flow (TDF) and Electrical Network (EN) and it is both simple and realistic. Moreover, a physical implementation, from which real measures can be extracted, is available. This allows us to compare the monitored simulation values with real values.

8.3.1.1 A Passive RFID reading system

In the RFID domain there exists several different technology to construct a RFID system. These designs are usually classified as *active* or *passive* [82, 83]. A standard RFID reading system is composed of two parts:

- *A tag* also named transponder, which carries a unique identification marker composed of 8 bytes.
- *A tag reading interface*, also named reader or transceiver.

An active RFID tag has an on-board battery and transmits its identification marker periodically, its local power source allows it to operate at long distance from a RFID reader. A passive RFID tag has no on-board battery and hence, relies on the reader to provide power supply. There also exists an in-the-middle approach where the system is described as a battery-assisted passive RFID tag.

In this case study we focus on a passive RFID reading system as described in Figure 8.7. The transceiver temporarily supplies the tag with electric energy thanks to the radio energy it transmits. Then, it serially reads the tag identification markers.

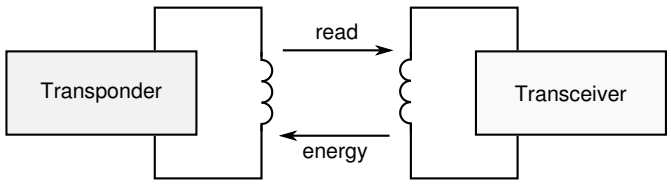

Figure 8.7: A passive RFID system.

The transceiver and the transponder work together as a contactless, wireless and coupled electromagnetic communication system. Both parts operate at a low frequency (125 KHz) and contain an inductor (a coil). When the transceiver primary coil and the transponder secondary coil are physically near (1 to 5 centimeters), they act as a transformer (except that there is no iron core but thin air between coils). These coils are tightly tuned to maximize the global resonance factor of the system.

The transceiver generates an oscillating electromagnetic field that momentarily provides energy to the transponder, allowing the tag to power up and its digital subsystem to emit the serial tag value. The great idea behind RFID is that the transponder uses the bitstream tag value

to modulate the transformer characteristics requesting more or less power from the transceiver according to the transmitted bit. In other terms, when the transponder needs to send a logic value '0', it modifies its power supply line to get more power from the transceiver. Consequently, on the transceiver side a noticeable voltage drop is observed, it can then be used in an aim to distinguish the logical values transmitted, and hence reconstruct the tag bitstream.

8.3.1.2 Detailed Principles

The real circuit, modeled in the case study, has been designed and sized by Francis Bras from UPMC. It is used in many Electrical Engineering and Computer Sciences courses, especially with problem-based curricula. A simple representation of the circuit is depicted in Figure 8.8.

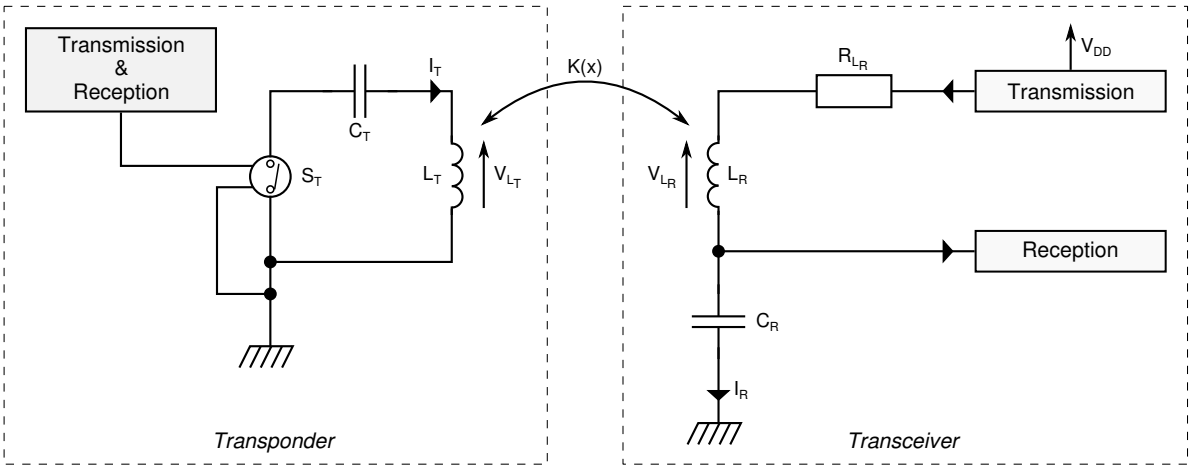

Figure 8.8: Simple representation of the RFID system modeled.

The key point of the presented RFID system lies in the transformer which is composed of the coil L_R (on the transceiver side) and the coil L_T (on the transponder side). The distance between the two coils physically modifies the mutual induction factor $K(x)$. On the Figure 8.8 L_R and C_R constitute a 125 KHz sine oscillator that is triggered by the digital transmission subsystem.

8.3.1.3 Modeling of the RFID system

We present in Figure 8.9 a detailed representation of the electrical circuit that constitutes the transceiver. The system modeled in this case study follows this representation of the electrical circuit; the parameters values of each electrical component are presented in Table 8.2.

Table 8.2: Parameters values associated with the circuit shown in Figure 8.9.

L_R	C_R	C_1	C_2	C_3	R_{L_R}	R_1	R_2	R_3	R_4	R_5
1.5mH	1.6nF	270pF	47pF	8nF	39Ω	15kΩ	230kΩ	47kΩ	1MΩ	1MΩ

Figure 8.10 depicts a detailed representation of the electrical circuit that constitutes the

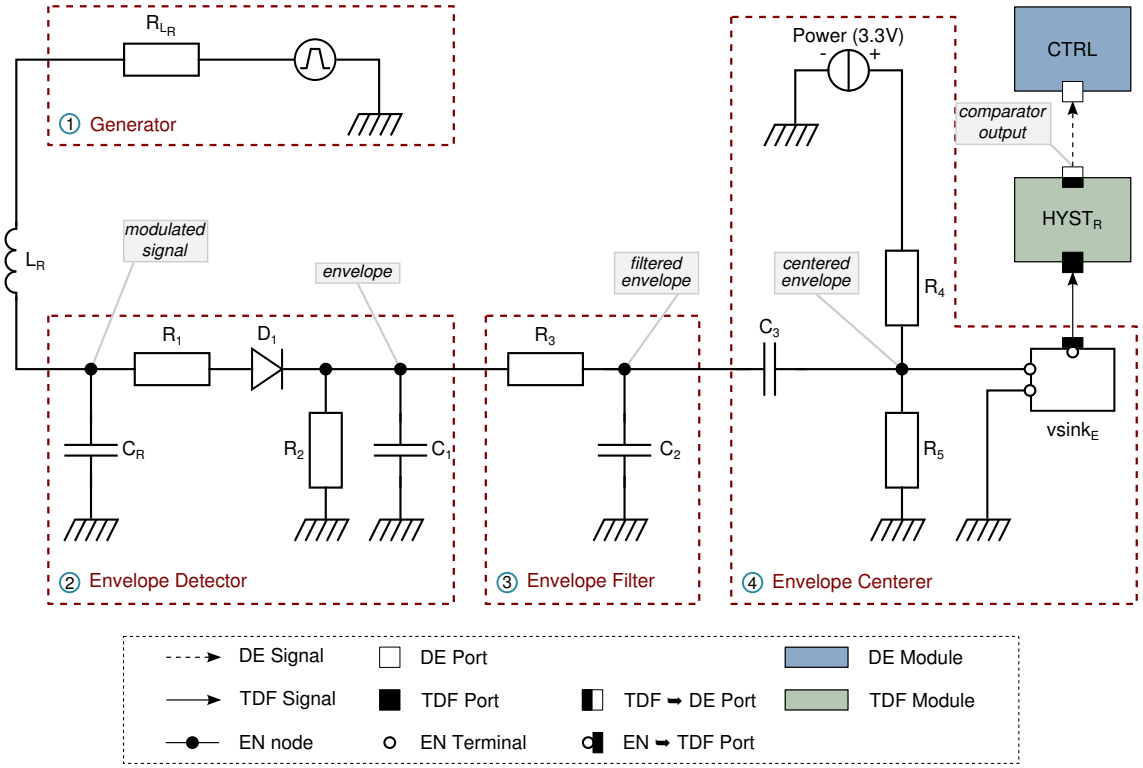

Figure 8.9: Detailed representation of the RFID transceiver modeled.

transponder and the parameter values of each electrical component are introduced in Table 8.3

Table 8.3: Parameters values associated with the circuit shown in Figure 8.10.

L_T	C_T	C_P	R_C	R_P
0.5mH	1.5nF	1nF	100Ω	10kΩ

While we previously only detailed the electrical circuit modeled using the EN MoC, the RFID system is modeled using three different MoCs. Figure 8.11 describes the model representation of the whole passive RFID reading system, including details about the components modeled using other MoCs. It includes the components modeled with the TDF and the DE MoCs. The details of all the components involved in the model of the RFID system are given below.

The transceiver is designed with a transmission chain and a reception chain connected to the primary coil L_R. The transceiver's transmission chain is composed of a generator, *GEN*, (Figure 8.9 ①) described with the EN MoC.

