FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Achieving Interoperability between *SystemC* and *System#*

Mário Lopes Ferreira

MSc DISSERTATION

Mestrado Integrado em Engenharia Eletrotécnica e de Computadores

Supervisor at FEUP: Prof. João Canas Ferreira

Supervisor at FZI: Christian Köllner

June 25, 2013

Abstract

This Master Dissertation is integrated in the *SimCelerate* project (result from a partnership between *FZI - Forschungszentrum Informatik*, *ITI GmbH* and *SET Powersystems GmbH*) whose main focus is the development of an automated approach for transforming a *Modelica* model into a hardware design for FPGAs. In the context, *FZI* developed a framework consisting of a *.NET* library intended for *high-level synthesis* of physical simulation for FPGA-based real-time execution, and for the description of real-time embedded systems (*system level description language*). The name of the framework is *System#*.

After *Modelica* compilation of a model, *System#* performs *high-level synthesis* and generates a RTL specification of the system, which can then be used for simulation and verification, prior to FPGA integration. As the simulation/verification loop with *System#* or VHDL generated code present some performance drawbacks, an alternative simulation procedure is needed. *SystemC* simulation appears to be a faster alternative, but demands the integration of automatic *SystemC* code generation on *System#*.

On the other hand, the technological advances on embedded systems caused the increasing of systems complexity and the emergence of heterogeneous systems, which can include subsystems designed with different languages. To co-simulate these subsystems together is a challenge in nowadays embedded systems project. Being also a recent SLDL, it would be beneficial to *System#* to have some degree of interoperability with a popular and widely-used SLDL such as *SystemC*. So, the main goals of this Master Dissertation are: *implementing automatic* SystemC *code generation from* System# *projects* and *developing a SystemC/System# co-simulation mechanism*.

System# already provides a synthesizable VHDL code generation engine. In order to enlarge its *high-level synthesis* capabilities, the VHDL code generation procedure was studied and adapted towards *SystemC* code generation. Facing the challenge of *SystemC/System#* co-simulation, it was divided in two parts: *communication* and *synchronization* between *SystemC* and *System#*. The communication part was implemented using an IPC mechanisms (*named pipes*), while the synchronization part was attacked with a conservative synchronization algorithm - an adaptation of the *conditional-event* approach.

The *SystemC* code generation developed cover the main *SystemC* constructs, the code generated for the tested projects showed to produce correct results and its simulation ran faster. The fact that the code generation is integrated in *System#* framework makes it easy to use during the *high-level synthesis* stage and the simulation/verification loop.

The co-simulation mechanism implemented proved to produce accurate simulation results and presents a modular nature, using various test models/systems. Thus, it is an initial contribution to the Interoperability between both SLDLs considered: *SystemC* and *System#*.

Keywords: System Level Description Languages, heterogeneous systems, *SystemC*, *System#*, *SystemC* code generation, parallel/distributed discrete-event co-simulation

Resumo

Esta Dissertação surge no âmbito do projeto *SimCelerate* (fruto de uma parceria entre o *FZI - Forschungszentrum Informatik*, a *ITI GmbH* e a *SET Powersystems GmbH*), cujo principal objetivo é a automatização do processo de transformação de um modelo descrito em *Modelica* numa descrição de hardware a implementar numa FPGA. Neste contexto, o *FZI* desenvolveu uma plataforma (*System#*) que consiste numa biblioteca *.NET* destinada à *síntese de alto nível* (*high-level synthesis*) de simulações físicas em tempo real usando FPGAs, bem como à descrição de sistemas embarcados que operam em tempo real (*system level description language*).

Após a compilação em *Modelica*, o *System#* executa *high-level synthesis* e gera uma especificação RTL do sistema, que pode ser depois utilizada para a simulação e verificação do sistema antes da sua integração numa FPGA. Como a execução do ciclo de simulação/verificação com *System#* ou *VHDL* apresenta uma performance lenta, um procedimento de simulação alternativo seria conveniente. A simulação com *SystemC* aparenta ser uma alternativa mais rápida, mas requer a integração de geração automática de código *SystemC* na plataforma *System#*.

Por outro lado, os avanços tecnológicos na área dos sistemas embarcados causaram um aumento da complexidade dos sistemas e provocaram a emergência dos *sistemas heterogéneos*, que podem incluir subsistemas projetados em linguagens diferentes. Cossimular estes subsistemas é um desafio atual no projeto de sistemas embarcados. Sendo também uma SLDL recente, seria benéfico para o *System#* ter algum grau de interoperabilidade com uma SLDL largamente utilizada como o *SystemC*. Assim, os objetivos principais desta Dissertação são: a *implementação da geração automática de código SystemC a partir de projetos desenvolvidos em System#* e o *desenvolvimento de um mecanismo de cossimulação entre SystemC e System#*.

O *System#* já inclui um mecanismo de geração de código *VHDL* sintetizável. Para alargar as suas capacidades de *high-level synthesis*, o mecanismo de geração de código *VHDL* foi estudado e adaptado para a geração de código *SystemC*. A cossimulação entre *SystemC* e *System#* foi dividida em duas partes: *comunicação* e *sincronização* entre *SystemC* e *System#*. A parte de comunicação foi implementada recorrendo a um mecanismo de IPC (named pipes), enquanto que a sincronização foi atingida com um algoritmo conservativo - uma adaptação do algoritmo *conditional-event*.

A geração de código *SystemC* desenvolvida cobre os principais conceitos do *SystemC*. O código gerado para os sistemas de teste produziu resultados corretos e a sua simulação mostrou ser mais rápida. O facto de o mecanismo de geração de código *SystemC* estar integrado na plataforma *System#* facilita o seu uso durante a fase de *high-level synthesis* e durante o ciclo de simulação/verificação.

O mecanismo de cossimulação implementado produz resultados de simulação corretos para os sistemas testados e apresenta uma natureza modular. É, portanto, uma contribuição inicial para a interoperabilidade entre as SLDS consideradas: *SystemC* e *System#*.

Palavras-chave: System Level Description Languages, sistemas heterogéneos, *SystemC*, *System#*, geração de código *System#*, cossimulação de eventos discretos paralela/distribuída

Acknowledgements

My first acknowledgement words go to my family. To my parents for their unconditional and constant love and support. Without them, this dissertation would not have been possible. Also, a special gratitude to my grandparents for all the intangible knowledge they gave me and for the life examples they represent to me.

I would like to thank to my university friends with who I shared the ups and downs of the academic path. This experiences exchange resulted in a sharp contribution to my personal development.

I would like to express my appreciation and gratitude to my supervisors, Christian Köllner and João Canas Ferreira, for their support, availability and patience. Their guidance and suggestions were determinant to this dissertation.

Thank you very much! Muito Obrigado!

Mário Lopes Ferreira

“Tudo neste mundo tem uma resposta.
O que leva é tempo para se formular as perguntas.”

“Everything in this world can volunteer some reply,
what takes up time is posing the questions.”

José Saramago

Contents

List of Figures

List of Tables

Abreviaturas e Símbolos

AST	Abstract Syntax Tree
CIL	Common Intermediate Language
DC	Direct Current
DES	Discrete-Event Simulation
EME	Electric Motor Emulator
ESL	Electronic System Level
FPGA	Field-Programmable Gate Array
FSM	Finite State Machine
FZI	*Forschungszentrum Informatik* - Research Center for Information Technology
HDL	Hardware Description Language
HiL	Hardware-in-the-Loop
HLA	High-Level Architecture
HLS	High-Level Synthesis
I/O	Input/Output
IP	Intellectual Property
ISS	Instruction Set Simulator
LP	Logical Process
MIMD	Multiple Instruction, Multiple Data
P-HiL	Power Hardware-in-the-Loop
RTL	Register-Transfer Level
SD	Simulation Domain
SIMD	Single Instruction, Multiple Data
SLDL	System Level Description/Design Language
SoC	System-on-Chip
TNLE	Time-to-Next-Local-Event
TNRE	Time-to-Next-Remote-Event

Chapter 1

Introduction

In this introductory chapter, the context as well as the motivation which sustain the developed work are presented. A document structure description is also given in the end of the chapter.

1.1 SimCelerate project contextualization

This Master dissertation is part of the Integrated Master in Electrical and Computers Engineering (Mestrado Integrado em Engenharia Electrotécnica e de Computadores) from the Faculty of Engineering of the University of Porto (Faculdade de Engenharia da Universidade do Porto). The work covered by this dissertation is within the scope of the *SimCelerate* project which results from a partnership between *FZI - Forschungszentrum Informatik - Research Center for Information Technology* (based in Karlsruhe), *ITI GmbH* (based in Dresden) and *SET Powersystems GmbH* (based in Wangen). Before diving on the details about the developed work, a general contextualization of the *SimCelerate* project will be done.

In the automotive industry, testing electrified drive-trains of a vehicle is a very important part regarding the device safety. So, during development phase, electrified drive-trains are exhaustively tested in order to figure out whether the system meets the desired requirements and behaviour. One can test an algorithm using a purely mathematical model describing the system behaviour. However, another approach has gained popularity in the automotive industry within the last 25 years: *Hardware-in-the-loop* (HiL) simulation [5]. Shortly, HiL simulation consists in the integration of a part of the real hardware in the simulation loop. HiL simulation also makes it possible to perform real-time simulation. Considering the case of testing an electronic motor controller, one can use the real controller, in order to achieve a higher accuracy, and replace the surrounding environment (motor and vehicle mechanics) by a virtual device - the Electric Motor Emulator (EME). Usually, the starting point for the creation of an EME is a Modelica description of the system one wants to simulate. Modelica [6] is an object-oriented, equation-based language intended for modelling of complex systems of different kinds (e.g. electrical, electronic, mechanical, control, thermal, hydraulic). In Modelica, models are mathematically described using differential, algebraic and discrete equations. A key difference of Modelica regarding to typical object-oriented programming languages has to do with the compilation process: Modelica classes are translated into objects and afterwards theses objects are operated by the simulation engine. The industry provides EMEs covering synchronous, asynchronous and DC machines with power up to 300kW [7]. These emulators also allow the recreation of the high voltages and currents (*Power Hardware-in-the-loop*, P-HiL), running at sampling rate up to 800kHz. Consequently, a real-time simulation of an electric motor running at a cycle time around 1μs is needed. This real-time requirement is hard to achieve with processor-based systems due to the unpredictable time overhead introduced by the I/O operations required for the HiL simulation. So, the state-of-the-art emulators execute the motor simulation on an FPGA. The great concern regarding to the Modelica-to-FPGA design flow is the high engineering effort needed to execute it. Consequently, the entire process becomes very expensive. It was in this context that the *SimCelerate* project arose. This project focuses on the development of an automated approach for transforming a Modelica model into an hardware design for FPGAs. The main emphasis is given to the domain of electric motor controllers. The key aspects of this project are:

- Design of hardware architectures for efficient real-time calculation of numerical simulations;
- Identification and implementation of domain-specific hardware modules (e.g. parallel solution of Systems of Linear Equations);
- Design of an integrated transformation process.

In order to face the challenge of automating the Modelica-to-FPGA design flow, one framework was developed by FZI: *System#*. This framework is one of the most important domains studied and used during the Master dissertation development. As a first reference, *System#* is a .NET library which is intended for both high-level synthesis of physical simulation for FPGA-based real-time execution and for the description of real-time embedded systems.

1.2 Motivation

The general approach used in *SimCelerate* project for the Modelica-to-FPGA procedure comprises 3 main stages: Modelica compilation, high-level synthesis and FPGA design implementation [1]. High-level synthesis is executed by *System#* and during this stage a complete RTL system specification is produced. Then, it can be used for system verification and simulation or synthesizable VHDL generation. Before FPGA system implementation, the design should be iteratively simulated and verified. It is possible to do it using *VHDL* or *System#* but both of them are not satisfactory in terms of performance. As it is a recent framework, *System#* can still be improved in order to render a better simulation performance. Indeed, some work is currently being done at FZI seeking for these improvements. However, it would be also beneficial to have an alternative to VHDL and *System#* simulation. In turn, this alternative should be integrated in the high-level synthesis tool - *System#* - so that the design flow is kept automated.

Although *System#* is being referred as a high-level synthesis framework it is also intended for the description of real-time embedded systems, as stated in the end of the previous section. *System#* provides constructs and proper design rules needed for designing and modelling complex electronic systems from the scratch. In other words, *System#* can also be viewed as a *system level description language* (SLDL) based on the .NET framework. In the past decades, technological progresses in the field of embedded systems made it possible to produce more complex systems which wrap different subsystems. These subsystems can vary in function and form (hardware or software) - *heterogeneous systems*.

In this context, SLDLs arose as well-suitable platforms for designing, modelling and simulating complex electronic systems. Several SLDLs are available nowadays, being *SystemC* the most popular and accepted one among industry and academy. However, given the increasing complexity and heterogeneity of embedded systems, it is plausible that the different system subcomponents are modelled and simulated individually, using different languages or frameworks. Nevertheless, global simulation and verification of heterogeneous systems assumes particular importance during the project and production cycle. So, interoperability and co-simulation mechanisms between different subsystems composing a broader system are needed. The targeted SLDLs in this Master dissertation are *SystemC* and *System#*.

1.3 Document Structure

Apart from the introductory chapter, this document includes other seven chapters. In Chapter 2, a description of the problem and objective which sustain the Master Dissertation are presented. Furthermore, a short overview of the methodology adopted is shown together with a list of tasks performed and technologies used. A bibliographic revision is made in Chapter 3, presenting the fundamental knowledge required and the state-of-the-art of co-simulation and interoperability between discrete-event simulators. Chapter 4 and Chapter 5 target the implemented work within this Master Dissertation: *SystemC* code generation from *System#* projects and co-simulation/interoperability between *SystemC* and *System#*, respectively. After, the implemented work is evaluated and results are shown - Chapter 6. Chapter 7 focuses on conclusions, contributions from the developed work and future work proposals. Then, Appendix A shows some code examples for co-simulation. Finally, bibliographic references are presented.

Chapter 2

Problem and Objectives

This chapter addresses the problem targeted by this Master dissertation, its objectives and general considerations on the methodology adopted. First, a more detailed description of the problem is presented. Then, the intended objectives of this work are exposed. Finally, remark about methodology, tasks to perform and technologies and tool used during the developed work are presented.

2.1 Problem Description

In section 1.2, the motivation behind the developed work was shortly presented. In this section, a more detailed overview of the targeted problem is exposed.

In figure 2.1, there is an overview of the *SimCelerate* project approach for the Modelica-to-FPGA design flow. It is possible to identify three main stages.

The first stage culminates with the Modelica compilation, resulting in a calculation rule which includes all computations required to execute a single integration step. The Modelica compiler employs in-line integration [8] to produce a compact calculation rule suitable for hardware mapping. In this case, the intermediate representation resulting from Modelica compilation is not *C* code, but a XML-based format consisting of: system interface description, control and dataflow graph in assembler-like language and special support for fixed-point arithmetic and mathematical functions.