The transceiver's reception circuit corresponds to an amplitude demodulator that aims to retrieve the information transmitted by the transponder. The demodulation chain is composed of an envelope detector (Figure 8.9 ②), an envelope filter (Figure 8.9 ③) and an envelope center (Figure 8.9 ④). The envelope detector, *DETECT*, aims to straighten the signal received, then removing its negative component as well as the carrier that supported the information transmitted. The envelope detector provides the envelope of the signal transmitted. Then this envelope goes through the envelope filter, *FILT*, which consists of a low pass filter, and the envelope center,

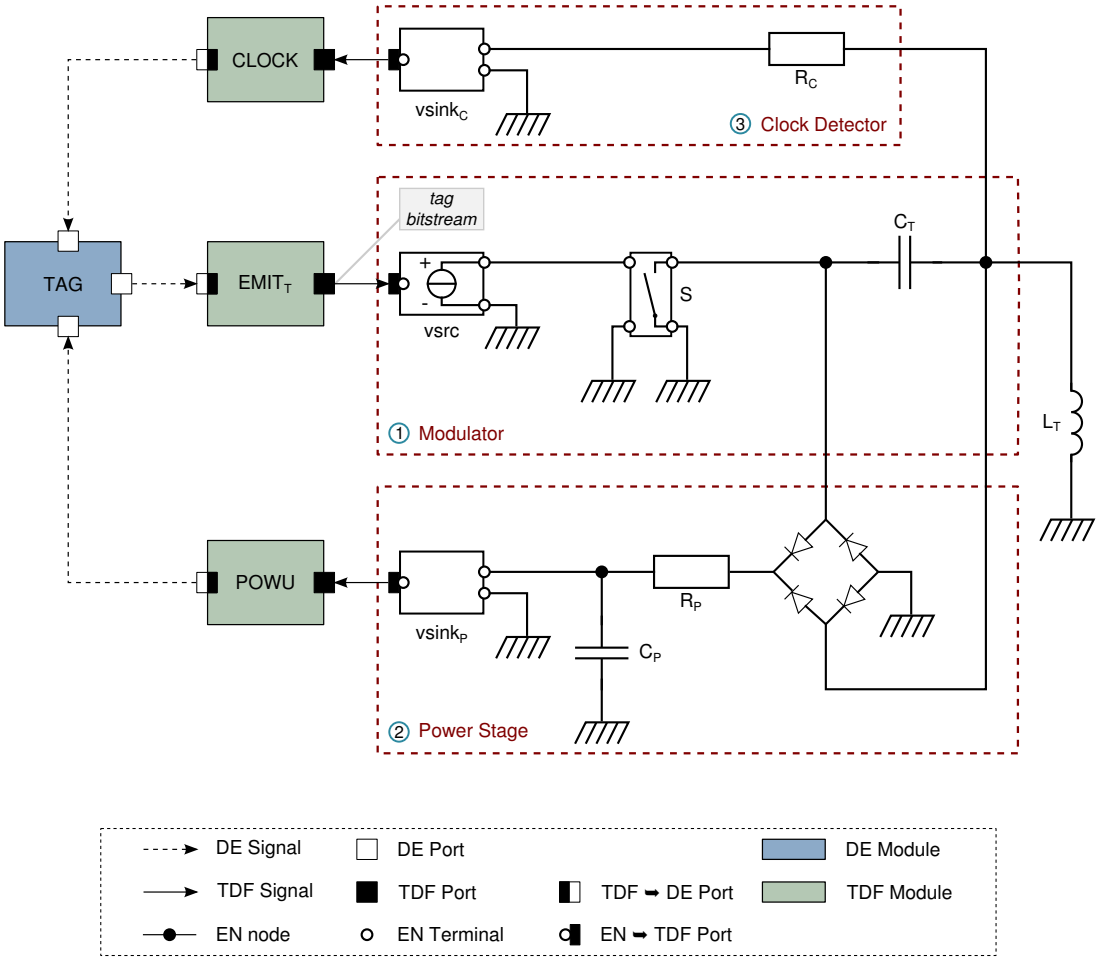

Figure 8.10: Detailed representation of the RFID transponder modeled.

CENT, which consists of a high pass filter. This is achieved in order to improve the signal received. All the components presented in this reception chain are modeled thanks to the EN MoC.

In addition, there is a hysteresis comparator, *HYST*, modeled using the TDF MoC which aims to identify the logical values associated with the signal received. A hysteresis comparator represents a comparator with two thresholds - th_H and th_L. When the input value is higher than the threshold th_H the comparator output is high. When the input value is lower than the threshold th_L (lower than th_H) the comparator output is low. When the input value is between the two thresholds the output retains its current value. This approach means that in order to change the output of the comparator the input value of the comparator must significantly change. Consequently, this approach prevents undesired bit-flips and oscillations in the received bitstream compared to a simple comparator.

Eventually, this circuit is controlled by a digital chain, *CTRL*, modeled by means of DE MoC; it receives the values transmitted by the transponder after being through the reception chain.

In the transponder circuit we can highlight three major mechanisms connected to the secondary coil L_T: an emission chain, a clock detector and a power detector. The emission chain is modeled by means of an analog-to-digital converter, $EMIT_T$, (TDF MoC) linked to an amplitude modulator,

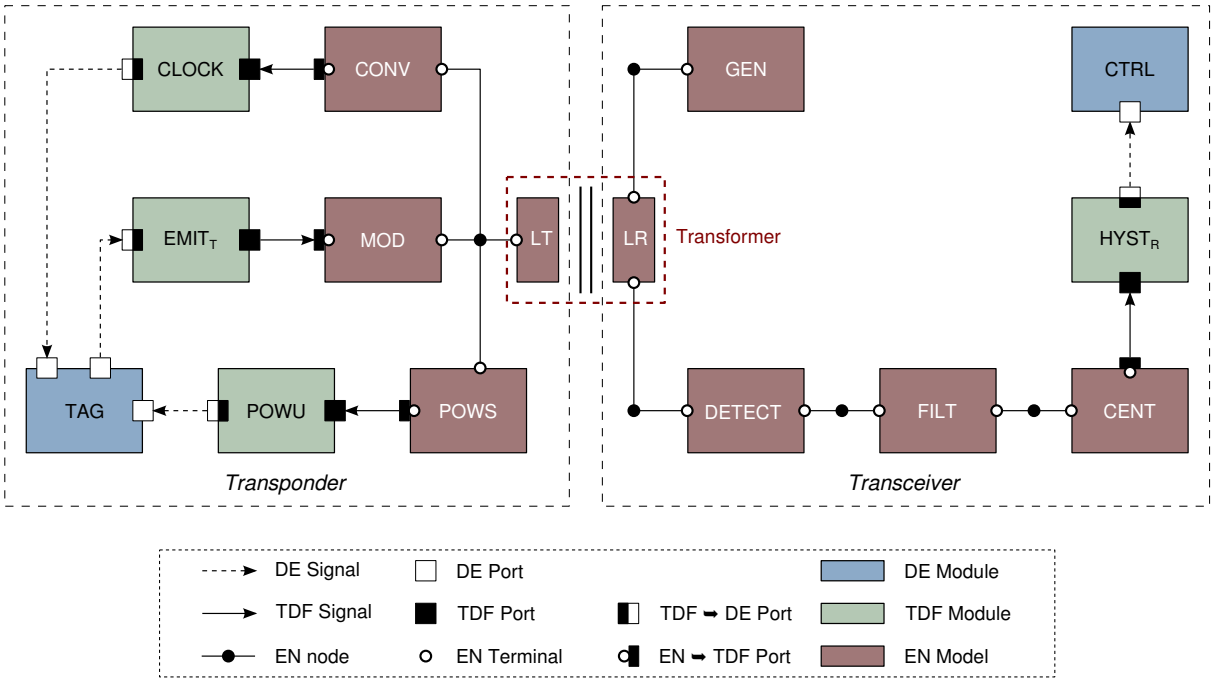

Figure 8.11: Model of the whole passive RFID reading system.

MOD, (Figure 8.10 ①) modeled with the EN MoC.

In order to be supplied by the transceiver the transponder contains a power detector mechanism composed of a power stage, *POWS*, (Figure 8.10 ②), modeled with the EN MoC, and a hysteresis comparator that acts as a power-up, *POWU*, modeled using the TDF MoC.

The clock detector mechanism is composed of a converter, *CONV*, (Figure 8.10 ③) described with the EN MoC and a hysteresis comparator, *CLOCK*, modeled with the TDF MoC.

Finally, there is a digital tag frame generator, *TAG*, modeled using the DE MoC. It uses the clock (provided by the clock detector mechanism) and the energy (provided by the power detector mechanism) in order to generate (through the emission mechanism) the tag identification marker of the transponder.

8.3.2 Results

In order to test and verify our modeling of the passive RFID reading system in the SystemC MDVP environment, we used as a reference a real physical prototype made by Francis Bras from UPMC. This real physical prototype is shown in Figure 8.12. We present a front view of the RFID transceiver system (Figure 8.12, a.) along with a full view of this transceiver (Figure 8.12, b.). We observe the previously described primary coil connected to a few electronic components and test points. The design of this transceiver corresponds to the detailed representation of the RFID transceiver presented in Figure 8.9.

We also present the transponder of the real physical prototype (Figure 8.12, c.). We observe the secondary coil, previously described, connected to electronic components. The aforementioned digital tag frame generator, is designed with an Arduino board which embeds an *ATMEGA2560* processor.