It is this intermediate representation produced by the Modelica compiler which is the input of the second stage of the Modelica-to-FPGA design flow: *high-level synthesis*. This stage is performed by *System#*. After assigning instructions to clock cycles, binding them to hardware resources and allocating data transfers to intermediate storage registers, a control path is constructed. In turn, this control path can provide a complete RTL specification of the system intended for simulation. This RTL specification can be used for verification and simulation as well as for synthesizable *VHDL* generation.

The third and last stage of the design flow consists of simulation and verification of the system description provided by the high-level synthesis, and also the design implementation on an FPGA. There are two alternatives to perform system simulation and verification. One consists of taking

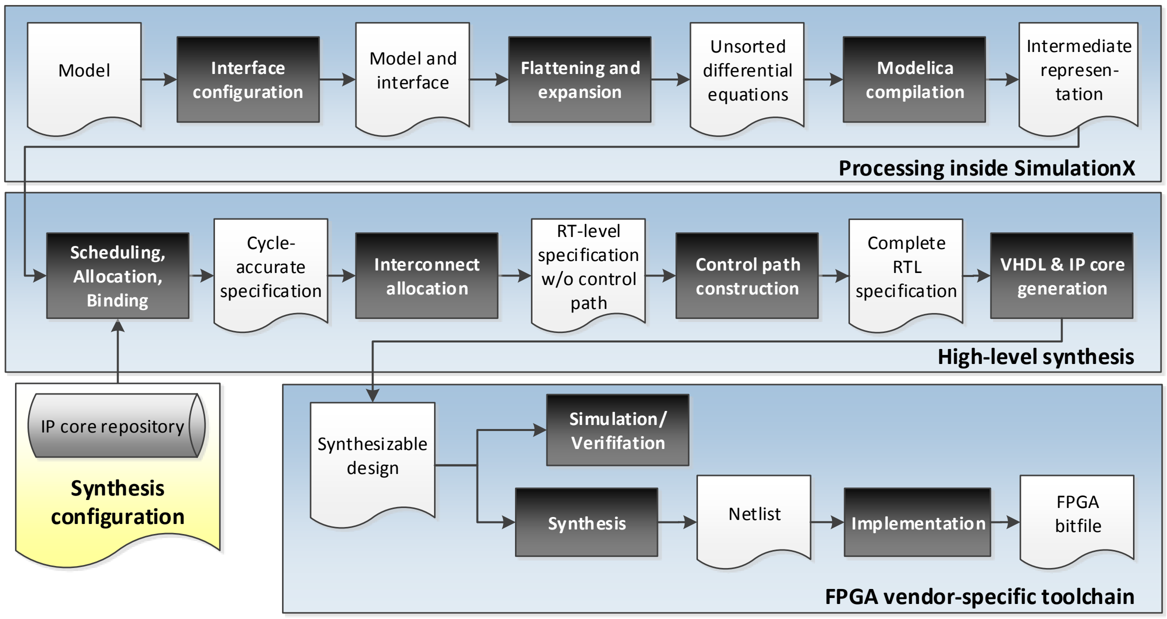

Model
Interface configuration
Model and interface
Flattening and expansion
Unsorted differential equations
Modelica compilation
Intermediate represen-tation
Processing inside SimulationX
Scheduling, Allocation, Binding
Cycle-accurate specification
Interconnect allocation
RT-level specification w/o control path
Control path construction
Complete RTL specification
VHDL & IP core generation
High-level synthesis
IP core repository
Synthesis configuration
Synthesizable design
Simulation/ Verifiation
Synthesis
Netlist
Implementation
FPGA bitfile
FPGA vendor-specific toolchain

Figure 2.1: *Modelica-to-FPGA* design flow [1]

the previously produced RTL specification and simulating it using *System#*. The other consists of performing *VHDL* code generation from the RTL specification (using *System#*) followed by *VHDL*-based simulation and verification.

One practical problem arises from the described design flow. Before integrate the design on an FPGA, it is desirable to be able to simulate and verify it in an expedited manner. However, the *VHDL*-based simulation is slow and requires user interaction. On the other hand, the *System#*-based simulation can be done automatically but it is even slower, as it is a recent framework not fully optimized.

One possible approach to address this problem can be the automatic generation of a *SystemC* model which is used for exhaustive numerical verification. Once the verification results are acceptable, VHDL code is generated and a new verification procedure is done but now focused on aspects such as timing and I/O operations. The advantage of using a *SystemC* model for simulation and verification resides on the higher performance of *SystemC* simulator, compared with *System#*/VHDL-based simulation, and only a *C++* compiler is required to do so. In order to keep the complete design flow automated, the *SystemC* code generation should be integrated in the high-level synthesis tool - *System#*.

From a SLDL domain perspective, another problem arises. The increasing complexity of embedded systems may require the use of different languages or Models of Computation (MOC) to design a whole heterogeneous system. If one wants to simulate and verify the whole system, a homogeneous approach can be taken by using a single language to describe the whole system [9] [10]. However, it is hard to have a language able to cover the semantics of all MOCs in a given embedded system.

The heterogeneous approach describes the models in their native languages, keeping the conceptual differences of each domain [11]. Considering a system composed by *SystemC* and *System#* modules, the challenge consists of developing and validating concepts of *SystemC/System#* co-design and co-simulation.

Additionally, *SystemC* has a large community of users and associated software tools, contrasting with the recent *System#*. So, some degree of interoperability between both domains would clearly improve *System#*'s perception.

2.2 Objectives

Considering the problem described before, the work developed within this Master Dissertation aims the development and validation of co-design and co-simulation concepts between two SLDLs - *SystemC* and *System#* - in order to achieve some degree of interoperability between projects and/or components designed in both frameworks. Thereafter, this Master dissertation has two main objectives:

- **Implement *SystemC* code generation from *System#* projects:** from a high-level synthesis tool perspective, *System#* must be able to automatically generate *SystemC* code from an elaborated *System#* model. As the generated code may be used to speed-up the simulation and verification loop, it should provide a correct description of the original system and thus, produce correct simulation results;

- **Develop a *SystemC/System#* co-simulation mechanism:** both simulation domains - *SystemC* and *System#* - must be able to communicate and transfer data related with events affecting the remote domain. Apart from it, the co-simulation mechanism requires time synchronization between the simulation domains in order to the coherency and accuracy of the process. This way, the co-simulation environment comprises a communication mechanism and a synchronization algorithm.

2.3 Methodology

Having in mind the objectives stated in section 2.2, a brief description of the methodology used to attack the problems referred in section 2.1 is now presented.

As stated before, the *SystemC* code generation engine must be integrated in *System#* platform. So, the chosen approach was the study of the VHDL code generation engine already present in *System#* and its adaptation towards *SystemC* code generation. The study of the code generation engine required the understanding of how concepts like components, ports, channels or processes are internally represented in *System#*. The particular study of the *VHDL* code generation procedure was also useful through the identification and comparison of common elements and concepts between *VHDL* and *SystemC*, and the verification of how the *VHDL* syntax for those elements or concepts is produced. However, as *SystemC* is a broader language than *VHDL*, the implementation of procedures for the syntax generation of specific *SystemC* concepts was also required. Other important issues were the data types equivalence and compatibility.

Regarding to *System#/SystemC* co-simulation, the problem was divided in two parts: *communication between SystemC and System#* and *synchronization between SystemC and System#*. The first part has to do with the IPC mechanism used and the characteristics of the data transferred between both domains. The second part is related to the algorithm that controls and synchronizes the simulation flow in both domains.

When deciding which approaches to choose for both communication and synchronization problems, a trade-off between available work time, complexity and performance was taken into account. So, a review of the available APIs for IPC mechanisms in the used operating system was done. With respect to simulation coupling and synchronization, it was decided to avoid the modification of the SLDLs kernels, for sake of compatibility between different versions. Given the time constraints imposed for the Master dissertation development, the adopted synchronization was of conservative nature. On the other hand, this approach does not require as many memory and processor resources as the optimistic approaches.

2.4 List of Tasks

The Table 2.1 enumerates and describes the tasks which composed the work developed. These generic tasks formed a guideline to better organize and manage the available work time.

Task ID	**Task Description**
1	Study of fundamental knowledge and state-of-the-Art
2	*SystemC* code generation engine implementation
3	Design of a *SystemC/System#* co-simulation environment comprising communication and synchronization procedures
4	Implementation of the co-simulation environment designed
5	Evaluation of code generation engine and/or co-simulation engine correctness and performance
6	Thesis writing

Table 2.1: List of tasks performed during the project

2.5 Technologies and Tools

For the development of this Master dissertation, a computer equipped with a *Microsoft Windows* operating system was used. As the two main domains of this project, the SLDLs *System#* and *SystemC* (version 2.3) were tools exhaustively used. The use of *System#* also requires the use of the *.NET* framework. The *Windows* APIs for IPC was employed.

The IDE used for programming tasks within this project was the *Microsoft Visual Studio* (2012 edition). At the same time, the documentation related with this Master dissertation was produced in LaTex.

Chapter 3

Background Information and State of Art

In this chapter, fundamental knowledge about SLDL (particularly *SystemC* and *System#*) and discrete-event simulation is presented. Co-simulation and interoperability of discrete-event simulators state-of-the-art and related work is also referred.

3.1 *System Level Description Languages*

Recent trends in SoC projects move towards an increasing complexity of systems composed by hardware and software components and the need of third-party IP integration. Under this scenario, the traditional hardware/software co-design approach, in which languages like *C/C++* are used for software design and HDLs are used for hardware design, introduces a big separation between both design levels. This situation is difficult to overtake if one wants to generate synthesizable code. This way, the gap between the system complexity and the engineering effort required for the system conception - *productivity gap* - substantially increases.

It was in this context SLDL emerged as tools or frameworks intended for complex electronic systems modelling, design and verification. These languages provide a suitable environment for the co-design of hardware and software components in an heterogeneous system, supplying appropriate data types, libraries and simulation mechanisms, as well as supporting several abstraction levels like RTL (typically used by HDLs) and ESL.

Several SLDLs were presented and introduced over last years, being *SystemC* the most widely used. *SpecC* [12] is another SLDL example. Although they are closer to *hardware verification languages* than to SLDLs, platforms as *SystemVerilog* [13], *OpenVera* [14] and *e* [15] try to present some typical features from object-oriented programming languages, while keeping typical properties from HDLs [16]. For example, in *SystemVerilog* there is an extension of *Verilog* functionalities, being *Verilog* one of the most largely used HDLs together with *VHDL*.

Recently, taking advantage of the features provided by *.NET* [17] software development framework, specially the ability of a program to observe and/or modify its structure in run time - *reflection* - [18] new SLDL proposals have been presented. Examples of these new SLDLs are *System#* and *ESys.NET* [19].

In the scope of this Master dissertation, *SystemC* and *System#* are the studied and used SLDLs.

3.1.1 *SystemC*

SystemC [20] consists of a *ANSI C++* library intended for electronic systems design, modelling and verification. Beyond inherited features from *C++* (Object-oriented language) which allow software modelling, *SystemC* also provides adequate concepts and constructs for hardware modelling (modular hierarchy, notion of time and concurrent execution, hardware data types) and system architecture modelling (interfaces and communication channels, models with several abstractions levels - Gate level, RTL and ESL).

According to the report elaborated by *Doulos* [21], *SystemC* is mainly used in performance modelling, architectural exploration, *Transaction-level Modelling* [22] and hardware/software co-simulation. Nevertheless, *SystemC* is also widely used for verification and testbench creation, surpassing some languages/platforms specially directed for this kind of tasks, as *Vera* and *e*.

The *SystemC* announcement and presentation was done in 1999 by the *Open SystemC Initiative - OSCI* and since then, the popularity of this tool considerably increased. This is visible due to the big community of users, both in industry and academy, and the variety of support tools available. Several elements contributed to the big *SystemC* popularity. Besides of having been accepted as an IEEE norm (IEEE 1666®), *SystemC* includes an *Open Source* simulator which only requires a *C++* compiler to run (ensuring high flexibility regarding to the Operating System in which it is used) and is easily integrable with *C/C++*.

Next, some contents extracted from [20] will be exposed, aiming a better understanding of *SystemC* structure and functionalities.

In order to effectively deal with systems complexity, *SystemC* calls upon the separation between functionality (specification level) and architecture (implementation level) and between computation and communication. Towards the computation/communication separation, computation is performed by *processes* associated to system *modules*, while communication is ensured by communication *channels*. The connection between both domains is done by the connection of *ports*, contained in *modules*, to *interfaces* provided by *channels*.

SystemC library includes classes which fall in one of four categories: *Core Language*, *Predefined channels*, *Utilities* and *SystemC Data Types*.

Among the most used functional constructs in *SystemC*, one can highlight: *modules*, *processes*, *ports*, *events*, *channels* and *interfaces*.

A *module* (macro *SC_MODULE*) is the basic structural block of a system described in *SystemC*. A generic system is constituted by a hierarchy of modules which are connected. A module can contain ports, processes, events, channels and other modules.

Processes describe a module functionality and present concurrent execution. There are three types of processes: *SC_METHOD* (similar to *C++* functions; executes when a certain condition is verified and its execution cannot be interrupted), *SC_THREAD* (executes only once, but can be kept active through the inclusion of an infinite loop; can be interrupted) and *SC_CTHREAD* (its execution is similar to *SC_METHOD* one, but it is triggered by clock impulses).

Ports supply an abstraction which permits the isolation of a module implementation from the surrounding environment. They allow the communication between modules or channels and can be unidirectional or bidirectional. The most common port types are the unidirectional *built-in* ports *sc_in* and *sc_out*. Still, it is possible to create and define ports according to the project needs.

Events (*sc_event*) consist of flexible low-level synchronization primitives. They can be used to build other forms of synchronization.

SystemC provides some *built-in channels* which can model signals (*sc_signal*), clock signals (*sc_clock*), mutexes (*sc_mutexes*), semaphores (*sc_semaphore*) or FIFO queues (*sc_fifo*). Together with these primitive channels, interfaces which define a set of channel access methods are available, making channels useful for communication between modules. There is also the possibility to define hierarchic channels, in order to model more complex behaviours. Shortly, a hierarchical channel consists of a module which implements methods declared in an interface.

With respect to data types, *SystemC* supports native *C/C++* data types, as well as specific data types suitable to system modelling: integers of variable dimension, logic values, logic values vectors or fixed-point numerical values. Table 3.1 reproduces a table from [20] which shows the *SystemC* specific data types.

Class Template	**Base class**	**Generic base class**	**Representation**	**Precision**
sc_int	sc_int_base	sc_values_base	signed integer	limited
sc_uint	sc_uint_base	sc_values_base	unsigned integer	limited
sc_bigint	sc_signed	sc_values_base	signed integer	finite
sc_biguint	sc_unsigned	sc_values_base	unsigned integer	finite
sc_fixed	sc_fix	sc_fxnum	signed fixed-point	finite
sc_ufixed	sc_ufix	sc_fxnum	unsigned fixed-point	finite
sc_fixed_fast	sc_fix_fast	sc_fxnum_fast	signed fixed-point	limited
sc_ufixed_fast	sc_ufix_fast	sc_fxnum_fast	unsigned fixed-point	limited
		sc_fxval	fixed-point	variable
		sc_fxval_fast	fixed-point	limited-variable
	sc_logic		single bit	
sc_bv	sc_bv_base		bit vector	
sc_lv	sc_lv_base		logic vector	

Table 3.1: *SystemC* specific data types

The execution of a *SystemC* application comprises two fundamental steps: *elaboration* and *simulation*. During elaboration, a hierarchical structure of the system is created, including the instantiation of modules and channels, as well as the connection between ports and channels. Once the structure is created, it cannot be changed during the rest of elaboration and simulation. The simulation is performed by *SystemC scheduler*. During this step, the scheduler manages the execution of different processes, creating the illusion of a concurrent execution. The beginning of simulation is triggered by calling the function *sc_start()*. To perform elaboration and simulation, it is required to run the application code and the *SystemC* simulation kernel code, which consists of a part of classes from *SystemC* library where the scheduler and fundamental elaboration functionalities are implemented.

SystemC simulation kernel implements a discrete-event simulation system and presents a non-preemptive task manager mechanism - *cooperative multitasking*. This simulation kernel presents some features similar to the ones from *VHDL* simulation mechanism like, for example, the deterministic behaviour of the simulation (the simulation result must not depend on the order by each process is executed, for a given simulation step). The basis for the deterministic behaviour is the concept of *delta cycle*. A *delta cycle* consists of a scheduler control cycle comprehending an evaluation phase followed by an update phase. It is the alternation between computation (evaluation phase) and communication (update phase) that guarantees the deterministic behaviour. A *delta cycle* is a zero-time step. In other words, it does not contribute to the simulation time advancement.

The simulation features referred before are typical from hardware simulation. However, as *SystemC* aims to be a language suitable for hardware/software systems modelling, its simulation kernel also exhibits some non-deterministic features which allow correct software modelling. The use of events and notifications between processes and primitive channels intended for critical sections access control are examples of features which add non-determinism to *SystemC* simulation kernel.

Then, a brief description of each execution step performed by *SystemC* simulation kernel is presented [23]:

- **Initialization:** execution of all processes (except SC_CTHREADs) in an undefined order until they reach the first synchronization point (if any);

- **Evaluation:** a process ready to run is selected and its execution is resumed. This can cause immediate event notifications which may turn some processes ready to run;

- The previous step is repeated until there are no more processes ready to run;

- **Update:** execution of all pending update requests (calls to *update()*), as a result from calls to *request_update()* made during last evaluation phase;

- If event notifications made during current *delta cycle* exist, processes are inspected in order to know which ones are ready to run due to all those events. Proceed to another evaluation phase;

- If there are no timed events, simulation is finished. Otherwise, simulation time is advanced to the time of the earliest pending timed event notification;

- Determination of which processes are ready to run as consequence of all timed events (current time) and execution of another evaluation phase.

Below, figure 3.1 presents a flowchart illustrating the scheduler execution.

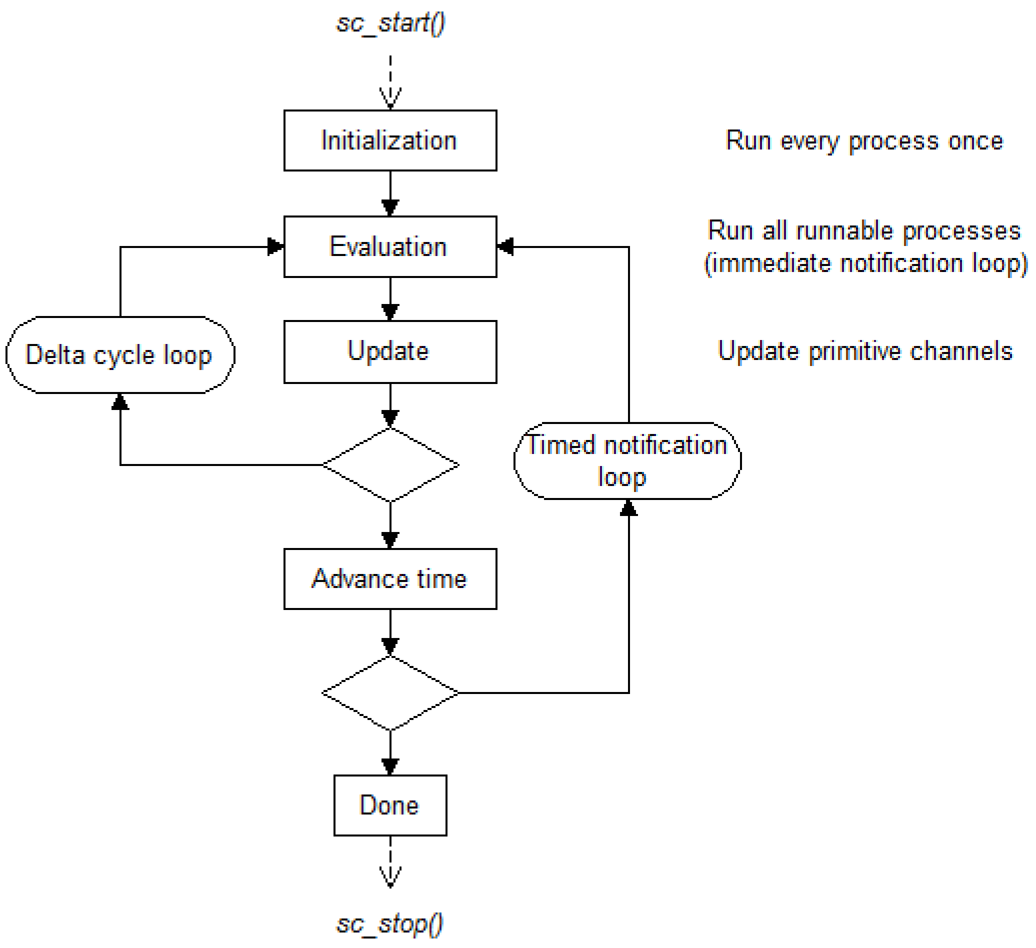


Figure 3.1: *SystemC* scheduler operation flowchart [2]

3.1.2 *System#*

As stated in Section 1.2, *System#* is a high-level synthesis framework and also a SLDL. In this section, the main features and constructs of *System#*, as an SLDL, will be targeted.

System# consists of an *open source C#* library (*.NET* framework) which provides structures suitable to the description and simulation of real-time embedded systems. Being a platform which addresses the design of FPGA-based systems, *System#* includes a mechanism for synthesizable VHDL code generation. This SLDL supports RTL descriptions as well as some synthesizable TLM concepts. *System#* was presented in 2012 [4] by the *Embedded Systems and Sensors Engineering (ESS)* research division of FZI, within the scope of *SimCelerate* project. Thus, it is a recent tool and, consequently, is not so mature as *SystemC*.

Regarding to the synthesizable VHDL generation, *System#* makes use of *.NET* framework typical features - reflection and decompilation of assembly code (CIL) - to build an AST, taking into account the system structure. In turn, the built AST obeys to a *Document Object Module* - DOM - for describing component-based reactive systems: *SysDOM*. The transition from an AST to VHDL code is done through an unparsing process.

Modelling a system with *System#* is similar to do it with *SystemC*. There is also the base concept of separation between computation (modules) and communication (channels). As it was previously done with *SystemC*, basic concepts and functionalities which allow systems description and modelling with *System#* will be presented. Most of these contents were extracted from [24].

A system described in *System#* typically presents a hierarchy of components which represent and describe the operation of some system parts. A *component* consists of a *C#* class derived from *Component* (namespace *SystemSharp.Components*) and can contain instances of other components in the system.

The interaction between a component and the surrounding environment is done through *ports*. In *System#*, ports are actually *C#* properties. Examples of input and output ports are *In<data_type>* and *Out<data_type>*, respectively. Ports merely represent a communication interface between a component and the exterior. In order to perform this communication, ports must be connected to *channels*.

By storing information exchanged between ports, a channel models the concept of communication and is represented in *System#* by the abstract class *Channel*. This class is used in the definition of every kind of channel. One of the most common channels in *System#* is *Signal<data_type>*, which represents a signal.

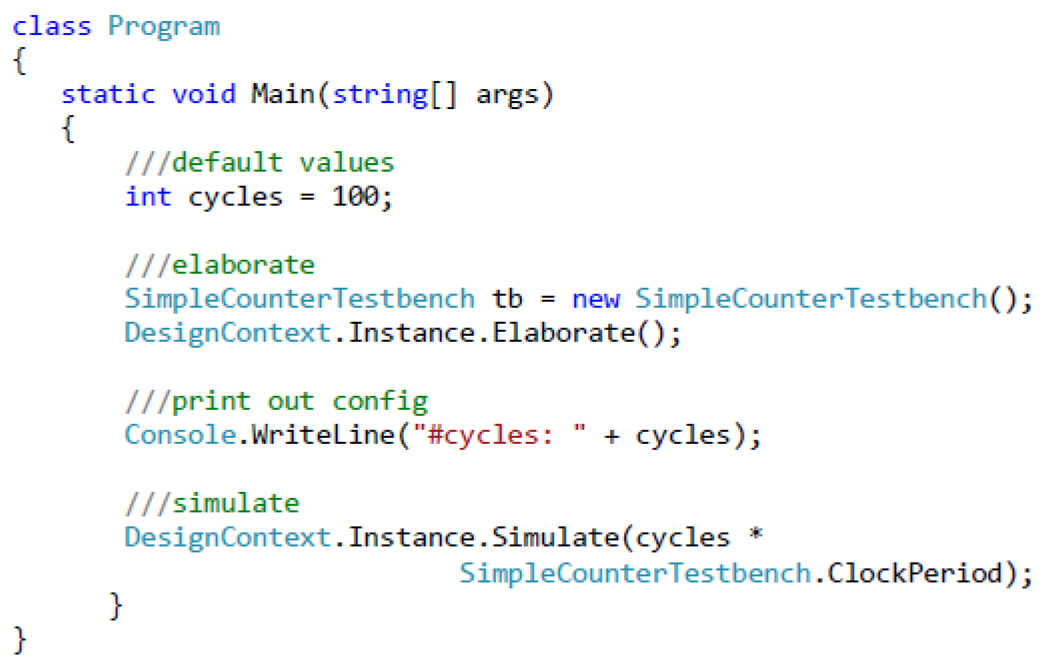
As in *SystemC*, a *process* describes the module functionality. In a system, processes execute concurrently. There are three types of processes in *System#*: *Triggered process* (executes due to some event(s) described in a sensitivity list; it consumes no simulation time; its registration is done through *AddProcess()*), *Threaded process* (starts its single execution in the beginning of simulation; can be pause due to some event and is registered with *AddThread()*) and *Clocked thread* (a special thread intended for synthesizable synchronous FSMs modelling; is sensitive to an edge of a clock signal; *AddClockedThread()* registers this kind of processes).

System# also provides data types suitable for embedded systems modelling, namely bit-accurate data types and resolved logic. These hardware data types belong to the *SystemSharp.DataTypes* namespace.

Resolved logic data types in *System#* are *StdLogic* and *StdLogicVector*. *StdLogic* is a nine-value data type equivalent to the *std_logic* data type in *VHDL* and *StdLogicVector* is simply a vector whose elements are from *StdLogic* type.

Regarding to bit-accurate data types, arbitrary-length integers can be represented by the types *Signed* and *Unsigned*, while fixed-point arithmetic is provided by data types *SFix* and *UFix*.

As in *SystemC*, the execution of a *System#* application includes elaboration phase and simulation phase. During elaboration, the system design is not only prepared for simulation, but also for analysis and program transformation (for example: *VHDL* code generation). This phase is triggered by the instruction *DesignContext.Instance.Elaborate()* and after that, the system structure or functionality cannot be changed. Simulation is performed by the instruction *DesignContext.Instance.Simulate()*. An example for the call of *Simulate/Elaborate functions* is presented in Figure 3.2 *System#* simulation kernel also performs discrete-event simulation and its operation is similar to the one in *SystemC*. The main difference resides in the management of thread processes: while in *SystemC* we have cooperative multitasking, *System#* maps thread processes to tasks through the *Task Parallel Library* from *.NET* framework. So, *System#* simulation kernel assume characteristics of a parallel discrete-event simulator (PDES).



```
class Program
{
    static void Main(string[] args)
    {
        ///default values
        int cycles = 100;

        ///elaborate
        SimpleCounterTestbench tb = new SimpleCounterTestbench();
        DesignContext.Instance.Elaborate();

        ///print out config
        Console.WriteLine("#cycles: " + cycles);

        ///simulate
        DesignContext.Instance.Simulate(cycles *
                                  SimpleCounterTestbench.ClockPeriod);
       }
}
```

Figure 3.2: Calls to Elaboration and Simulation in *System#*

In terms of performance, *System#* simulation is slower than *SystemC* one. Being *System#* a young tool, some efforts within *SimCelerate* project are currently in course aiming a better simulation performance. At the time this document writing, some work on using the new *C#* 5.0 features to improve *System#* performance is in progress.

Apart from taking advantage of *.NET* features like reflection and providing VHDL code generation, *System#* presents other distinctive features comparing with other SLDLs: synthesizable sequential descriptions and IP-based Design. The first of these features is related with the usual need for FSMs specification in FPGA design. Sometimes this task can become troublesome. Facing this, *System#* enables the generation of synthesizable FSMs from clocked threads. If one wants to derive a FSM from a cloked thread, the *C#* attribute *TransformIntoFSM* must be placed before the method implementation.

Usually, FPGA vendors furnish their FPGA design tools with an IP core library. Having this IP cores in mind, the designer have to customize parameters and store them in a proprietary file format. Right now, *System#* only supports a limited set of cores from the *Xilinx* FPGA toolchain. However, *System#* can be extended in order to support other FPGA vendors tool-chains.

3.2 Discrete-Event Simulation

During the process of designing and evaluating a system, one may want to know how the system will react under certain circumstances in order to figure out if it serves the purposes for what it is being designed. One of the most broadly used techniques of imitating the operation of a system over time is simulation. To simulate a system, an abstraction is needed and it usually takes the form of a model. In turn, a model describes and represents the target system through mathematical or logical relationships. Often during a system project, there is a big interest in studying the system response to time constraints. This way, the evolution of the system state must be well defined in the model.

If the state of the system one wants to simulate changes continuously over time, it is classified as a *continuous system*. On the other hand, if the system state only changes at specific points of time, it is called a *discrete system*. There are also *hybrid systems* comprising continuous and discrete components.