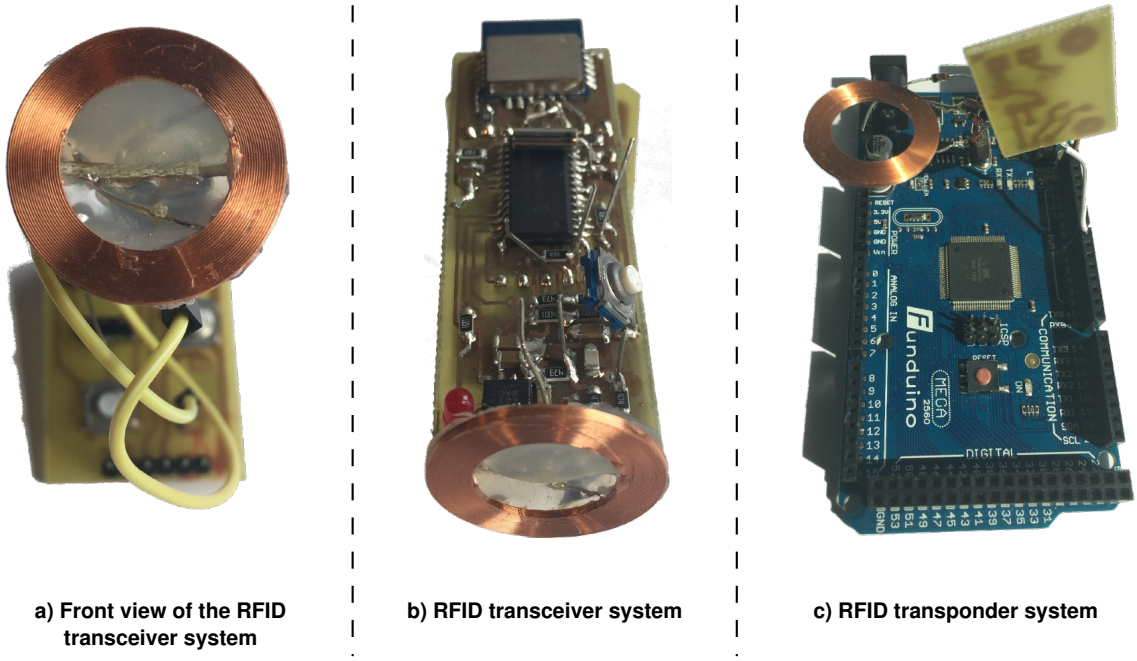

Figure 8.12: Existing physical RFID tag reader.

In order to verify the simulation results obtained using SystemC MDVP we collected information on this physical prototype. We used a numerical oscillator to probe the RFID reading system[1]. We probed six points in the RFID reading system; these six points are highlighted on Figure 8.9 and Figure 8.10.

First, we probed the output of the tag generator to obtain the *bitstream* generated by the transponder. Second, we probed the signal received on the transceiver side at various points in the reception chain. We measured the modulated signal received (*signal_mod*), the envelope detected (*env*), the envelope filtered (*env_filter*), the envelope centered (*env_center*) and eventually the output of the comparator (*comp_out*).

Figure 8.13 presents the measures obtained at the six points of the system; only a section of the tag transmitted is represented in order to keep the waveforms understandable. In the first waveform we see the bitstream generated, followed by the modulated signal. We observe a drop on the modulated signal when a logical '1' value is transmitted. We have in the following waveforms the envelope detected, the envelope filtered and then the envelope centered. The envelope represents the modulated signal where the carrier is removed, the envelope filtered and centered depict a clearer representation of the transmitted information. Although the carrier is still present in the detected envelope (strong oscillation on the high and low levels), we can start to distinguish a pattern which represents the transmitted tag. Eventually, the last waveform

[1]We used a numerical oscillator Lecroix to probe the physical prototype.

represents the output of the comparator, and therefore the information received by the transceiver. We observe that this waveform maps the tag generated and represented in the first waveform. We can conclude that the transceiver accurately received the exact bitstream generated by the transponder.

Figure 8.14 presents a closer look on the measures obtained at the six points of the system. This perspective shows the details of the data observed during a single milliseconds. It allows us to apprehend the nature of the data and better observed the phenomenon involved in a RFID reading system throughout the whole reception chain. We can easily observe the different variations that occur along the different point in the system.

We then described the passive RFID reading system previously detailed within SystemC MDVP. We followed the representation presented in Figure 8.9 and Figure 8.10 which gives us the global representation illustrated in Figure 8.11. We monitored the same points as with the real physical prototype using the monitoring mechanism provided by SystemC MDVP. The monitoring mechanism is currently not supported within the EN MoC; we, therefore, inserted TDF probes at key points in order to have access to the data of interest. TDF probes correspond to converter modules that translate the information from the EN semantics to the TDF semantics.

The simulation results obtained are presented in Figure 8.15. The waveforms are presented in the same order as with the measures on the physical prototype. We show in the first waveform the bitstream generated, followed by the modulated signal. Then, comes the envelope detected, the envelope filtered, and the envelope centered. Ultimately, we present the output of the comparator, i.e. the information received by the transceiver. As with the measures presented above, we only display a section of the tag transmitted.

In terms of behavior the virtual prototype is faithful to the behavior of the physical prototype. We observe the same behavior when a logical '1' is transmitted, that is to say a drop on the modulated signal. The envelope detected is pretty close to the reference measures. However, we note that the signal is *cleaner*. In effect, within the virtual prototype we did not model potential noises, which could occur with analog circuit, making the detection more efficient. We also should mention an offset of 1.5 Volts, approximatively, in the simulation results compared to the reference measures. This can be explained by the small differences in the model compared to the physical system such as the use of a switch in the virtual model while a transistor is used in the physical one, or the presence of an additional power source in the virtual prototype (which may introduce a continuous components in the circuit).

The envelope filtered expresses the same characteristics as the envelope detected. It is faithful to the behavior of the physical prototype, the signal is cleaner, and we also observe the aforementioned 1.5 Volts offset. In turn, the envelope centered is also faithful to the behavior of the physical prototype. The continuous component previously observed is not present anymore and the signal is align on zero as within the physical prototype. Finally, the output of the comparator, depicted in the last waveform, matches the tag generated and presented in the first waveform.

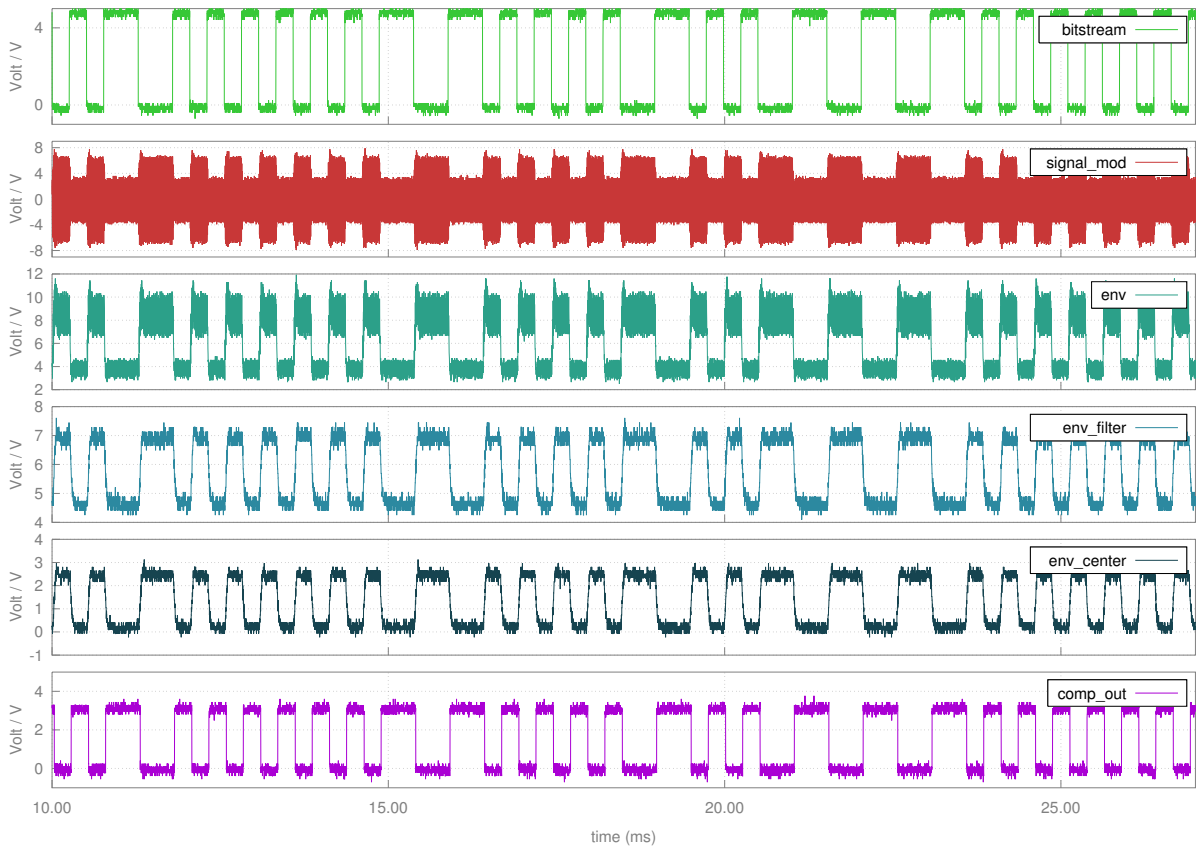

Figure 8.13: Measures on the physical prototype during the transmission of a tag marker.