Taking into account how the system state evolves over time, suitable simulation progression of time must be implemented. Therefore, there are *time-driven* (or *time-stepping*) simulation and *event-driven* (or *discrete-event*) simulation. In the former, time is incremented and measured in small steps, giving the illusion of a continuous time evolution; in the later, time jumps through different points of time (*events*), making the system operation to be a chronological sequence of events. By their definitions, it is possible to discern that time-driven simulation suits better to continuous systems, whereas discrete-event simulation suits better to discrete systems.

Due to the context of this dissertation and the *SystemC/System#* simulation nature, special attention will be given to discrete-event simulation (DES).

A discrete-event simulation has 3 main elements: *state variables*, an *event-list* and a *clock variable* [25]. The state variables store information about the state of the system. The event list is a data structure basically consisting of a priority queue that orders events according to their scheduled time and the clock variable keeps track of current simulation time.

The discrete-event simulation operation is described by the simulation cycle. Though it may vary from simulator to simulator, there is a common basic execution structure which consist of a loop. In this loop, the simulator repeatedly extracts an event with the smallest timestamp from the event list, updates the clock variable to this event timestamp and then processes the event. The event processing can change the model state and give origin to future events that may be inserted into the event list. This loop continuously executes until an ending condition is met.

Traditional discrete-event simulators are characterized by presenting a cooperative multitasking (also referred as non-preemptive multitasking), meaning that a process/thread can self-interrupt and voluntarily give control to other process/thread. This approach highly simplifies the communication through events and the access to variables in shared memory. On the other hand, the available parallelism in recent multi-core CPUs is wasted [26], as well as the possibility to perform simulation through different computers spatially separated.

As systems complexity grows, issues related with simulation speed and memory usage arise. In order to face these challenges, parallel and distributed DES became an interesting and relevant research area combining modelling and simulation concepts with high-performance computing [27]. Several surveys, reviews or books about this topic were published by different authors like Misra [28], Fujimoto [25], Liu [27] or Ferscha [3].

Although the terms *parallel DES* and *distributed DES* are sometimes used indiscriminately, some authors make a distinction between both concepts. According to Ferscha [3], parallel DES refers to a SIMD environment in which a set of processors execute similar operations (fetched from a central control unit) on different data. This central control unit ensures the synchronization between independent computations. On the other hand, distributed DES is characterized by MIMD environment. In this case, the system event structure is decomposed and different processes operate "asynchronously in parallel". Here, communication between processes is required with the intention of exchange data and correctly synchronize each process.

Fujimoto [29] makes a slightly different distinction: a parallel DES consists of the execution of a single simulation program through a group of tightly coupled processors, as in a shared memory multiprocessor architecture; in a distributed DES there is an execution of a single simulation program on a set of loosely coupled processors, as in a computer network.

Fujimoto also refers an alternative approach which targets the execution of several, independent simulations concurrently on different processors - *replicated trials*. Although it is a simple approach, it has the disadvantage of not providing any kind of speed-up and of requiring each processor to have enough memory space for simulation. It is also an approach which is not adequate for interactive environments.

Usually, in a parallel or distributed DES the events are distributed among a collection of communicating *logical processes* (LPs) in order to divide a global simulation task and exploit parallelism through the concurrent execution of these processes. Ferscha [3] classifies these simulation strategies as *logical process simulation* (LP simulation) and presents a basic architecture of it (Figure 3.3).

LP simulation is easily understandable if one views a simulation as a set of events characterized by a temporal coordinate (*timestamp*) and a spatial coordinate (the location of the state variables affected by the event). This *space-time* view of simulation was presented in 1989 by Chandy and Sherman [30]. The space-time spectre is then divided into different *regions* which are assigned to different LPs. Each LP keeps and maintains its own local clock and event list, and executes *local events* through its *Simulation Engine*. The execution of local events can affect other LPs, generating *remote events*. These remote events also need to be processed by the affected LPs. So, an LP needs to exchange their local data in order to perform event notification regarding to other LPs and also to let them know about its local simulation time. The *Communication System* makes

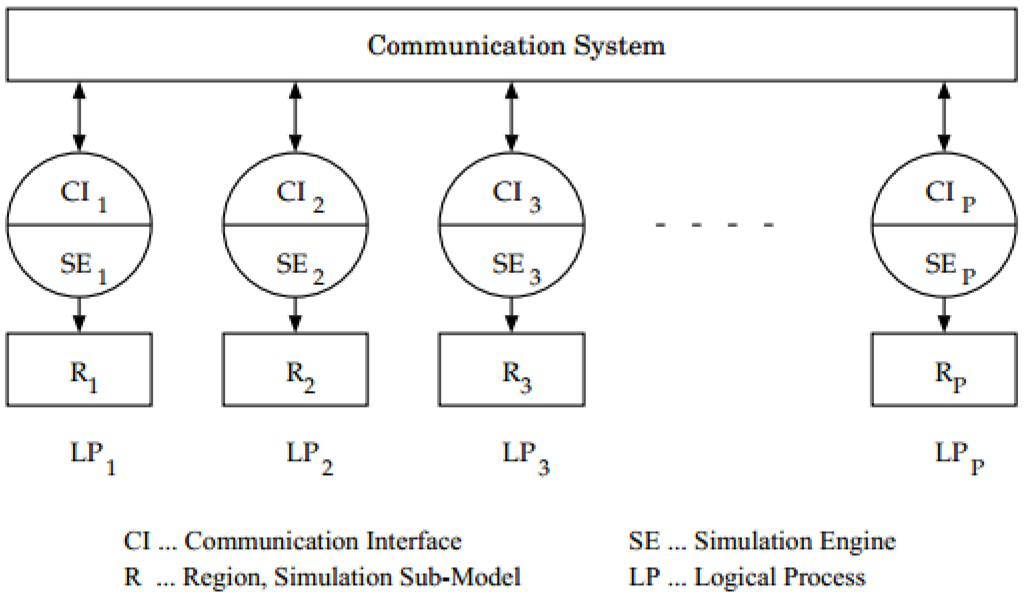


Figure 3.3: Architecture of an LP simulation [3]

this data sharing between LPs possible. An LP can access the Communication System thanks to the *Communication Interface* attached to the Simulation Engine.

In order to guarantee the logic coherency and consistency of a DES, events have to be processed in a non-decreasing timestamp order, because an event with a smaller timestamp can potentially affect the system state and consequently also affect future events. This DES requirement is called *causality constraint* and involves a total ordering of events.

In LP simulation, the global event list is spread among the LPs local event lists. Although the Simulation Engine of each LP can ensure the causality constraint for all local events - *local causality constraint* - it cannot predict the arrival of remote events from other LPs or the timestamp of these remote events. This way, it can happen that, while an LP is processing a local event, a remote event with a smaller timestamp arrives. So, this remote event with a smaller timestamp will be processed out of timestamp order, breaking the causality constraint for that LP. In other words, not processing events on timestamp order can cause "the computation for one event to affect another event in its past" [29]. These failures are called *causality errors* and the problem caused by these errors is the *synchronization problem*. Liu [27] shortly describes the challenge arising from the synchronization problem as "the difficulty of preserving the local causality constraint at each LP without the use of a global simulation clock". In turn, Fujimoto [25] points to the complexity and high data dependency of the constraints that command the order in which operations have to be performed relative to each other.

To address the *synchronization problem*, a collection of algorithms has been developed - *synchronization algorithms*. A review of some of these algorithms will be present in the next section.

3.3 Synchronization in Discrete-Event simulation

Synchronization in parallel/distributed DES can be achieved through two algorithm categories: *conservative* and *optimistic*. Conservative synchronization algorithms do not allow the occurrence of any causality error by blocking an LP from processing its next event until it is certain that this event will not generate out-of-order event processing due to later events from other LPs. On the other hand, optimistic synchronization algorithms allow causality errors to occur. However, this class of algorithms has the ability to detect causality errors and recover from them, by invoking a *rollback* mechanism. Rollback requires state saving and recovery mechanisms.

3.3.1 Conservative Synchronization

Chandy and Misra [31], as well as Bryant [32] independently presented the first parallel/distributed DES algorithm which is commonly known as the *CMB algorithm*. In this algorithm, LPs are connected via directional links used to exchange events between LPs in a non-decreasing timestamp order. For each incoming link at an LP there is an input queue used to place the received events. A clock variable is attached to each input queue and its value is the timestamp of first event in the queue, if any. If the queue is empty the clock variable assumes the value of the last processed event, which is initially zero. An LP iteration first selects the input queue with the smallest clock value and then processes its first event. If the queue is empty, the LP blocks until the arrival of an event in that queue. Once it arrives, the LP execution continues to the next iteration.

Given the nature of this algorithm, the occurrence of a blocking cycle among all LPs is possible, driving to a *deadlock* situation. To avoid these situations, CMB algorithm uses *null messages* which do not represent any event but only transport a timestamp. This timestamp can be seen as a guarantee from the sending LP that it will not send futures events with a timestamp smaller than the null message timestamp. The LP's input queue which receives a null message can advance its clock variable and notify other LPs about this time advancement (also using null messages). So, the *null message protocol* is a *deadlock avoidance* mechanism. There are also some protocols which allow deadlock to occur, being this occurrences detected and recovered [33].

The null message synchronization remarks to a fundamental property common to all conservative synchronization algorithms - *lookahead*. It is related with the capacity of determining if the next event in a simulator is safe to process. According to Liu [27], lookahead is the "amount of simulation time that an LP can predict into the simulated future". Nicol [34] presents another definition of lookahead: considering two processes, p and q, process p has lookahead regarding to process q if, being the simulation clock of p at time s, process p can determine that there is no way it will insert/delete an event in/from process q event list with timestamp $t > s$. Apart from this definition, Nicol also presents some subtleties of lookahead.

There are other conservative synchronization protocols beyond the null message protocol. In [35] and [36], Nicol and Reynolds present the *appointment protocol*. It is an adaptation of the null message protocol, but replaces the null messages for *appointments* which consist of a promise by the sending LP to not send any message with a timestamp larger than the appointment time. Although it seems to be the same mechanism as the one from the null message protocol, there is one main difference. A null message is enqueued and is only processed when the LP's clock reaches the null message timestamp. In contrast, the appointment information can be immediately used by the receiving LP. Thus, at any time, an LP can access the appointment information in order to calculate lookahead and eventually notify other LPs.

Based on the appointment algorithm and taking into account the LP topology, the *time-of-next-event algorithm* was presented by Groselj and Tropper [37]. It computes the lower bound on the time of the next message sent from one LP to the other - *link time*. To do so, the algorithm considers the time of next event in each LP, existing link times, the minimum delay an LP takes to process a message and the shortest path algorithm. The link times can be viewed as appointments.

Until now, the synchronization algorithms presented are *asynchronous* in the sense that each LP locally computes its lookahead, based on its data and data received from other LPs. However, there are some algorithms in which there is a global lookahead calculation across all processors in order to identify, for each LP, a time instant up to which it is safe to advance. The *bounded lag algorithm* [38] is an example of this class of algorithms. Assuming a static LP topology, a time interval B - *lag* - is used to compute a *sphere of influence*: a group of LPs that might affect an LP within B simulation time units. The LPs belonging to the sphere of influence need to be considered to determine if a certain LP with timestamp between the current simulation time T and T+B can be processed in a safe way.

Another example of a synchronous algorithm is the *conditional event approach* [39] presented by Chandy and Sherman. In a sequential simulation, all the events on an event list, except the one with the smaller timestamp, are *conditional*. It means that the only event which cannot be preempted, removed or changed - *unconditional* - is the earliest event and the simulator can safely advance to its timestamp. In a parallel/distributed simulation, the event list from an LP can have conditional events, unconditional events and events which do not affect other events in this local event list. However, the event with the smallest timestamp in one LP is not necessarily unconditional because it can be affected by events with a smaller timestamp from other LPs.

Through the conditional event algorithm, each LP determines its conditional event with the smallest timestamp and delivers it to a global reduction operation which returns the timestamp of the earliest conditional event in the whole system. This returned timestamp defines the simulation time up to which all LPs can concurrently simulate.

3.3.2 Optimistic Synchronization

The most widely known optimistic synchronization approach is the *Time Warp* [40] [41]. In this approach, each LP stores events received from other LPs in an input queue, while the events sent to other LPs are stored in an output queue. Before an event is processed, an LP saves the state variables in a state queue. When an event with a timestamp smaller than the current simulation clock arrives - *strangler event* - the LP must *rollback* to the saved state immediately before the strangler event timestamp. In addition, it is required that all actions (with a timestamp bigger than the strangler event timestamp) which affected other LPs are cancelled. This is accomplished through the use of *anti-messages*. These messages correspond to the original messages saved in the output queue. An LP which receives an anti-message should remove the corresponding message from its input queue. It can happen that the timestamp of the received anti-messasge is smaller that the LP's current simulation clock. This means that the anti-message corresponds to an event already processed, and thus it is also a strangler event. Under these circumstances, the LP which received the anti-message must perform *rollback* as well.

A problem can arise from this approach: rollback cannot be applied to irrevocable operations (such as I/O). So, the algorithm must define when these operations can be performed. These operations must be executed at a point of time T such that the system will never rollback to a time earlier than T. This point of time T is known as the *Global Virtual Time* (GVT) and is defined as the minimum timestamp among all events and messages (including anti-messages) in the system at a certain point of time. The computation of GVT is a primary source of overhead in the Time Warp algorithm [27].

3.3.3 Conservative vs. Optimistic Synchronization

Conservative algorithms are usually easier to implement than the optimistic ones. That happens because the state saving and recovery mechanisms required by optimistic algorithms are code intensive tasks. Additionally, these mechanisms ask for more memory and processor resources. On the other hand, optimistic algorithms have the advantage of allowing the simulator to exploit parallelism in a better way, by permitting situations where causality errors are possible but actually do not occur. Qualitative comparison of conservative and optimistic algorithms is a sensitive point. The performance of a given model relies on the strategy followed in the implementation, as well as on the platform targeted. Thereon, Ferscha states that "General rules of superiority cannot be formulated, since performance [...] cannot be sufficiently characterized by models, although exceptions do exist." [3].

3.4 Interprocess Communication mechanisms

In parallel/distributes DES, the labour associated with the global simulation is divided among different processes which need to communicate and exchange data between them. The methods or mechanisms which allow such operations are named *Interprocess Communication (IPC) mechanisms*.

Usually, processes or applications which use IPC mechanisms are categorized as *clients* or *servers*. A client is viewed as a process or application that requests a service from other process or application, while a server is a process or application which replies to a client request. It is also possible that a process/application acts as client and server.

There is a considerable variety of IPC methods and in this section just few of them will be briefly referred.