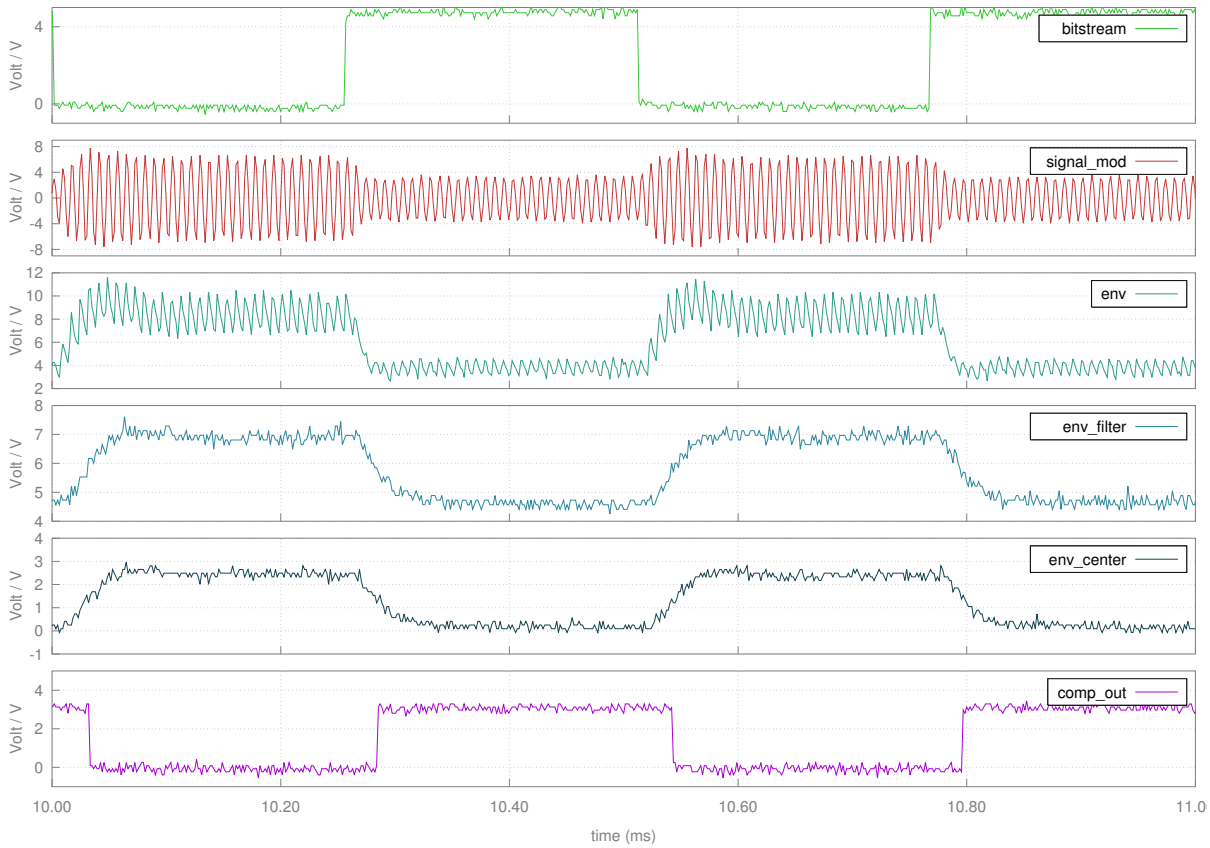

Figure 8.14: Closer look on the measures on the physical prototype during the transmission of a tag marker.

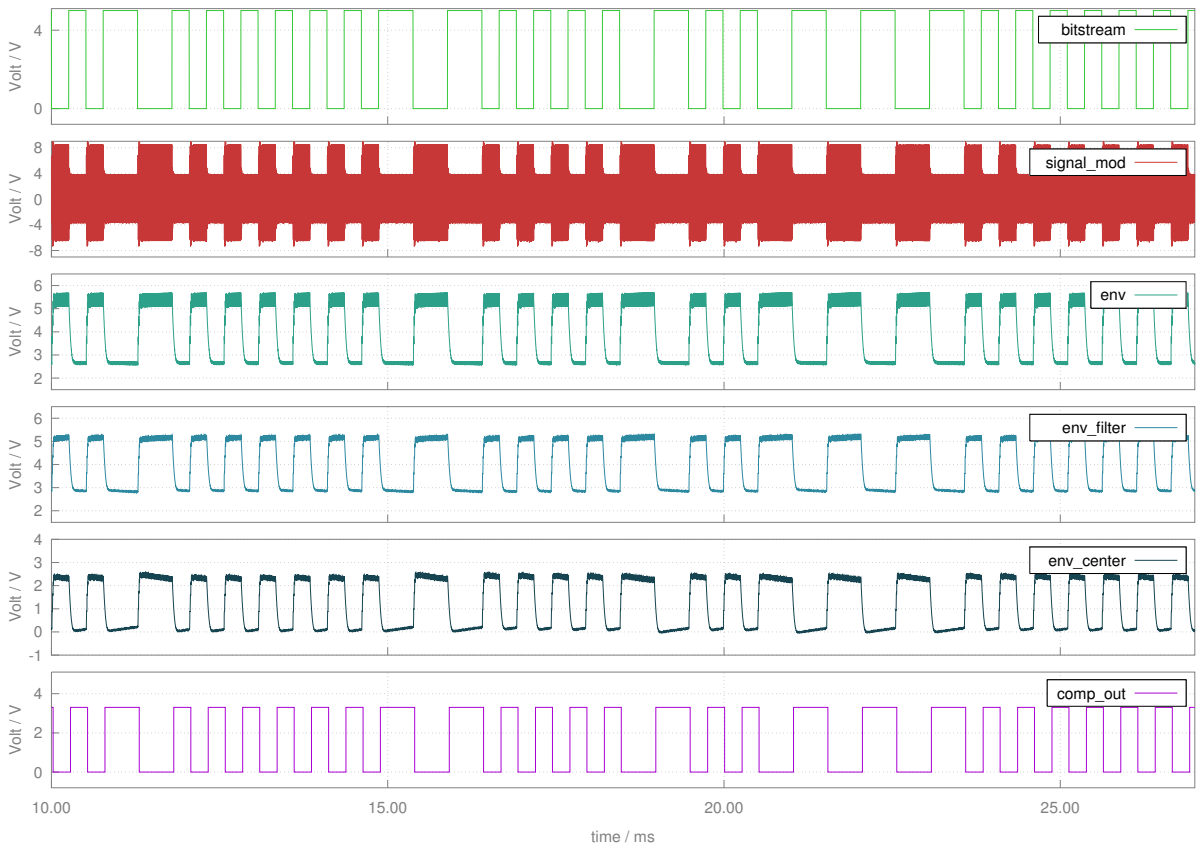

Figure 8.15: Simulation results for the transmission of a tag marker with the virtual RFID prototype.

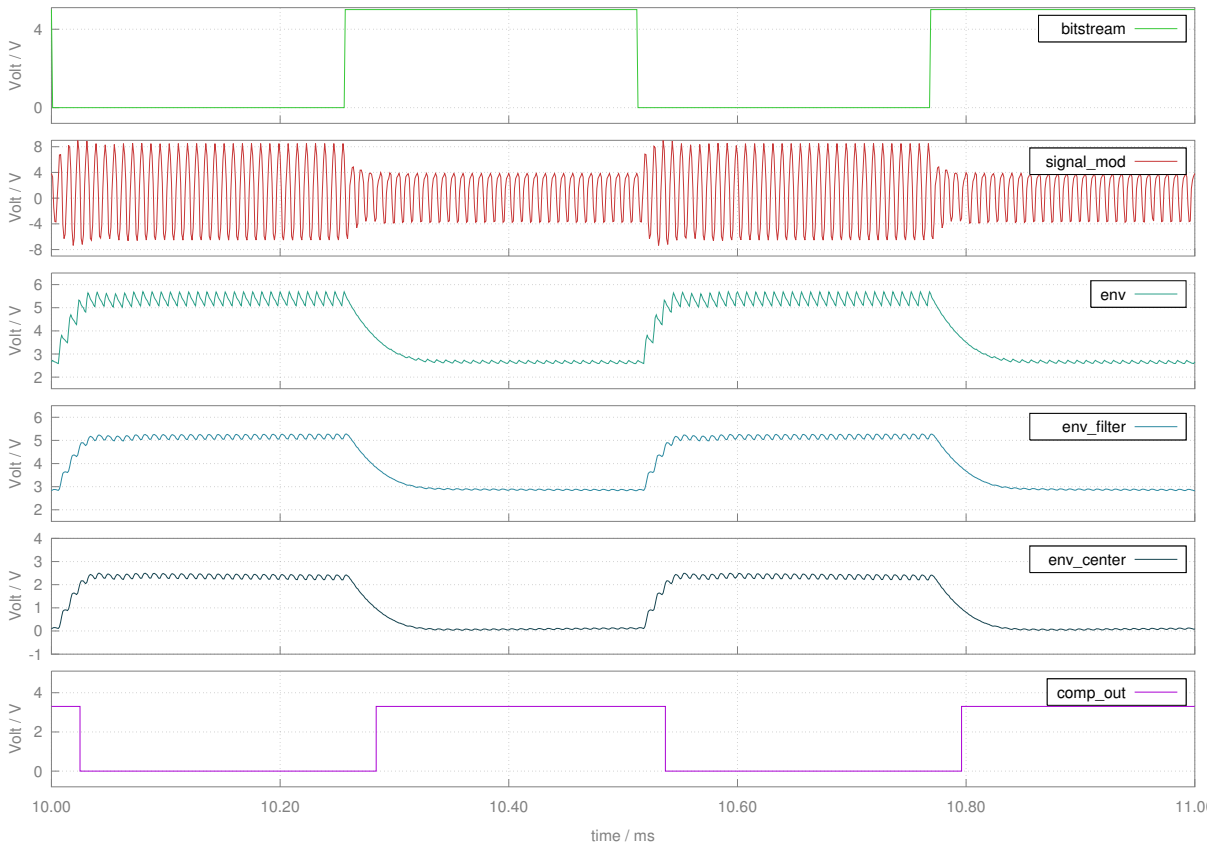

Figure 8.16: Closer look on the simulation results for the transmission of a tag marker with the virtual RFID prototype.