A common way to perform exchange data between two processes is the use of *shared memory*. It consists of a portion of memory attached to some well-known address. Knowing this address, two or more processes can access the shared memory and perform read or write operations. In order to maintain data integrity among all processes, some memory access control mechanism has to be employed to avoid *race conditions* - accesses to a shared memory space by two or more processes at the same time.

Sockets are other IPC mechanism which allow information exchange between processes on the same machine or in different machines across a network. A socket is basically a connection endpoint with a given name and address. A socket is also characterized by the transport protocol used in communication (such as TCP/IP or UDP). The communication via sockets can be unidirectional or bidirectional. Several implementation of sockets have been presented, being the *Berkeley sockets* one of the most popular ones.

Consisting of FIFO communication channels with two endpoints, *pipes* are another IPC approach. Pipes can be classified in two categories: *anonymous pipes* or *named pipes*. Anonymous pipes are unnamed and one-way pipes typically used for data exchange between a parent process and a child process. They can only be used by processes in the same machine. In contrast, named pipes are named, one-way or duplex pipes usually used in a client/server configuration. A named pipe server can be accessed by multiple instances of client pipes. This category of pipes is suitable for communication between processes running in the same machine or in different machines.

Message passing is an IPC mechanism in which messages are transferred between processes. The messages content can vary from complex data structures to small quantities of bytes. One of the most broadly used *message passing* implementations is the *MPI* which is a language-independent protocol that provides communication functionalities between a group of processes previously mapped to nodes or computer instances.

3.5 Related Work

The advent of SLDLs, the demand for high simulation performance and the need of running simulators spatially separated drove to the development of several attempts to integrate techniques of parallel and distributed DES into hardware/software projects.

Although they belong to the domain of continuous/discrete systems co-simulation, [42] and [43] contain some interesting heterogeneous systems design concepts, by presenting frameworks which allow automated co-simulation between *SystemC* and *Simulink* components.

Bombana and Bruschi [44] focus on *SystemC/VHDL* co-simulation in order to mix different abstraction levels. This approach combines co-simulation with synthesis through a design flow that allows the automatic generation of VHDL test benches from SystemC code. In [45], *SystemC* simulation in multiprocessor systems is targeted. The authors propose some approaches for co-simulation between GDB (GNU Debugger)-based ISSs and SystemC modules running in different processes.

Heterogeneous models are targeted by Dubois and Aboulhamid [46], who show approaches for simulators interconnection based on interoperability and managed code. In this work, different communication mechanisms (such as shared memory, TCP/IP, COM) are explored having in mind the co-simulation between different tightly synchronized simulators. The presented case studies cover *ESys.NET/SystemC* and *Simulink/ESys.NET* co-simulation. Simulation speed measurements are performed, being COM the fastest approach and shared memory the slowest one.

An alternative procedure for heterogeneous systems co-simulation is proposed in [9] and [10]. Instead of keeping the different system components in their native languages, all models in the system are translated to common and unique format/language. In fact, the original heterogeneous system is transformed in a homogeneous system. This approach comprises three major phases: "AST generation", "Dependency Analysis and Scheduling" and "Code Generation".

Trams [47] developed a plug-in library intended for distributed simulation, to be used with *SystemC*. The system setup and partitioning is done manually through library calls. As Trams had no intention to modify *SystemC* kernel, a conservative synchronization approach is chosen and *explicit lookahead* is implemented. This way, the lookahead values are defined and known at the latest in the beginning of the simulation. Therefore, this approach only targets a class of systems with a regular behaviour whose signals vary in a predictable way. The data transmission between the distributed modules is done using TCP/IP.

Exploring parallel DES techniques, Cox [48] developed a distributed version of the *SystemC* simulation environment. This approach tries to minimize user interaction and performs modifications on *SystemC* kernel. The distributed *SystemC* models communicate via MPI. As opposed to the previously work developed by Trams, Cox has the intention to cover a wide variety of systems and rejects explicit lookahead. The synchronization algorithm used is an adaptation of the *conditional event approach*.

Chopard *et al.* [49] also propose a parallelization of *SystemC* kernel using a conservative algorithm. In this implementation, a new construct is added to *SystemC* - *sc_node_module*. It is actually a new kinf of *sc_module* which comprises all *sc_modules* belonging to a given node of the entire system. A *sc_node_module* cannot include other *sc_node_module*s and it can be viewed as an entity similar to an LP. All local schedulers synchronize with each other at the end of every *delta-cycle* and two kinds of synchronization are considered: *channel synchronization* and *time synchronization*. The later one is performed by a master node which gathers the timestamp of the earliest event in every node in the system, computes the next simulation time and informs all nodes about it. As a continuation of this work, Combes *et al.* [50] seek for some strategies to face the high level of synchronization which can affect the overall performance.

Recently, Roth *et al.* [51] presented a distributed *SystemC* environment based on the HLA international standard [52]. HLA was originally developed by the Defense Modeling and Simulation Office for the U.S. Department of Defense. It was intended for military training simulations and consists of a software architecture which combines all the components needed for parallel DES.

This chapter provided fundamental knowledge which constituted a useful and important basis to understand the following chapters. Particularly, the informations contained in this chapter can be considered the cornerstones for the implemented work presented in the next two chapters.

Chapter 4

SystemC code generation from *System#* projects

This chapter draws the attention to one of the two main goals of this Master dissertation: implementation of *SystemC* code from *System#* projects. This way, *System#* will be viewed from a *high-level synthesis* (HLS) tool perspective. It is intended to use the system representation produced by *System#* compilation in order to generate the corresponding *SystemC* code.

4.1 *System#* as a High-Level Synthesis Tool

According to McFarland *et al.* [53] high-level synthesis consists of moving from "an algorithmic level specification of the behaviour of a digital system to a register-transfer level structure that implements that behaviour". The authors also clarify the meanings of *behaviour* (how is the interaction between the system/its components and the surrounding environment?) and *structure* (what are the system's components and how are they interconnected?)

Although section 3.1.2 describes *System#* as a SLDL, it is also a HLS framework which has the ability to convert an elaborated model to synthesizable *VHDL* code. However, the *VHDL* code generation itself is merely the final step which makes use of a system representation built during the *System#* design flow. The design flow is illustrated in Figure 4.1.

After *System#* compilation of a design system, a CIL representation of the whole system (including code relative to processes and methods) is produced. The CIL code is the input of the *System#* design flow. Through model reflection, *System#* framework is able to perform structural analysis on an elaborated system model. In practice, each element presented in the system model is reflected by a suitable descriptor object belonging to the *SystemSharp.Meta* namespace. Examples of these object descriptors are shown in Table 4.1. All the descriptor classes and interfaces derive directly or indirectly from *DescriptorBase* class, and are defined in the *SystemSharp.Meta* namespace.

The structural analysis also infers the relationships between model elements and uses the object descriptors previously referred to make these relationships perceptible.

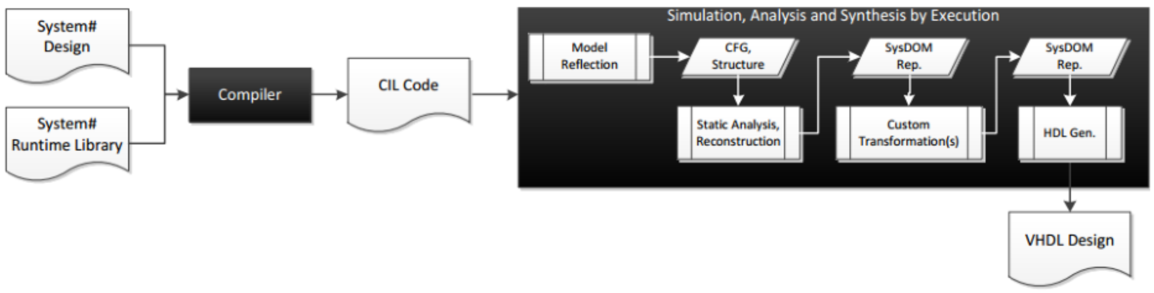


Figure 4.1: *System#* design flow [4]

Apart from the structural analysis, the CIL code is also analysed in order to identify instructions and recover a control flow graph which characterizes the system/component operation. Several classes from the *SystemSharp.SysDOM* namespace represent statements and control flow structures common in a wide range of programming languages. Table 4.2 presents some of those classes. The classes names are self-explanatory.

In turn, the existing loops and conditional statements are inspected and an AST is constructed. This AST follows the *SysDOM*, a document object model for code which is limited to a small group of imperative language elements, but also includes system and hardware modelling domain elements. At the end of *System#* design flow, a complete representation of the system structure and behaviour is produced. This system representation (AST following *SysDOM* model) is, in fact, the starting point for the *VHDL* code generation.

So, the code generation task shortly consists of extracting from the system representation informations about the system (for example: name of a component, list of its sub-components, list of input/output ports, etc.) and using those informations to write the corresponding code in a target file, according to a given language syntax (*VHDL*, *SystemC*, *Verilog*, etc.)

4.2 *SystemC* code generation

The implementation of *SystemC* code generation is actually an adaptation of the *VHDL* code generation. As stated in the previous section, the code generation is just a final step which uses the final system representation produced during *System#* design flow. More precisely, it consists in the AST unparsing. The system representation is independent of the desired generated code language.

The first step towards *SystemC* code generation was the inspection of the *VHDL* code generation engine. This included the run of the example projects included on the *System#* version available in [24], the study of the *VHDL* generated code for these projects and also the study and understanding of the *System#* core files present in the *Systemsharp/Synthesis/VHDLGen* folder.

Then, a manual code pattern analysis was made through observation and comparison between *System#* project code and *VHDL* generated code. Moreover, code generation functions (from

Descriptor class	**Entity represented**
ComponentDescriptor	Component/Module
PortDescriptor	Port
SignalDescriptor	Channel
FieldDescriptor	Variable/Constant/Field
ProcessDescriptor	Process
MethodDescriptor	Method

Table 4.1: Examples of *System#* element descriptor classes and entities represented

System# core code files present in *Systemsharp/Synthesis/VHDLGen* folder) were inspected in order to map them to the respective generated code section. The next step consisted in studying the typical *SystemC* code structure and identifying structural similarities and differences between *SystemC* code sections and their *System#/VHDL* counterparts. Concurrently, particular *SystemC* code features were pointed out.

An initial map between common *SystemC* code sections and code generation functions was created. Some of the code generation functions were adapted from the ones present in *VHDL* code generation and others were created to cover some specific *SystemC* features (for example, the concept of constructor exists in *SystemC*, but not in *VHDL*). The code generation implementation itself started with the creation of the *SystemSharp/Synthesis/SystemCGen* folder and its files. The folder and its contents were mirrored from the *SystemSharp/Synthesis/VHDLGen* folder. Two files were created:

- **SystemCGen.cs:** contains the functions responsible for generating *SystemC* syntax related with module *header* files (whose structure is illustrated in Figure 4.2) and the *main.cpp* file;
- **SystemCTypes.cs:** contains functions responsible for generating *SystemC* syntax related with data type values and conversions;

These files were then edited and developed in order to implement the code required to achieve the *SystemC* code generation from *System#* projects.

4.2.1 *SystemC* modules structure and generation

As referred in section 3.1.1, a module is the basic structural block in *SystemC*. It is used to implement the concept of *Hierarchy*, which is tremendously important to deal with large designs in *system-level design*. A module allows the designer to work on a piece of design separately. Usually, one defines a *SystemC* module in a *C++* header file. An example of a typical structure of a header file which implements a *SystemC* module is shown in Figure 4.2.

Next, and following the *SystemC* pseudo-code from the previous figure, an overview of the main aspects, classes and functions involved in *SystemC* modules code generation is presented. The majority of *System#* methods involved in the generation of a *SystemC* module code belongs to the *SystemCGenerator* class defined in the *SystemSharp/Synthesis/SystemCGen/SystemCGen.cs* file.

Class
IfStatement
CaseStatement
BreakLoopStatement
GotoStatement
NopStatement
StoreStatemet
LoopBlock

Table 4.2: Examples of *System#* classes representing statements and control flow structures

The *C++* header file intended for a *SystemC* module implementation is created by the method:

void GenerateComponent(IProject project, ComponentDescriptor cd)

This method adds the new created file to the current working project - *IProject project* - and writes *SystemC* code on it. This code corresponds to the current working module which is represented by *cd*. To perform Code generation, this method calls other methods from the *SystemC-Generator* class.

Pre-Processing directives and the *Dependencies section* are generated by:

void GeneratePreProcDir(ComponentDescriptor cd, IndentedTextWriter tw)

and

void GenerateDependencies(ComponentDescriptor desc, IndentedTextWriter tw, string)

, respectively. This later method inspects code dependencies in order to figure out which *C++/SystemC* libraries are required to successfully compile the final code. Actually, before the call to *GenerateComponent()*, *System#* performs reflection over the targeted system and creates a collection of libraries needed, due to the use of certain data types or functions. So, once *GenerateDependencies()* is called, a list of dependencies already exists. This way, this method has to go over the pre-existent list of libraries and generate the corresponding *#include* comands. Obviously, the header files related to *SystemC* sub-modules which are a constituent part of the current working module must be also considered as dependencies. The *ComponentDescriptor* object representing the current working module provides a list of *ComponentDescriptor*s, which corresponds to its sub-modules.

The code section starting by the macro *SC_MODULE* is generated within:

void DeclareAndGenerateModule(ComponentDescriptor cd, IndentedTextWriter tw)

The method argument *cd* plays a crucial role here, once it comprises information about all structural and functional constructs of a module, as well as other information like the component name. All those informations are reachable through a set of methods implemented by *Component-Descriptor*.

```
// Pre-Processing directives

// Dependencies section
#include "dependency_1_name"
#include "dependency_2_name"
//...

SC_MODULE(module_name) {

       Port Declaration;
       Sub-Modules Instantiation;
       Local Channels declaration;
       Variables and Constants declaration;
       Processes Declaration/Implementation;
       Other Methods declaration/implementation;

       // Constructor
       SC_CTOR(module_name): instance_nameA("A"), instance_nameB("B") ...
       {
              Ports initialization;
              Processes Registration and Sensitive list;
              Module Variables/Constants Initialization;
              Local Channel Initialization;
              Port Binding;
       }
};
```

Figure 4.2: Structure of a *SystemC* module header file

Several methods are sequentially called in *DeclareAndGenerateModule()* body, in order to generate sections of the pseudo-code from Figure 4.2 (for example, *Sub-Modules Instantiation*, *Processes Declaration/Implementation*, *Constructor*, etc).

The declaration of a module's ports is in charge of the following method:

void DeclarePortList(ComponentDescriptor cd, IndentedTextWriter tw, bool extScope)

In turn, this method calls the method *GetPorts()* (supported by *ComponentDescriptor* objects) which returns an iterable set of *PortDescriptor* objects - one for each module port. A *PortDescriptor* object encapsulates informations like port name, direction and data type exchanged on the port.