We can conclude that the simulation results match the experimental results measured on the real physical RFID prototype in terms of behavior. In terms of timing analysis, we observe that the time constant on the virtual and physical prototypes are the same. The time required to transmit a tag marker is the same on both versions. As with the real physical prototype, a closer look on the measures obtained during the simulation is provided in Figure 8.16.

The transaction time needed to transmit a tag identifier marker takes approximatively 35 milliseconds. In order to evaluate the performance of our virtual prototyping environment we performed several simulations with different simulation parameters. We varied the timestep of the simulation to observe the impact on the simulation. We chose several timesteps ranging from 1 microseconds to 100 microseconds. The impact of the variation of the timestep on the simulation time is illustrated in Figure 8.17.

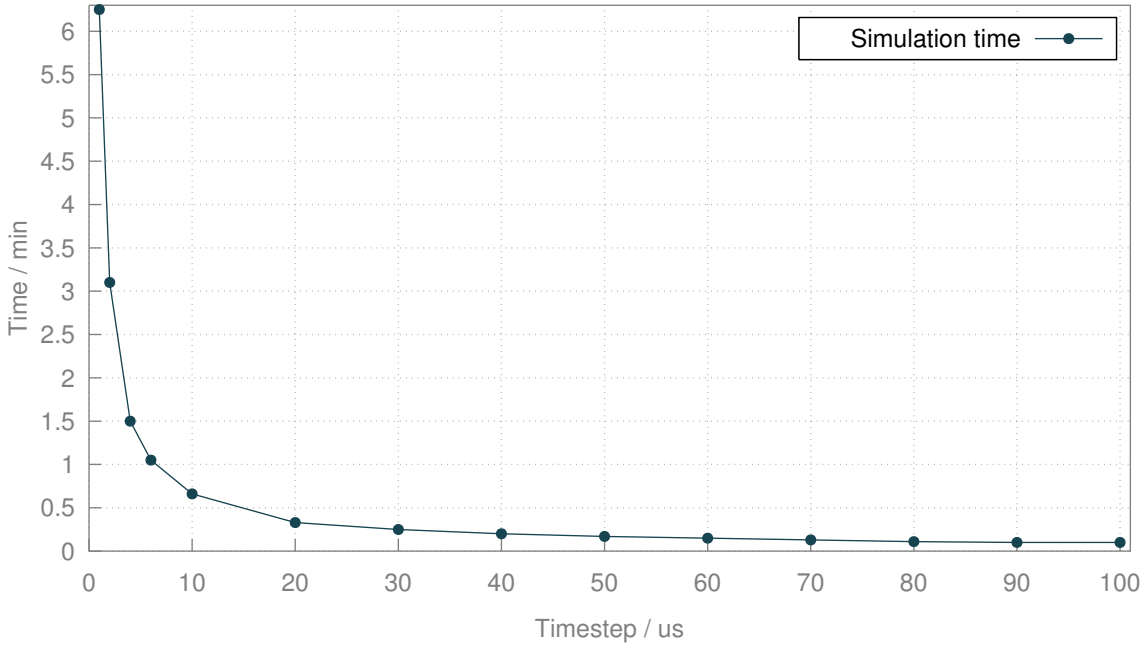

Figure 8.17: Simulation time to emit a RFID tag depending of the simulation timestep.

In such a context, with the smallest timestep, it took us 6 minutes and 16 seconds to perform 35 milliseconds of simulation of the whole RFID system using SystemC MDVP. With such a small timestep, we obtain quite accurate simulation results concerning the analog signals compared with the real physical prototype; however, it heightens the simulation duration. If we choose a bigger timestep, such as 100 microseconds, it took us only 6 seconds to perform 35 miliseconds of simulation of the whole RFID system. With such a big timestep, the simulation results obtained for the analog signals are not relevant while the digital values (generated tag and received tag) remain correct.

We compared the analog data measured at different timesteps with the values obtained at the smallest timestep. This analysis highlights that each sample, no matter the timestep, is close to the corespondent sample obtained at 1 microseconds, i.e. the sample with the same timestamps, with a similarity approaching 99%. Two conclusions can be drawn. Firstly, increasing the timestep

only reduces the number of sample while keeping qualitative results. Secondly, the representation of the analog signals becomes irrelevant since the waveforms do not express the analog behavior anymore due to less sample.

We can conclude that these variations of timestep bring out the fact that SystemC MDVP can fulfil the expectation of designers interested in the analog details as well as designers more interested in the digital part.

If we take a look at the overhead induced by the monitoring, we observe that it is quite small. Indeed, the monitoring increases the simulation by only 1.5%, as illustrated in Table 8.4. To evaluate our monitoring mechanism we compared the overhead induced by the tracing mechanism implemented within SystemC with our approach.

Table 8.4: Simulation time of the passive RFID reading system.

Simulation without monitoring	6m10.369s
Simulation with SystemC MDVP monitoring	6m16.569s
Simulation with SystemC monitoring	6m16.861s

We introduced DE probes at key points in order to have access to the relevant data. DE probes represent converter modules that translate the information from the analog part to the digital part. We observe no significant difference in the simulation time between both approaches. While our monitoring mechanism impacts the simulation the same way the monitoring of SystemC does, it has more to offer. With the same impact, we offer a single interface to perform the monitoring of both analog and digital part, and it does not require the insertion of converter modules to transform every signal in a DE one.

8.4 Conclusion

The purpose of this chapter was to illustrate the possibilities provided with our virtual prototyping environment. First, we presented a case study describing a fluidic network. The conception of the fluidic network was carried out by several approaches in a bid to stay as fair as possible. One approach was based on SystemC AMS and a MoC called PFN that allows the description of a fluidic network using an electrical equivalent circuit. The second approach was carried out on SystemC MDVP and the SPH MoC. Both approaches were compared with experimental results measured on a real prototype of the fluidic network.

The PFN modeling mechanism allows a simple description of the fluidic system. However, the timing results obtained were very precise over several types of experiments. Moreover, simulation approach demonstrated to be fast in terms of modeling and run time.

When it comes to the holistic modeling of a micro-fluidic system that takes into account the exact geometry of the fluidic network, it appears that the SPH approach is fast enough to offer

the end-user quite approximate results in a reasonable time, compatible with the key idea that SystemC MDVP is mainly dedicated to the building of first order executable specifications that in turn aim to develop the embedded software as soon as possible with a quite faithful representation of all the hardware parts. As such, it is a good trade-off between the simpler (but quite accurate) PFN scheme and FEM.

Furthermore, considering fluids as a set of particles, the SPH approach has a lot of potential capabilities for the next steps in the emulation of a complex fluidic system. Processes like magnetic trapping of polarized cells, fluidic mixing or counting specific particles could be managed quite naturally.

Second, we introduced a complete case study to validate the modeling and simulation principles carried out by our virtual prototyping environment SystemC MDVP. This case study consisted in the modeling of a passive RFID reading system. A RFID system is composed of two subsystems - a transponder (a tag) and a transceiver (a reader). A tag carries a unique identification marker and, in a passive system, does not have a local power source. Therefore, the transceiver provides energy to the tag and, then, serially read the bitstream transmitted by the tag.

We showed that our approach, which relies on a block components association, allows for an easy and straightforward conception of a model. The instantiation of the different components is quite easy and does not require to the SoC architect to deal with simulator's parameter.

With this case study, we used three different Models of Computation: Electrical Network, Timed Data Flow and Discrete Event. This model strongly exploits the interaction mechanism that we defined within SystemC MDVP and demonstrates its efficiency.

The simulation results obtained during the simulation of the RFID prototype are really satisfying. Indeed, these results remarkably match the real values obtained using the physical prototype of this RFID system. The simulation is also fast while remaining accurate and under the control of the designer. It is worth noting that the simulation can go significantly faster at a price of losing the representation of the analog signals.

This case study demonstrates the efficiency of our solution SystemC MDVP, we integrate several MoCs in order to model an heterogeneous platform. Further work will be done in order to complete the transceiver of this passive RFID system. Eventually, the digital controller on the transceiver side will be modeled by a DE micro-controller.

Chapter

Conclusion

Contents

9.1 Conclusion 138
9.2 Perspective 140

9.1 Conclusion

Current and future microelectronics systems are increasingly and intrinsically complex, they integrate more and more interaction with their surrounding environment. Hence, they become multi-disciplinary microelectronics-assisted systems. In this thesis, we explored the requirements to perform the simulation of such heterogeneous systems as a whole within a unified virtual prototyping environment. We presented our solution SystemC MDVP which defines a new simulator prototype on top of SystemC, allowing the simulation of heterogeneous systems within a single environment in a monolithic way. The definition of SystemC MDVP was inspired by others existing simulators such as Ptolemy II [20] and SystemC AMS [50].

During this thesis we identified and addressed these requirements as presented below.

Smooth Management of Heterogeneity

When dealing with heterogeneous systems the first thing that comes to mind is to define what is intended by heterogeneity and how to smoothly manage it. Our approach relies on the interaction between Models of Computation, these MoCs defining a heterogeneous entity within our framework. In Chapter 4 we defined our vision of a MoC and we proposed an abstraction of the notion of MoC. This abstraction expresses all the requirements that we believe a MoC should meet. In every MoC we should retrieve several notions that characterize the MoC:

- ***Time Representation***: the abstraction of time used by the MoC (continuous time (EN), discrete time (DE), sampled time (TDF), etc...).
- ***Primitive Behavior***: the basic blocks which describe an elementary behavior, or the way to associate a behavior with a basic block when allowed to do so.
- ***Channel Representation***: the communication mechanism used to exchange data between basic blocks.
- ***Composition***: the way to compose a bigger model through ports and sub-model instantiation.
- ***Solving Algorithm***: the algorithm used to resolve the model (solver, scheduler, etc...).
- ***Interaction***: the way to communicate with another Model of Computation.

This abstraction allows for a clear definition of the entity involved in the model from the simulator point-of-view and also from the end-user point-of-view. Indeed, this clear interface is used by the simulator kernel to handle the different MoCs, as shown in Chapter 5. On the other side, the explicit representation of primitive behavior, channel and port allows the end-user to easily design a system as illustrated along the Chapter 4.

Sound Management of Interacting Entities

We identify that one of the key challenges in the simulation of heterogeneous system lies in the interaction between the different entities in the model. Our solution consists in representing these interactions by means of master-slave semantics, as depicted in Chapter 4. This approach allows for a seamless interaction between different MoCs subject to the respect of the semantics rules. The main rules states that a master MoC does not need to be aware of the existence of potential slave MoCs; hence, the slave MoC has to comply with its master interface. This means that all the mechanisms to interact with a master MoC are only managed by the slave MoC. Consequently, each MoC should express through a MoC interface all the requirements it expects in order for other MoCs to setup a master-slave relationship with it. The main constraint imposed to a MoC is that it cannot be simultaneously the slave of several distinct master MoCs.

Our solution allows for a sound management of interacting entities within our virtual prototyping environment. We expressed a hierarchical organization of MoCs and hence, a clear interaction pattern is defined. This approach allows us to free the end-user from the responsibility of defining himself the interaction rules that would have prevailed in the model he is designing. Indeed, our framework automatically handles the interaction between the MoCs, as presented in Chapter 5.

Flexible Virtual Prototyping Environment

It seems natural that the simulation of heterogeneous systems implies some flexibility in the simulator due to the very nature of the entities modeled. Indeed, heterogeneous entities that may take part in the conception of a multi-physical system are numerous, ranging from biological to optical through mechanical, etc. Our solution provides this flexibility since it is conceived following a completely generic approach. Chapter 5 illustrated the implementation details, i.e. the generic algorithms that guarantee the flexibility of SystemC MDVP. Defining a clear definition and an abstraction for the MoCs allow us to implement the SystemC MDVP simulator in a completely MoC-independent way. This approach ensures the flexibility of the framework since it does not rely on any MoC-definition. This provides the opportunity to enrich the set of entities within the framework without requiring any modification in the simulator.

We explained in Chapter 7 the methodology to follow in order to define and integrate new Models of Computation within SystemC MDVP. Since the definition of MoCs relies on the abstraction defined in the SystemC MDVP kernel, the integration within the framework is quite straightforward and easy. The methodology is supported by the integration of a new MoC called Smoothed Particle Hydrodynamics that aims to model fluidic elements.

Multi disciplinary Monitoring

The monitoring of heterogeneous systems represents an important feature and also is a real challenge. The main challenge lies in the fact that such mechanism should adapt to and fit the different digital/physical parts involved in the design. In our solution, this translates in the challenge to unify within a single entity the monitoring of SystemC parts (digital) with SystemC MDVP parts (physical). We presented in Chapter 6 the principles and implementation details that allow SystemC MDVP to perform multi-disciplinary monitoring while remaining flexible and generic.

Our monitoring mechanism relies on generic algorithms while remaining able to gather relevant and various data from different MoCs. To achieve this purpose, we developed a solution to address the monitoring of SystemC components inspired by the tracing mechanism of SystemC itself. This was a difficult task since we chose to stay compliant with the standard SystemC, therefore we were not allowed to alter the SystemC simulation kernel. On the other side, we developed a solution for the monitoring of SystemC MDVP components inspired by the communication mechanism implemented in QT - the signal/slot pattern.

We successfully provide a single interface in order to monitor digital and physical parts within SystemC MDVP. Thanks to this interface, monitoring set up from the end-user point-of-view is straightforward. The generic approach that we manage to keep in the definition of the monitoring mechanism ensures the flexibility of SystemC MDVP.

The solution to the simulation of heterogeneous systems that we proposed tackles all the challenges identified in the beginning of this document. It relies on a clear definition of the entities involved in the simulation, both from the simulator and the end-user point-of-views. We defined a strong and efficient interaction mechanism, allowing a high level of flexibility. Our framework expresses the capacity to be enhanced with new entities. Eventually, SystemC MDVP supports an efficient monitoring mechanism that fits the specificities of multi-disciplinary systems. We illustrated the efficiency of our framework through validation case studies in Chapter 8. We modeled a passive RFID reading system using three different MoCs and compared the simulated results with the physical ones. We demonstrated the simplicity of use of our solution, and the satisfying performance exhibited by our framework. We met the expectation in terms of speed of simulation while remaining quite accurate.

9.2 Perspective

The principles and mechanisms presented herein represent the basic of the requirement for truly achieve simulation of heterogeneous systems. Areas for improvement are worth exploring.

Smoothed Particle Hydrodynamics (SPH) Model of Computation (MoC)

The SPH MoC presented in this thesis represents the first version of a MoC with a great potential. There is room for improvement with this MoC. First the inner mechanisms of the MoC are still very sensitive to parameters and a more generic approach should be investigated in order to express all the dependencies between the different parameters. This would lead to ease the tuning of the MoC when modifying the nature of the fluid simulated. Second, the detection and handling of collisions is not a trivial problem and the answer implemented within the MoC could benefits from the use of a third parties library in charge of 3D rigid-body interactions for example. Eventually, we can improve the model in order to take into account other forces that may be applied to the fluid (such as magnetic forces for example).

Integration of new Models of Computation

In order to validate and support the principles and mechanisms presented in this thesis, new Models of Computation are required. These new MoCs should express different time domains and represent different physical domains to demonstrate the efficiency and genericity of SystemC MDVP. Currently, an ODE MoC is under integration within SystemC MDVP and other MoCs are considered, such as a Quantum MoC.

Improvement within SystemC MDVP

The development of a unified design environment for virtual prototyping of heterogeneous systems requires a mechanism to express functional properties that must be fulfilled in order to validate the system designed. In the context of heterogeneous systems, these properties may refer to a subset of the system involving only one discipline or to a subset of the system involving several disciplines up to the entire system. Thus, it is required to provide a mechanism that permits the expression of properties that cross the disciplines. Within this thesis, we believe that we laid the foundation that can lead to the multi-physical functional verification of heterogeneous systems through simulation. Indeed, the hierarchical representation of the system and its abstraction through the notion of cluster defines a well-suited structure for the partitioning of properties that cross the disciplines. Moreover, this functional verification mechanism can also benefit highly from the monitoring mechanism developed within SystemC MDVP; we already provide in our framework a way to probe the system in order to gather relevant information. This information may, eventually, be used as part of an assertion-based approach that will cover the models described with SystemC MDVP and also SystemC.

Another area of improvement concerns the performance and especially the parallelization of the execution. Indeed, we believe that the hierarchical representation that we defined within SystemC MDVP can allow for a multi-threading simulation of the models designed. Moreover, the

elaboration and simulation mechanisms presented herein are by essence, thanks to distributed and in cascade algorithms, well-suited to support parallelism. Therefore, we believe that we designed a path to a parallel multi-physical simulation environment.

SystemC MDVP is currently already used in the framework of another thesis [12] with an application in the automotive area. Our framework begins to be a support for internship in our laboratory. I am ambitious that SystemC MDVP could become a key actor in the conception flow of future heterogeneous systems.

Publications

- Victor Fernandez, Elier Wilpert, Herique Isidoro, Cedric Ben Aoun, and Francois Pecheux. SystemC-MDVP modelling of pressure driven microfluidic systems. In *Embedded Computing (MECO), 2014 3rd Mediterranean Conference on*, pages 10–13, June 2014

- Torsten Maehne, Zhi Wang, Benoit Vernay, Liliana Andrade, Cédric Ben Aoun, Jean-Paul Chaput, Marie-Minerve Louërat, François Pêcheux, Arnaud Krust, Gerold Schropfer, et al. Uvm-systemc-ams based framework for the correct by construction design of mems in their real heterogeneous application context. In *Electronics, Circuits and Systems (ICECS), 2014 21st IEEE International Conference on*, pages 862–865. IEEE, 2014

- Víctor Fernández, Andrés Mena, Cédric Ben Aoun, François Pêcheux, and Luis J Fernández. Virtual prototyping of pressure driven microfluidic systems with SystemC-AMS extensions. *Microprocessors and Microsystems*, 39(8):854–865, 2015

- Liliana Andrade, Torsten Maehne, Alain Vachoux, Cédric Ben Aoun, François Pêcheux, and Marie-Minerve Louërat. Pre-simulation formal analysis of synchronization issues between discrete event and timed data flow models of computation. In *Proceedings of the Design Automation & Test in Europe (DATE) Conference 2015*, 2015

- Liliana Andrade, Cédric Ben Aoun, Benoit Vernay, Torsten Maehne, François Pêcheux, and Marie-Minerve Louërat. Understanding the heterogeneous hardware: Do not forget the interconnection! In *2nd Workshop on Design Automation for Understanding Hardware Designs (DHUDe) at DATE*, 2015

- Cédric Ben Aoun, Liliana Andrade, Torsten Maehne, François Pêcheux, Marie-Minerve Louërat, and Alain Vachoux. Generic eda-standard based elaboration scheme for the efficient monolithic simulation of heterogeneous systems. In *Work-in-progress session at 52th Design Automation Conference (DAC) 2015*, 2015