Then, one proceeds to the instantiation of sub-modules contained by the current working module. To know which sub-modules must be instantiated, another method implemented in *ComponentDescriptor* is called - *GetChildComponents()* (returns an iterable set of *ComponentDescrip-*

tors corresponding to the sub-modules which constitute the current working module). Each sub-module is then instantiated in the current working header file.

Similarly to *GetPorts()*, there are other methods which return information about local channels, module variables and constants: *GetSignals()* (returns a set of *SignalDescriptor* objects), *GetVariables()* and *GetConstants()* (both return a set of *FieldDescriptor* objects), respectively.

Afterwards, code related with the module's behaviour is generated. A set of *ProcessDescriptor* objects, representing the processes executed by the current working module, is provided by *GetProcesses()*. For each *ProcessDescriptor* returned, a call to

void DeclareProcess(ProcessDescriptor pd, IndentedTextWriter tw)

is performed. In this method, code relative to a given process is generated including process header, local variables declaration and process body. The method *GenerateMethodBody()* performs the generation of method bodies in both processes and local module methods. The information required to generate code for a given process is provided by a *ProcessDescriptor* object.

An analogous procedure is executed to generate the code correspondent to other methods present in the module. Here, the set of *MethodDescriptor* objects is returned by the *GetActiveMethods()*.

SC_MODULE is a macro for the base *C++* class *sc_module*. Following the logic of *object-oriented programming*, objects usually need to initialize variables or define dynamic memory issues at the time they are created. In *SystemC*, a *SC_MODULE* constructor is responsible for significant actions like process registration and port binding. A common way to declare the constructor for a *SystemC* module is using the macro *SC_CTOR*. The generation of a module's constructor code is carried out by the method:

void GenerateCtor(ComponentDescriptor cd, IndentedTextWriter tw)

After the code segment *SC_CTOR(module_name)*, an initializer list is generated if there are sub-modules or arrays of ports/channels in the current working module. The survey of these elements is done through the same methods previously used to declare ports/channels and instantiate sub-modules (*GetPorts()*, *GetSignals()* and *GetChildComponents()*). Next, output ports having an initial value defined are initialized. Once one gets the *PortDescriptor* object referring to a port intended for initialization, it is possible to get its initial value through the *PortDescriptor C#* property *InitialValue*.

Regarding to processes, declaring and implementing them is not enough in *SystemC*. Apart from the functional meaning, processes distinguish themselves from other methods within a module by being registered in the simulation kernel. This registration allows a function/method to be recognized as a *SystemC* process and is performed in a module constructor. To generate the *SystemC* code which executes this action, one uses the method:

InitializeProcess(ProcessDescriptor pd, IndentedTextWriter tw)

Within this method, the *ProcessDescriptor* passed as an argument is analized in order to know its kind. This information is stored in the *ProcessDescriptor C#* property *Kind*. Evaluating it, it is possible to generate the correct process macro (*SystemC* process macros were referred in section 3.1.1). *InitializeProcess()* not only generates the code for process registration, but also generates the static sensitivity list for *SC_METHOD*s and *SC_THREAD*s. The *C#* property *Sensitivity* from *ProcessDescriptor* provides a set of signals or ports to which a given process is sensitive. Using again *GetConstants()* and *GetVariables()* methods to gather the existing constants and variables, one proceeds to the initialization of those elements. For each *FieldDescriptor* element gathered, a call to

InitializeField(FieldDescriptor fd, IndentedTextWriter tw)

is made. In this method, the *FieldDescriptor C#* property *ConstantValue* is used to produce the syntax for the initialization of a given constant/variable.

In a similar way, Local Channels initialization code is generated through:

InitializeSignal(SignalDescriptor sd, IndentedTextWriter tw)

Here, *C#* property *InitialValue* (from *SignalDescriptor* class) allows the generation of the signal initialization.

The port binding code generation is performed by the method

PortBinding(ComponentDescriptor cd, IndentedTextWriter tw)

, which is called for each sub-module contained by the current working module. This method gets a set of ports of each sub-module and for each port (*PortDescriptor* object) looks for the signal bound to that port, using the *C#* property *BoundSignal* (defined in *PortDescriptor* class). Having this information, the task of generating the correct *SystemC* syntax for port binding operation is simplified.

In order to concretize what was previously exposed, an example of *SystemC* code generated for a counter module is presented in Figure 4.3. The original *System#* class representing the same counter is shown in Figure 4.4.

4.2.2 *main.cpp* file generation

The only entry point to a *SystemC* application is the function:

int sc_main(int argc, char argv[])*

This function is not declared inside any module and has no representation within the *System#* internal representation of a given system. One of the most important tasks of the *sc_main()* function is triggering *elaboration* and *simulation*. Actually, elaboration starts once *sc_main()* starts its execution and finishes at the point immediately before the first call to the *sc_start()* function.

So, the instantiation of the module(s) intended for simulation must be done within the elaboration phase. In turn, *sc_start()* triggers the simulation phase. As a HLS tool, *System#* hierarchizes a system in a way such that the whole system is encapsulated in a *top module*. This top module includes, directly or indirectly, instances for all system elements. This way, during the elaboration phase in *sc_main()*, it is enough to instantiate only the top module. For this reason, the *C++* source file which includes the *sc_main()* function - *main.cpp* - is generated immediately after the generation of the top module header file generation. It means that *main.cpp* is generated in the end of *GenerateComponent()* function if, and only if, the current working module is the top module. Once this condition is met, a call to:

void GenerateMainFile(IProject project, ComponentDescriptor cd

is made. The *ComponentDescriptor* passed as an argument refers to the top module. This method creates a *C++* source file named *main.cpp* and adds it to the current working project. Then, it generates code dependencies (in this case, the most relevant dependency is the *#include* referring to the top module header file) and the *sc_main()* code. Basically, the *sc_main()* function comprises and instantiation of the top module and a call to *sc_start()*. Figure 4.5 illustrates the aspect of the *main.cpp* file generated. The *System# Main* function in which code generation is triggered is presented in Figure 4.6.

4.2.3 Data types operations

An important issue during *SystemC* code generation from *System#* projects had to do with data types conversion between *System#* and *SystemC*, as well as the operations supported by each data type. When studying both SLDLs, a comparison between data types from both domains was made. Not all *System#* data types have a *SystemC* counterpart which fully represents the same type of data and provides the same data operations. Taking these factors into account, a correspondence table was built, seeking for the most accurate symmetry between *System#* and *SystemC* data types. Table 4.3 exposes the data type correspondence settled.

Issues with respect to data type operations were notorious when dealing with logic data types - *StdLogic/sc_logic* and *StdLogicVector/sc_lv*. Internally, *System#* represents a *Bitwise Not* (operator '∼') operation of a logic value as a *BoolNot* operation (operator '!'). So, the following situation can be generated:

sc_logic x = '0', y;
...
y = !x;

This will originate a syntax error, once the *sc_logic* data type does not support the '!' operator. To overtake this situation, the operator '!' was overloaded to be used with *sc_logic* values. In fact, the '!' operator was overloaded to return the same result as the *sc_logic Bitwise Not* operator would return. So, in the code example previous shown, the instruction:

System#	**SystemC**
bool	bool
sbyte	sc_int<8>
byte	sc_uint<8>
short	short
ushort	unsigned short int
int	int
uint	unsigned int
long	sc_int<64>
ulong	sc_uint<64>
float	float
double	double
char	char
string	string
StdLogic	sc_logic
StdLogicVector	sc_lv
Signed	sc_bigint
Unsigned	sc_biguint
SFix	sc_fixed
UFix	sc_ufixed
Time	sc_time

Table 4.3: Correspondence table between *System#* and *SystemC* data types

y = !x;, will actually perform $y = \sim x$;

Apart from the '!' operator overloading, the concatenation function (*concat()*) was also overloaded in order to be able to perform concatenation between two *sc_logic* values, resulting in a *sc_lv<2>* value:

sc_lv<2> concat(sc_logic a, sc_logic b)

These two operations were declared and implemented in a header file whose name is *sc_logic_add_ons.h* and a file using its functions must include it.

In *System#* arithmetic operations over arrays of logic values (*StdLogicVector*) are allowed, while *SystemC* does not support this kind of operations. To overcome this problem, the arithmetic operators (+, -, *, /) were overloaded in order to be used with *sc_lv* variables.

System# also allows the conversion from a *StdLogicVector* value to a fixed-point value (*SFix* or *UFix*), using a single instruction. On the other hand, in *SystemC* these procedure requires more than one instruction. Once generating *SystemC* code, it is practical to have a *1-to-1* direct mapping between instructions. Therefore, two functions were implemented aiming the conversion from *sc_lv* to *sc_fixed* and *sc_ufixed* data types:

sc_fixed<W, IW> lv_to_fixed(sc_lv<W> value, sc_fixed<W, IW> arg)

sc_ufixed<W, IW> lv_to_ufixed(sc_lv<W> value, sc_ufixed<W, IW> arg)

The arithmetic operators for *sc_lv* and the function which perform conversion from logic vectors to fixed-point values are defined and implemented in the file *sc_lv_add_ons.h*.

4.2.4 Implementation details

During the code generation implementation, some issues arose because the system representation internally produced by *System#* is target language-independent and some constructs or elements have no direct mapping between the system representation and the *SystemC* syntax. Due to this, the generated code produced is not exactly the same code a system designer would ideally write. However, the main concern was to correctly translate the semantics of a *System#* project to *SystemC* code. Some implementation options taken during this process will be described next.

In the *System#* internal representation, two instances of a given component are viewed as instances of two different components which have the same structure and behaviour but different names. For instance, considering a bus arbitration system with ten bus masters, instead of generating one header file containing the structure and behaviour description of a system module representing a bus master which can then be instantiated ten times, the code generation engine will produce ten header files, each one describing a bus master.

The *System#* arrays of signals and ports are translated to *SystemC* using the *sc_vector* construct, which is a feature introduced in the last *SystemC* released version - IEEE 1666-2011. The use of this construct has the advantage of turning the *port binding* code section easier to implement.

Regarding to processes - the entities which describe the behaviour of a module - the internal system representation only considers the existence of two types of processes: Triggered processes (*EProcessKind.Triggered*) and Threaded processes (*EProcessKind.Threaded*). However, both *SystemC* and *System#* SLDLs allow a third kind of process: the Clocked Thread process. A Clocked Thread process from a given *System#* module is internally represented as a Triggered process which is sensitive to a clock edge. In practice, this means that a *System#* Clocked Thread process is translated to *SystemC* as a *SC_METHOD* - the *SystemC* keyword used to register a Triggered Process.

There are also some particularities related with functions/processes body code. When performing system behaviour analysis, *System#* internally represents the majority of control flow statements (such as *while-loops*, *do-while-loops* or *for-loops*) using *if-then-else* statements and unconditional jumps (*goto <label>*). The only control flow statements which is not represented this way are the *swith-case* statements, which are crucial constructs heavily used for *Synthesizable Sequential Descriptions* that is, the automatic transformation of cycle-accurate sequential behaviour into FSMs (a feature provided by *System#* intended for synthesizable FSMs specification in FPGA designs).

This way, the generated code often employs the *goto* control flow statement which is "considered harmful" by many computer scientists [54]. Once again, the main concern of code generation was the achievement of a correct code semantics rather than and ideal syntax.

In *C++* (*SystemC*), only integer, enumeration and *char* data types are allowed in *switch-case* statements. However, *System#* system behaviour analysis also considers the existence of *switch-case* statements which evaluate *string* values. To overcome this situation, before a *switch-case* statement is generated, the data type of the expression evaluated is inspected. If it isn't one of the *C++* mentioned before, the *switch-case* statement is converted in a *if-then-else* statement. This conversion is performed by the method

IfStatement ConvertToIfStatement()

, which belongs to the class *CaseStatement* from the *SystemSharp.SysDOM* namespace.

```
#ifndef M_M_CTR_H
#define M_M_CTR_H
#define SC_INCLUDE_FX
#include "systemc"
#include <iostream>
#include "sc_lv_add_ons.h"
#include "sc_logic_add_ons.h"
using namespace sc_core;
using namespace sc_dt;
using namespace std;

SC_MODULE(m_m_ctr)
{
	sc_in<sc_logic> Clk;
	sc_out<sc_lv<10>> Ctr;

	sc_signal<sc_lv<10>> m_ctr;

	void Processing()
	{
		if (Clk.posedge())
		{
			Ctr.write(m_ctr.read());
			m_ctr.write(m_ctr.read() + sc_lv<1>("1"));
		}
	}

	SC_CTOR(m_m_ctr)
	{
		Ctr.initialize(sc_lv<10>("UUUUUUUUUU"));

		SC_METHOD(Processing);
		sensitive << Clk;

		m_ctr.write(sc_lv<10>("1111111111"));

	}
};
#endif
```

Figure 4.3: *SystemC* generated code for a counter

```
class Counter : Component
{
     public In<StdLogic> Clk { private get; set; }
     public Out<StdLogicVector> Ctr { private get; set; }

     private SLVSignal _ctr;

     public Counter(int width)
     {
         _ctr = new SLVSignal(StdLogicVector._1s(width));
     }

     [TransformIntoFSM]
     private async void Processing()
     {
         await Tick;
         Ctr.Next = _ctr.Cur;
         _ctr.Next = _ctr.Cur + "1";
     }

     protected override void Initialize()
     {
         AddClockedThread(Processing, Clk.RisingEdge, Clk);
     }
}
```

Figure 4.4: *System#* code for a counter

```
#define SC_INCLUDE_FX
#include "systemc"
#include "top0.h"
#include <iostream>
#include "sc_lv_add_ons.h"
#include "sc_logic_add_ons.h"
using namespace sc_core;
using namespace sc_dt;
using namespace std;


int sc_main(int argc, char* argv[])
{
  sc_report_handler::set_actions (SC_WARNING, SC_DO_NOTHING);

  top0 top0("top0");

  sc_start(1000, SC_NS);

  return 0;
}
```

Figure 4.5: Generated *main.cpp* file

```
class Program
{
    static void Main(string[] args)
    {
        int cycles = 100;

        SimpleCounterTestbench tb = new SimpleCounterTestbench();
        DesignContext.Instance.Elaborate();

        DesignContext.Instance.Simulate(cycles *
                                 SimpleCounterTestbench.ClockPeriod);

        //Now convert the design to VHDL and
        //embed it into a Xilinx ISE project
        XilinxProject project = new XilinxProject(@".\hdl_output",
                                                 "SimpleCounter");
        project.PutProperty(EXilinxProjectProperties.DeviceFamily,
                                         EDeviceFamily.Spartan3);
        project.PutProperty(EXilinxProjectProperties.Device,
                                              EDevice.xc3s1500l);
        project.PutProperty(EXilinxProjectProperties.Package,
                                                 EPackage.fg676);
        project.PutProperty(EXilinxProjectProperties.SpeedGrade,
                                                 ESpeedGrade._4);
        project.PutProperty(EXilinxProjectProperties.PreferredLanguage,
                                                      EHDL.VHDL);

        VHDLGenerator codeGen = new VHDLGenerator();
        SynthesisEngine.Create(DesignContext.Instance,
                               project).Synthesize(codeGen);
        project.Save();

        //Now convert the design to SystemC
        XilinxProject project_SC = new XilinxProject(@".\SystemC_output ",
                                                     "SimpleCounter");

        SystemCGenerator codeGen_SC = new SystemCGenerator();
        SynthesisEngine.Create(DesignContext.Instance,
                         project_SC).Synthesize(codeGen_SC);
        project_SC.Save();
    }
}
```

Figure 4.6: *System# Main* function

Chapter 5

Co-Simulation and Interoperability between *SystemC* and *System#*

The main topic of this chapter is the implementation of a *SystemC*/*System#* co-simulation mechanism/environment, which is one of the main goals of this Master dissertation, besides *SystemC* code generation from *System#* projects.