- Cédric Ben Aoun, Liliana Andrade, Torsten Maehne, François Pêcheux, Marie-Minerve Louërat, and Alain Vachoux. Pre-Simulation Elaboration of Heterogeneous Systems: The SystemC Multi-Disciplinary Virtual Prototyping Approach. In *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XV), 2015 International Conference on*, July 2015

Bibliography

[1] Business Wire. Apple watch sport ihs teardown (graphic: Business wire). http://www.businesswire.com/news/home/20150430006412/en/CORRECTING-REPLACING-Apple-Watch-Lowest-Ratio-Hardware. 2

[2] idownloadblog. Apple watch motherboard. http://www.idownloadblog.com/2015/05/07/apple-watch-s1-samsung/. 3

[3] TechInsights. Apple watch packaging. http://www.techinsights.com/about-techinsights/overview/blog/apple-watch-teardown/. 3

[4] ABIresearch. Apple watch packaging. https://www.abiresearch.com/press/apple-watch-insides-pcb-details-revealed-for-the-f/. 3

[5] Ragunathan (Raj) Rajkumar, Insup Lee, Lui Sha, and John Stankovic. Cyber-physical systems: The next computing revolution. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 731–736, New York, NY, USA, 2010. ACM. 3

[6] INFSO D.4 Networked Enterprise & RFID INFSO G.2 Micro & Nanosystems in Co-operation with the RFID Working Group of the EPoSS. Internet of things in 2020: A roadmap for the future. *European Commission: Information Society and Media*, September 2008. 3

[7] Gartner Inc. Gartner says 6.4 billion connected "things" will be in use in 2016, up 30 percent from 2015. http://www.gartner.com/newsroom/id/3165317. 3

[8] G Gary Wang. Definition and review of virtual prototyping. *Journal of Computing and Information Science in engineering*, 2(3):232–236, 2002. 4

[9] James C. Schaaf, Jr. and Faye Lynn Thompson. System concept development with virtual prototyping. In *Proceedings of the 29th Conference on Winter Simulation*, WSC '97, pages 941–947, Washington, DC, USA, 1997. IEEE Computer Society. 4

[10] H-INCEPTION Consortium. Homepage of the CATRENE (CA701) Heterogeneous Inception (H-INCEPTION) project. https://www-soc.lip6.fr/trac/hinception. 5

[11] Liliana Lilibeth Andrade Porras. *Principes et réalisation d'une interface de synchronisation interopérable entre modèles de calcul SystemC AMS pour le prototypage virtuel optimisé de*

systèmes multi-disciplines. PhD thesis, Université Pierre et Marie Curie (UPMC)), January 2016. 11, 21

[12] Vanessa Tran. *Prototypage Virtuel Holistique Multi-Discipline d'un alterno demarreur et de son système de contrôle pour automobile hybrid*. PhD thesis, Université Pierre et Marie Curie (UPMC)), In progress. 11, 142

[13] FMI development group. Functional Mock-up Interface (FMI). https://www.fmi-standard.org/. 15

[14] M Arnold, T Blochwitz, C Clauß, T Neidhold, T Schierz, and S Wolf. Fmi-for-cosimulation. In *1st Conference on Multiphysics Simulation, Bonn*, 2010. 15

[15] Olaf Enge-Rosenblatt, Christoph Clauß, André Schneider, and Peter Schneider. Functional digital mock-up and the functional mock-up interface-two complementary approaches for a comprehensive investigation of heterogeneous systems. In *Proceedings of the 8th International Modelica Conference; March 20th-22nd; Technical University; Dresden; Germany*, number 063, pages 748–755. Linköping University Electronic Press, 2011. 15

[16] T. Blochwitz et al. The functional mockup interface for tool independent exchange of simulation models. In *Proceedings of the 8th International Modelica Conference*, 2011. 15

[17] Jens Bastian, Christop Clauß, Susann Wolf, and Peter Schneider. Master for co-simulation using FMI. In *Proceedings of the 8th International Modelica Conference; March 20th-22nd; Technical University; Dresden; Germany*, number 63, pages 115–120. Linköping University Electronic Press, 2011. 15

[18] MODELISAR. https://itea3.org/project/modelisar.html. 15

[19] Joseph T Buck, Soonhoi Ha, Edward A Lee, and David G Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. 1994. 16

[20] J. Eker et al. Taming Heterogeneity - The Ptolemy Approach. *Proceedings of the IEEE*, 91(1):127–144, January 2003. 16, 34, 138

[21] Christopher Brooks, Edward A Lee, Xiaojun Liu, Stephen Neuendorffer, Yang Zhao, Haiyang Zheng, Shuvra S Bhattacharyya, Elaine Cheong, II Davis, Mudit Goel, et al. Heterogeneous concurrent modeling and design in java (volume 1: Introduction to ptolemy ii). Technical Report UCB/EECS-2008-28, EECS Department, University of California, Berkeley, California, USA, 2008. 16

[22] Edward A. Lee. Heterogeneous Actor Modeling. In *Proceedings of the 9th ACM International Conference on Embedded Software*, EMSOFT '11, pages 3–12, New York, NY, USA, 2011. ACM. 16

[23] Christopher Brooks, Edward A Lee, Xieojun Liu, Stephen Neuendorffer, Yang Zhao, Haiyang Zheng, Shuvra S Bhattacharyya, Elaine Cheong, II Davis, Mudit Goel, et al. Heterogeneous concurrent modeling and design in java (volume 2: Ptolemy ii software architecture). Technical

Report UCB/EECS-2008-37, EECS Department, University of California, Berkeley, California, USA, 2008. 16

[24] Christopher Brooks, Edward A Lee, Xiaojun Liu, Stephen Neuendorffer, Yang Zhao, and Haiyang Zheng. Heterogeneous concurrent modeling and design in java (volume 3: Ptolemy ii domains). Technical Report UCB/EECS-2008-37, EECS Department, University of California, Berkeley, California, USA, 2008. 16

[25] Cécile Hardebolle and Frédéric Boulanger. ModHel'X: A component-oriented approach to multi-formalism modeling. In *Models in Software Engineering*, pages 247–258. Springer, 2008. 17

[26] Peter Fritzson. *Principles of object-oriented modeling and simulation with Modelica 2.1*. John Wiley & Sons, 2010. 17

[27] Peter Fritzson and Vadim Engelson. Modelica - a unified object-oriented language for system modeling and simulation. In Eric Jul, editor, *ECOOP'98 - Object-Oriented Programming: 12th European Conference Brussels, Belgium, July 20-24, 1998 Proceedings*, pages 67–90, Berlin, Heidelberg, July 1998. Springer Berlin Heidelberg. 17

[28] AB Dynasim and Sweden Lund. *Dymola Users' Manual*, 2006. 17

[29] Edward A. Lee et al. *System design, modeling, and simulation: using Ptolemy II*. Ptolemy. org Berkeley, 2014. 18

[30] Luca P Carloni, Roberto Passerone, Alessandro Pinto, Alberto L Sangiovanni-Vincentelli, et al. Languages and tools for hybrid systems design. *Foundations and Trends® in Electronic Design Automation*, 1(1–2):1–193, 2006. 18

[31] MATLAB. The MathWorks. *Inc., Natick, Massachusetts, United States*. 18

[32] Simulink. The MathWorks. *Inc., Natick, Massachusetts, United States*. 18

[33] David C Black, Jack Donovan, Bill Bunton, and Anna Keist. *SystemC: From the ground up*, volume 71. Springer Science & Business Media, 2009. 18, 20

[34] Karsten Einwich, Peter Schwarz, Christoph Grimm, and Christian Meise. SystemC-AMS: Rationales, State of the Art, and Examples. In Wolfgang Müller, Wolfgang Rosenstiel, and Jürgen Ruf, editors, *SystemC: methodologies and applications*, chapter 10, pages 273–297. Springer Science & Business Media, 2003. 18

[35] Christoph Grimm. Modeling and Refinement of Mixed-Signal Systems with SystemC. In Wolfgang Müller, Wolfgang Rosenstiel, and Jürgen Ruf, editors, *SystemC: methodologies and applications*, chapter 11, pages 299–323. Springer Science & Business Media, 2003. 18

[36] Felice Balarin, Yosinori Watanabe, Harry Hsieh, Luciano Lavagno, Claudio Passerone, and Alberto Sangiovanni-Vincentelli. Metropolis: An Integrated Electronic System Design Environment. *IEEE Computer*, 36:45–52, 2003. 19

[37] Felice Balarin, Harry Hsieh, Luciano Lavagno, Claudio Passerone, Alessandro Pinto, Alberto Sangiovanni-Vincentelli, Yosinori Watanabe, and Guang Yang. Metropolis: A design environment for heterogeneous systems. *Multiprocessor Systems-on-Chips*, 2004. 19

[38] Felice Balarin, Luciano Lavagno, Claudio Passerone, Alberto Sangiovanni-Vincentelli, Marco Sgroi, and Yosinori Watanabe. Modeling and designing heterogeneous systems. In *Concurrency and Hardware Design*, pages 228–273. Springer, 2002. 19

[39] IEEE Computer Society. *1666-2011 IEEE Standard for SystemC Language Reference Manual*. IEEE, January 2012. 20, 85

[40] IEEE Computer Society. Introduction to TLM-2.0. In *1666-2011 IEEE Standard for SystemC Language Reference Manual*, pages 415–425. IEEE, New York, USA, 2012. 20

[41] Hessa Al-Junaid and Tom Kazmierski. An Analogue and Mixed-Signal Extension to SystemC. *The Institution of Electrical Engineers Proceedings Circuits Devices Systems*, 152(6):1–10, 2005. 22

[42] Chenxu Zhao and Tom J Kazmierski. An extension to systemc-a to support mixed-technology systems with distributed components. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*, pages 1–6. IEEE, 2011. 22