Dubois and Aboulhamid [46] present a concept of *co-simulation*: "the ability to link different simulators in order to make them communicate and interoperate". In turn, *interoperability* is "the ability of two or more systems or components to exchange information and to use the information exchanged" [55].

Before diving into the implementation details, some co-simulation mechanism requirements are exposed. Then, special focus is given to the co-simulation mechanism implemented, including aspects related with communication and synchronization between the two considered SLDLs.

The whole co-simulation mechanism implementation was developed in a computer equipped with a Microsoft Windows operating system and the used IDE was Microsoft Visual Studio 2012.

5.1 Co-Simulation mechanism requirements

Prior to co-simulation mechanism implementation, some requirements were defined. First of all, the co-simulation of *SystemC* and *System#* models should produce correct and accurate results. As a term of comparison, the co-simulation results should be equal to the results one would obtain by performing a traditional simulation procedure, that is, a full system simulation using *SystemC* or *System#*. This characteristic can be denominated as *accuracy*.

It is desirable to be able to co-simulate previously written *SystemC* or *System#* models without any required modification to the models source code. Under these circumstances, the model designer can develop system models without being concerned about the way models will be simulated (single engine/language simulation or co-simulation of models written in different languages). This provides some *modularity* to the co-simulation mechanism and also some *compatibility* between previously written models and the co-simulation environment.

These requirements were the primary concerns during the co-simulation mechanism implementation. However, it is possible to point out other interesting characteristics for the mechanism to implement. Some of these characteristics are described next.

Ideally, from a user perspective, it is convenient that the execution of an heterogeneous system simulation (system modules described in different languages) would be similar to the execution of an homogeneous system simulation (system modules described in one language). This means that the system designer would have minimum interaction with the co-simulation mechanism - *transparency*.

Nowadays, one of the biggest embedded systems industry concerns is the reduction of design cycle duration, in order to meet *time-to-market* constraints. So, *speed* and *performance* are important and, as a relevant process during a system design, (co-)simulation should ideally be as fast as possible.

It is hard to reach a point where all of this characteristics meet each other. Moreover, some of these features can conflict. Knowing this, some trade-off's needed to be made during the co-simulation mechanism design and implementation.

With respect to speed and performance, it highly relies on the synchronization algorithm employed. However, while choosing an algorithm, one must have in mind the particular simulation kernel properties of the considered simulators. The use of a certain synchronization algorithm may require changes in a given simulation kernel and this can drive to conflicts with different versions of the corresponding SLDL. Among synchronization algorithms, the optimistic ones would require more simulation kernel changes in order to implement state saving and recovery algorithms.

Regarding to transparency, some interaction between the system designer and the co-simulation environment, like the definition of simulation parameters, is hard to avoid.

All the trade-off's made during implementation favoured the priority requirements first referred (accuracy and modularity/compatibility) rather the later referred co-simulation characteristics (transparency and performance).

5.2 Co-Simulation mechanism design and implementation

Considering the implementation driving forces exposed in the previous sections, a co-simulation mechanism approach was designed. The general architecture of the co-simulation environment is shown in Figure 5.1.

The analysis of the purposed design can be divided in two components: the physical topology of the whole system and the logic topology of each Simulation Domain (*SystemC*/*System#*).

The physical topology of the co-simulation environment is quite simple: two *Simulation Domains* (SD) are linked via a *Communication Channel*. Recalling Figure 3.3, it is possible to identify some similarities between the architecture of an LP simulation and the general architecture of the co-simulation environment. Each SD can be viewed as an LP with its own local clock and event

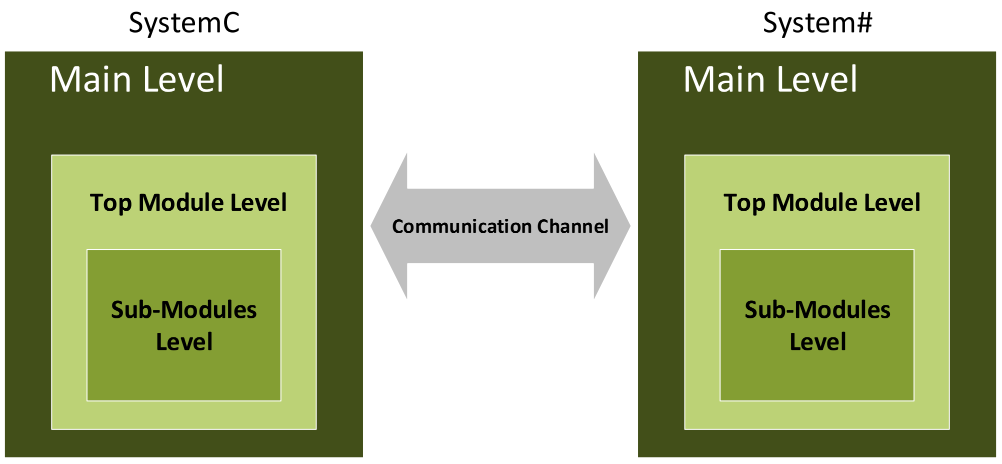


Figure 5.1: Co-Simulation mechanism general architecture

list, whose events are executed by its simulation engine (in this case, the *SystemC*/*System#* simulation kernel). The SDs (LPs) communicate through a Communication System - Communication Channel, in the purposed architecture.

Time synchronization information, as well as information related with the value of signals common to both SDs - *shared signals* - are transmitted through the Communication Channel. Subsection 5.2.1 exposes in detail the Communication Channel constitution and operation, including the IPC API which allows data exchange between SDs.

Each SD is characterized by a logic topology constituted by three hierarchical levels:

- **Sub-Modules Level:** consists of the *SystemC* modules/*System#* components one wants to (co-)simulate;
- **Top Module Level:** consists of a *SystemC* module/*System#* component - *Top Module* - which contains instances of modules/components from the *Sub-Modules Level*, instances of shared signals, as well as processes, methods and variables responsible for the update of shared signal values;
- **Main Level:** contains an instance of the *Top Module*, simulation parameters and the main co-simulation cycle. Communication operations between SDs occur in this level.

These levels are better described in subsection 5.2.2.

5.2.1 Communication between *SystemC* and *System#*

The problematic of communication between *SystemC* and *System#* consists, in fact, in achieving communication between two different processes: a discrete-event simulator written in *C++* and another written in *C#*. For this, an IPC mechanism available in both programming languages is required. The existence of an API for the chosen IPC mechanism would be convenient.

Among different IPC mechanisms available in *Microsoft Windows* operating system, *named pipes* were chosen. *Microsoft Windows* provides a named pipes API for *C++* (in *windows.h* header file) and *C#* (in *System.IO.Pipes* namespace).

An advantage of named pipes is their FIFO nature which ensures the order of received/sent data and this characteristic is useful in the co-simulation mechanism. Named pipes also allow different modes to write/read data: *message mode* (data is written to/read from the pipe as a stream of messages) or *byte mode* (data is written to/read from the pipe as a stream of bytes).

After choosing the IPC mechanism, the Communication Channel between both SDs was designed and implemented. The IPC operations introduce computation overhead to the co-simulation system and, as the operating system resources are limited, the number of used Named Pipes is also limited. Taking this into account, the implementation of a Communication Channel composed by one pipe for each shared signal in the co-simulation system doesn't seem to be an efficient and scalable solution.

The description of the communication between *SystemC* and *System#* implemented is divided in three parts: the Communication Channel topology, the structure/protocol of the transmitted data messages and the implemented API used to perform communication task between both SDs.

5.2.1.1 Communication Channel topology

Considering the factors previously referred, a physical topology for the Communication Channel was designed. It is composed by named pipes falling in two categories:

- **Signal data pipes:** pipes which transfer information relative to shared signals values;
- **Temporal data pipes:** pipes which transfer temporal information, more precisely, information about the time-of-next-event in the local SD.

The pipes categorization allows the separation of co-simulation information according to the information type. In turn, this allows a better data organization and helps in the data protocol simplification.

Three named pipes compose the Communication Channel:

- one signal data pipe for the *SystemC* $\longrightarrow$ *System#* data flow;
- one signal data pipe for the *System#* $\longrightarrow$ *SystemC* data flow;
- one temporal data pipe (bidirectional data flow).

Regarding to pipe write/read mode, all pipes in Communication Channel use the *message mode*. As the data transmitted through the pipes mainly consists of signal values and timestamps, the information related to events is usually composed by more than one byte. The message mode use frees the programmer from the task of inspecting the limits of a given message. Again, this leads to a better data organization and simplifies the data protocol.

5.2.1.2 Data Messages Structure/Protocol

The data transmission protocol used in the messages transmitted between both SDs is rather simple. With respect to the data transmitted through signal data pipes, it carries information about the new values of one or more shared signals. As it is possible that, for a given simulation point, several signals change their values, a message can contain information relative to more than one shared signal. Under these circumstances, an SD which receives a message containing information about several shared signals must figure out which information corresponds to which signal. This way, an identification of each shared signal is required and this identification must be unique in the whole co-simulation environment (a shared signal must have the same identification in both SDs).

Apart from it, an SD which receives a message with multiple signal values must discern which part of the message refers to a signal value and which part refers to the following signal value. This can be achieved if one knows the amount of bytes/elements which constitute the signal value (for example, the *int* data type has a size of 4 bytes, both in *C++* and *C#*). Thus, a message transmitted in a signal data pipe follows the structure/protocol illustrated in Figure 5.2.

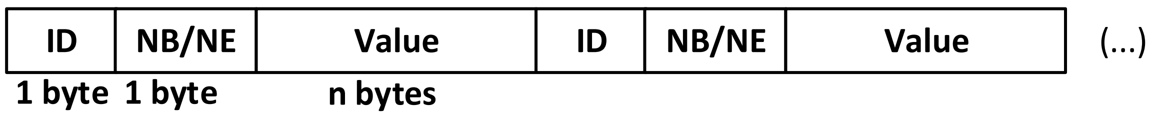


Figure 5.2: Data protocol used in signal data pipes

From the figure above, one can see that a signal data message consists of a stream of one or more $ID, NB/NE, Value$ triplets, in which:

- **ID** is one byte representing an unsigned integer ranging from 0 (0x00) to 254 (0xFE) (the value 0xFF is reserved for null messages). Shared signal IDs must be unique in the whole co-simulation system;
- **NB/NE** is one byte representing an unsigned integer ranging from 0 (0x00) to 255 (0xFF) which represents the amount of bytes or elements which constitute the signal value;
- **Value** consists of a variable number of bytes which represents the new value of a shared signal.

The read operations on pipes are *blocking*, meaning that, at certain moments during co-simulation, one SD will wait for a message from the other SD. It can happen that, at that moment, there are no signal changes to report. In order to avoid *deadlock* situations, *null messages* are passed between SDs when there is no data to transfer. A null message is composed by a single byte whose value is 255 (0xFF).

Regarding to the data transmitted through temporal data pipes, the messages only contain a timestamp which consists of a *double* value. This data types is similar in *C++* and *C#*, and in both languages the *double* data type size is 8 bytes. This value represents an amount of time units. The time unit considered in both SDs in the *pico-second* (ps).

5.2.1.3 Communication API

The functions intended for communication between SDs are defined and implemented in the files *cosim.h* and *cosim.cpp* (*SystemC* counterpart) and the *IPC* class from *CoSimulation* namespace (*System#* counterpart). The most relevant functions are similar in both languages and make use of the corresponding *Windows Named Pipes* APIs:

- **CreateAndConnectPipe():** this function is responsible for the communication set-up and establishment. Pipe parameters like read/write mode, read operation nature (blocking or non-blocking) or pipe buffer size are defined within this function;
- **ReadFromPipe():** it performs a read operation from a given pipe. A message consisting of a bytes stream is read and the read operation is blocking. The read message was previously written in the opposite pipe end;
- **WriteToPipe():** this function writes a bytes stream into a given pipe. The message can be read in the opposite pipe end;
- **ClosePipe():** flushes and disconnects a pipe.

This implementation considers that the *SystemC* counterpart behaves as a server and the *System#* behaves as a client. It means that the *SystemC* domain creates the pipe and waits for the *System#* domain connection to the created pipe. To avoid mistakes regarding to pipe identification, pipe names are uniquely defined in both SDs. All the Communication operations within the co-simulation mechanism are performed on the *Main Level* in both SDs. These operations include: pipe names and handle variables definition, connection set-up and establishment, all data transactions between SDs (read/write operations on pipes) and pipes closure.

5.2.2 Synchronization between *SystemC* and *System#*

In the developed work, synchronization between *SystemC* and *System#* was implemented through an adaptation of the conditional-event approach [39]. This adaptation is actually an algorithm simplification which assumes that, for every simulation time, all events within the co-simulation environment are *conditional events* (that is, they can be affected by another event), with the exception of the event with the lowest timestamp across all LPs, which is the only *unconditional* and safe event. In fact, this is equivalent to the conditional-event approach worst case.

In the co-simulation environment considered, there are only two LPs - SDs: *SystemC* and *System#*. So, for a given simulation time, the only event which is safe to process is the one with the smallest timestamp among all events present in *SystemC* and *System#* event-lists.

The implemented synchronization algorithm considers that the local clock variables from both SDs assume the same initial value (0 seconds) and evolve in the same way during the whole co-simulation period. This is an important issue regarding to Synchronization and it confers a tightly coupling and synchronization between both SDs:

$$t_{SIM}(co-simulation) = t_{SIM}(SystemC) = t_{SIM}(System\#)$$
$$\text{(for the whole simulation period)}$$

, being $t_{SIM}(co-simulation)$ the simulation time of the whole co-simulation mechanism, $t_{SIM}(SystemC)$ the simulation time in *SystemC* domain and $t_{SIM}(System\#)$ the simulation time in *System#* domain.

For a given simulation time, the co-simulation mechanism must be able to compute the amount of time simulation can be safely advanced in both SDs - *Lookahead*. From a generic perspective, one can consider two variables which help to define and compute lookahead in the developed approach: *Time-to-Next-Local-Event* (TNLE) and *Time-to-Next-Remote-Event* (TNRE).