[43] Hiren Patel and Sandeep Kumar Shukla. *SystemC Kernel Extension for Heterogeneous System Modeling: a framework for Multi-MoC modeling & simulation*. Kluwer Academic Publishers, 2004. 23

[44] Hiren D Patel and Sandeep K Shukla. Towards a heterogeneous simulation kernel for system-level models: a systemc kernel for synchronous data flow models. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(8):1261–1271, 2005. 23

[45] F. Herrera and E. Villar. A framework for embedded system specification under different models of computation in SystemC. In *Proceedings of the 43rd (ACM/IEEE) Design Automation Conference (DAC)-2006*, pages 911–914. ACM/IEEE, 2006. 23

[46] F. Herrera and E. Villar. Hetsc user manual. *Universidad de Cantabria. Santander*, 2008. 23

[47] Karsten Einwich, Peter Schwarz, Christoph Grimm, and Klaus Waldschmidt. Mixed-Signal Extensions for SystemC. In Eugenio Villar and Jean Mermet, editors, *System Specification & Design Languages – Best of FDL'02*, pages 19–28. Springer US, 2003. 23

[48] Alain Vachoux, Christoph Grimm, and Karsten Einwich. Towards analog and mixed-signal soc design with systemc-ams. In *Electronic Design, Test and Applications, Proceedings. DELTA 2004. Second IEEE International Workshop on*, pages 97–102. IEEE, 2004. 23, 24

[49] Christoph Grimm, Martin Barnasconi, Alain Vachoux, and Karsten Einwich. An introduction to modeling embedded analog/mixed-signal systems using systemc ams extensions. In *DAC2008 International Conference*, 2008. 23

[50] Accellera Systems Initiative, SystemC AMS Working Group. *Standard SystemC AMS extensions 2.0 Language Reference Manual*. Accellera Systems Initiative (ASI), March 2013. 23, 25, 138

[51] Karsten Einwich and Peter Schwarz. Systemc-ams steps towards an implementation. In *FDL*, volume 3, pages 1636–987, 2003. 23

[52] Martin Barnasconi, Christoph Grimm, Markus Damm, Karsten Einwich, Marie-Minerve Louërat, Torsten Maehne, François Pecheux, and Alain Vachoux. *SystemC AMS extensions User's Guide*. Open SystemC Initiative (OSCI). XI, 24

[53] Martin Barnasconi, Karsten Einwich, Christoph Grimm, Torsten Maehne, and Alain Vachoux. Advancing the SystemC analog/mixed-signal (AMS) extensions. Technical report, OSCI. 24

[54] Fraunhofer IIS/EAS. SystemC-AMS proof-of-concept (PoC) implementation. `http://systemc-ams.eas.iis.fraunhofer.de/`. 25

[55] Karsten Einwich. Application of SystemC/SystemC-AMS for the Specification of Complex Wired Telecomunication Systems. In *Forum on specification and Design Languages (FDL)*, pages 27–30, Lausanne, Switzerland, 2005. 25

[56] Yao Li, Zhi Wang, Marie-Minerve Louërat, François Pêcheux, Ramy Iskander, Philippe Cuenot, Martin Barnasconi, Thilo Vörtler, and Karsten Einwich. Virtual prototyping, verification and validation framework for automotive using systemc, systemc-ams and systemc-uvm. In *Embedded Real Time Software and Systems (ERTS2)*, pages 1–10, 2014. 25

[57] Coseda Technologies GmbH. Coside. `http://www.coseda-tech.com/coside-overview`. 25

[58] Thomas Uhle and Karsten Einwich. A SystemC AMS extension for the simulation of non-linear circuits. In *Proceedings of the 23 IEEE International SoC Conference (SOCC) 2010*, pages 193–198. IEEE, 2010. 25

[59] Torsten Maehne and Alain Vachoux. Bond graph support in SystemC AMS. In *Proceedings of the 10th International Conference on Bond Graph Modeling and Simulation (ICBGM) 2012*, pages 159–166. SCS, 2012. 25

[60] Torsten Maehne and Alain Vachoux. Supporting dimensional analysis in SystemC-AMS. In *Proceedings of the 2009 IEEE International Behavioral Modeling and Simulation (BMAS) Workshop*, pages 108–113. IEEE, 2009. 25, 59

[61] Matthias Christian Schabel and Steven Watanabe. *Boost.Units 1.1.0*. 25, 59

[62] Tim Rentsch. Object oriented programming. *ACM Sigplan Notices*, 17(9):51–57, 1982. 26

[63] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP'97 — Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer Berlin Heidelberg, 1997. 26

[64] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *European conference on object-oriented programming*, pages 220–242. Springer, 1997. 26

[65] Gregor Kiczales and Erik Hilsdale. Aspect-oriented programming. In *ACM SIGSOFT Software Engineering Notes*, volume 26, page 313. ACM, 2001. 26

[66] Olaf Spinczyk, Andreas Gal, and Wolfgang Schröder-Preikschat. Aspectc++: an aspect-oriented extension to the c++ programming language. In *Proceedings of the Fortieth International Conference on Tools Pacific: Objects for internet, mobile and embedded applications*, pages 53–60. Australian Computer Society, Inc., 2002. 28

[67] Olaf Spinczyk, Daniel Lohmann, and Matthias Urban. Aspectc++: an aop extension for c++. *Software Developer's Journal*, 5(68-76), 2005. 28

[68] Chris Lattner. Llvm: An infrastructure for multi-stage optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See* `http://llvm.cs.uiuc.edu`. 28

[69] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004. 28

[70] Homepage of the clang project. `http://clang.llvm.org`. 28

[71] Bjarne Stroustrup. *The C++ programming language*. Pearson Education, 2013. 59

[72] Torsten Mähne. *Efficient Modelling and Simulation Methodology for the Design of Heterogeneous Mixed-Signal Systems on Chip*. PhD thesis, École Polytechnique Fédérale de Lausanne (EPFL), EPFL/STI/IEL/LSM, Bâtiment ELD, Station 11, CH-1015 Lausanne, Switzerland. 60

[73] Alfred V. Aho et al. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006. 63

[74] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994. 67

[75] Joe J Monaghan. Smoothed particle hydrodynamics. *Reports on progress in physics*, 68(8):1703, 2005. 108

[76] Micky Kelager. Lagrangian fluid dynamics using smoothed particle hydrodynamics. *University of Copenhagen. Denmark*, 2006. 108

[77] Matthias Müller, David Charypar, and Markus Gross. Particle-based fluid simulation for interactive applications. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '03, pages 154–159, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association. 108

[78] David Staubach. Smoothed particle hydrodynamics, real-time fluid simulation approach. *University of Erlangen-Nuremberg. Germany*, 2010. 108

[79] Ernst Hairer, Christian Lubich, Gerhard Wanner, et al. Geometric numerical integration illustrated by the stormer-verlet method. *Acta numerica*, 12(12):399–450, 2003. 112

[80] Víctor Fernández, Andrés Mena, Cédric Ben Aoun, François Pêcheux, and Luis J Fernández. Virtual prototyping of pressure driven microfluidic systems with SystemC-AMS extensions. *Microprocessors and Microsystems*, 39(8):854–865, 2015. 119

[81] Ansys group. ANSYS CFX. https://www.ansys.com/. 124

[82] K Finkelzeller. The rfid handbook, 2003. 125

[83] Roy Want. An introduction to rfid technology. *IEEE Pervasive Computing*, 5(1):25–33, Jan 2006. 125

[84] Victor Fernandez, Elier Wilpert, Herique Isidoro, Cedric Ben Aoun, and Francois Pecheux. SystemC-MDVP modelling of pressure driven microfluidic systems. In *Embedded Computing (MECO), 2014 3rd Mediterranean Conference on*, pages 10–13, June 2014.

[85] Torsten Maehne, Zhi Wang, Benoit Vernay, Liliana Andrade, Cédric Ben Aoun, Jean-Paul Chaput, Marie-Minerve Louërat, François Pêcheux, Arnaud Krust, Gerold Schropfer, et al. Uvm-systemc-ams based framework for the correct by construction design of mems in their real heterogeneous application context. In *Electronics, Circuits and Systems (ICECS), 2014 21st IEEE International Conference on*, pages 862–865. IEEE, 2014.

[86] Liliana Andrade, Torsten Maehne, Alain Vachoux, Cédric Ben Aoun, François Pêcheux, and Marie-Minerve Louërat. Pre-simulation formal analysis of synchronization issues between discrete event and timed data flow models of computation. In *Proceedings of the Design Automation & Test in Europe (DATE) Conference 2015*, 2015.

[87] Liliana Andrade, Cédric Ben Aoun, Benoit Vernay, Torsten Maehne, François Pêcheux, and Marie-Minerve Louërat. Understanding the heterogeneous hardware: Do not forget the interconnection! In *2nd Workshop on Design Automation for Understanding Hardware Designs (DHUDe) at DATE*, 2015.

[88] Cédric Ben Aoun, Liliana Andrade, Torsten Maehne, François Pêcheux, Marie-Minerve Louërat, and Alain Vachoux. Generic eda-standard based elaboration scheme for the efficient monolithic simulation of heterogeneous systems. In *Work-in-progress session at 52th Design Automation Conference (DAC) 2015*, 2015.

[89] Cédric Ben Aoun, Liliana Andrade, Torsten Maehne, François Pêcheux, Marie-Minerve Louërat, and Alain Vachoux. Pre-Simulation Elaboration of Heterogeneous Systems: The SystemC Multi-Disciplinary Virtual Prototyping Approach. In *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XV), 2015 International Conference on*, July 2015.