Let's consider a co-simulation system comprising two SDs - SD_A and SD_B - each one having its own event list - l_A and l_B, respectively. The earliest events (the ones with the lowest timestamp) on l_A and l_B are e_A and e_B and their timestamps $t(e_A)$ and $t(e_B)$. For a given simulation time t_{SIM}, in SD_A, the Time-to-Next-Local-Event is:

$$TNLE(A) = t(e_A) - t_{SIM}$$

and the Time-to-Next-Remote-Event is:

$$TNRE(A) = t(e_B) - t_{SIM}$$

Following the same reasoning, for SD_B, we have:

$$TNLE(B) = t(e_B) - t_{SIM}$$
$$TNRE(B) = t(e_A) - t_{SIM}$$

Taking into account the adaptation of the conditional-event approach described above, the lookahead is computed in the following way:

$$Lookahed = min(TNLE(A), TNRE(A)) = min(TNRE(B), TNLE(B))$$

From a practical point of view, the synchronization algorithm implemented requires both SDs simulation kernels to be able to perform *single-step simulation* (in other words, *delta-cycle by delta-cycle simulation*). Beyond this, the SLDLs used should supply functions which provide information about the simulation scheduler. These functions should turn it possible to know if there is pending activity to perform at current or future simulation time, as well as the simulation time left until the next activity to be performed.

During implementation, an important question had to do with possible required changes on the SDs simulation kernels, in order to be integrated in the co-simulation environment. The implemented approach was designed having in mind the avoidance of modifications in *SystemC* and *System#* simulation kernels.

However, some changes in *System#* simulation kernel were actually performed. As *System#* simulation kernel didn't support *single-step simulation*, it was adapted towards the inclusion of that simulation feature. Apart from *single-step simulation*, the required functions for providing information about the simulation scheduler were added.

As *System#* is a recent framework, currently at a early development stage, the added simulation features can be viewed as improvements to the framework, which are useful to generic simulation purposes and not only for co-simulation purposes.

With respect to *SystemC*, modifications were avoided in order to keep the co-simulation mechanism compatible with future *SystemC* versions. Indeed, no *SystemC* simulation kernel changes were performed during the co-simulation mechanism implementation. All the required functionalities related with *single-step simulation* and scheduler information functions were already provided by the last released version of *SystemC* (IEEE 1666-2011).

Recalling SD's hierarchical levels presented in Figure 5.1, synchronization will be described in more detail by explaining them. It will be a *top-down* exposition, starting from the *Main Level* and finishing wih the *Sub-Modules Level*.

5.2.2.1 The *Main Level*

The *Main Level* of an SD is the highest hierarchical level concerning the co-simulation mechanism architecture. It comprises the *Top Module Level*, the *Sub-Modules Level* and also its own constructs. In terms of form, the *SystemC Main Level* consists of the *sc_main()* function, while *System#* counterpart consists of the *Program.Main()* function.

The *Main Level*'s generic structure is similar in both SDs. First, pipe variables (pipe names and handlers) and simulation parameters, like simulation temporal duration, are declared and defined. Regarding to these elements, coherency between both SDs is required, since some variables must be uniquely defined within the whole co-simulation mechanism. Then, the Top Module, which contains sub-systems one wants to co-simulate (*Sub-Modules Level*), is instantiated.

After Top Module instantiation, the "heart" of co-simulation - *co-simulation cycle* - is defined. Shortly, the co-simulation cycle operates a follows: for a given simulation time instant (the same in both SDs) both simulators perform *delta-cycle-by-delta-cycle simulation* (*single-step simulation*) until there is no more pending activity in both SDs. Once this point is reached, both SDs exchange information in order to compute the amount of simulation time both can safely progress. After time advancement, both SDs perform again single-step simulation.

This cycle is repeated until a certain end condition is met - when simulation time reaches the simulation temporal length initially defined. A more detailed co-simulation cycle analysis is exposed next, by following the flowchart presented in Figure 5.3.

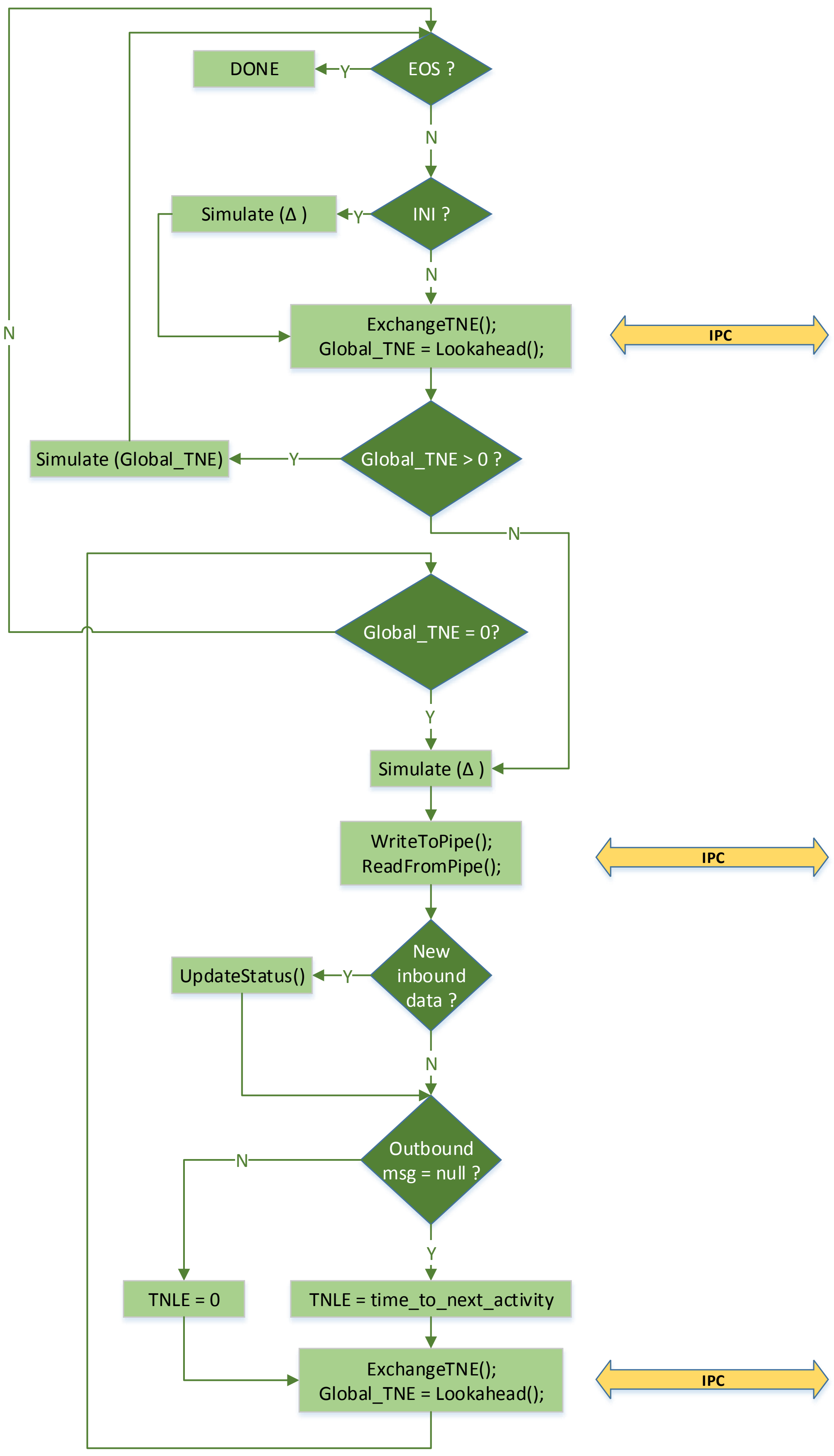


Figure 5.3: Co-simulation main cycle flowchart

The co-simulation cycle starts by verifying if the *End of Simulation* (EOS) was reached. If so, the co-simulation is finished. Otherwise, the cycle execution advances and checks if the current cycle iteration is the first one. Being the first cycle iteration, means that the local SD simulation has not yet started and some scheduler variables, which give information about scheduler activity, were not initialized. In order to initialize a SD's simulation process, a single delta-cycle is executed. If the current cycle iteration is not the first one, the initialization delta-cycle is skipped.

Then, the co-simulation cycle proceeds to an important stage, where the SDs exchange their TNLEs. To get the TNLE value, the SD makes use of constructs which provide information about scheduler activity. In *SystemC*, one can obtain the time until the earliest pending activity through the function:

sc_time sc_core::sc_time_to_pending_activity()

, while in *System#* this information is given by the following *C#* property:

Time DesignContext.Instance.TimeToPendingActivity

The TNLE exchange between both SDs is performed by the implemented function *ExchangeTNE()*. This function is defined and implemented in *cosim.h* and *cosim.cpp* files (*SystemC* counterpart) and in *CoSimulation.Synchronization* class (*System#* counterpart). In turn, *ExchangeTNE()* makes use of the Named Pipes API described in section 5.2.1.3 to read and write information. The data related with TNLE exchange flows in the *temporal data pipe*.

After TNLE exchange, SDs compute the *Global Time-to-Next-Event* (Global_TNE), that is the amount of time both SDs can safely advance - *Lookahead*. This amount of simulation time is calculated using another function implemented in *cosim.cpp* and *CoSimulation.Synchronization* class: *Lookahead()*. The lookahead computation is done the same way that was presented in section 5.2.2.

If *Global_TNE* is greater than zero, it means that there is no more activity to perform at current simulation time. Thus, the simulation time from both SDs can be advanced by *Global_TNE* time units. On the other hand, if *Global_TNE* is not greater than zero (meaning that it is equal to zero), there is some activity to perform at current simulation time in one or both SDs. Faced with this scenario, both SDs will perform *single-step simulation* until there is no more activity to perform in the co-simulation system at the considered simulation time.

After a delta-cycle simulation, both SDs write a message to the opposite SD, on the corresponding outbound pipe, containing information about eventual changes on outbound shared signals. Likewise, both SDs read, from the corresponding inbound pipe, a message from the opposite SD, which contains information about eventual changes on inbound shared signals values. These data transactions are performed on *signal data pipes*. The perception of *inbound*/*outbound* pipes, shared signals or messages is dual. For example, the *signal data pipe* dedicated to the *SystemC* $\longrightarrow$ *System#* data flow is the *inbound* pipe for the *System#* domain and the *outbound* pipe for the *SystemC* domain. The same happens with shared signals or messages. A signal which is connected to a *SystemC* module output and to a *System#* component input is viewed as a *SystemC* outbound signal and as a *System#* inbound signal.

The received inbound message is copied to a buffer from the Top Module and is inspected in order to figure out if it corresponds to a null message. If it is not a null message, a function defined in the Top Module is called. This function - *UpdateStatus()* - is responsible to set the conditions which allow the shared signals update when the simulation is resumed.

Afterwards, each SD computes again the simulation lookahead. However, before calling the function *ExchangeTNE()* and *Lookahead*, the SDs recall the outbound message previously sent. If the outbound message sent is different from the null message, this means that the SD sent a message containing information about new values on some shared signals. These new values can trigger events in the remote SD which, in turn, can affect the local SD at current simulation time. A SD cannot predict the effects an outbound message will have in the remote SD. Therefore, the implemented algorithm assumes that, each time a SD sends an outbound message different from the null message, the SD's TNLE is set to zero. On the other hand, if the outbound message is a null message, the TNLE is computed using *sc_core::sc_time_to_pending_activity()* or *DesignContext.Instance.TimeToPendingActivity*.

This algorithm assumption can force the execution of *empty* delta-cycles if an SD has no activity to perform while waits for possible effects from a not-null outbound message previously sent.

After TNLE definition, SDs proceed to TNLE exchange and lookahead computation. The new lookahead value (Global_TNE) is then inspected: if it is zero, Single-Step simulation runs once again; if not, a new co-simulation cycle iteration is started from the beginning.

Examples of the *Main Level* code for *SystemC* and *System#* are shown in Appendix A. The system considered in the example is the *Squirrel-Cage* induction machine used for evaluation in Chapter 6.

5.2.2.2 The *Top Module Level*

Both in *SystemC* and *System#* domains, the *Top Module Level* takes the shape of a typical module/componente in those languages. For a given SD, this module/component - *Top Module* - encapsulates the system's physical configuration. So, it contains the actual modules and components one wants to co-simulate, as well as the connections between them.

The Top Module itself has no input/output ports. The shared signals in the co-simulation environment are represented by auxiliary local channels (one for each shared signal). There are also two byte buffers - *InboundBuffer* and *OutboundBuffer* - which are accessible from the Main Level and are actually the bridge between that level and the Top Module Level. New values in outbound shared signals are inserted in the *OutboundBuffer* in the correct format, while new inbound shared signal values can be accessed through the *InboundBuffer*.

A Top Module declares an auxiliary variable and an event variable (an instance of *sc_event* in *SystemC*; an instance of *SystemSharp.Components.Event* in *System#*) for each inbound shared signal. The auxiliary variables store the new value of the respective inbound shared signal. The data type of these auxiliary variables is the data type of the correspondent signal, with the exception of logic values - the auxiliary variable is a *char* - and data types whose size is defined by the designer (for example, logic vectors, fixed-point values and arbitrary-length integers) - the auxiliary variable is a *char** in *SystemC* and a *StdLogicVector* in *System#*. The event variables are used to notify the system about new values in inbound shared signals and to trigger certain processes responsible for shared signals update.

Apart from the previously referred local channels and auxiliary variables declarations, the Top Module comprises instances of modules/components belonging to the *Sub-Modules Level* and also processes and other methods involved in operations over shared signals.

For each outbound shared signal, there is an associated triggered process sensitive to the respective signal. In other words, every time the value of an outbound shared signal changes, the triggered process associated to that signal is executed. By convention, these processes have the following prototype:

void out_<signal_name>()

, being *<signal_name>* the shared signal name to which the process is sensitive. Once triggered, these processes write the respective new outbound shared signal values in *OutboundBuffer*, according to the protocol described in 5.2.1.2.

For each inbound shared signal, there is a triggered process sensitive to an event associated to the respective inbound shared signal. The prototype of these processes is

void write_<signal_name>()

, where *<signal_name>* is the shared signal name associated to the event which triggers the process. A *write_<signal_name>()* process writes a new value, previously stored in an auxiliary variable, in the target shared signal.

Besides the referred processes, the Top Module from both SDs contains a routine whose prototype is

void UpdateStatus()

It is called by the Main Level during the co-simulation cycle, more precisely, after the delta-cycle simulation and the read/write operations on named pipes. This routine inspects *Inbound-Buffer*, seeking for new inbound shared signal values, puts them in auxiliary variables and notifies the event responsible for triggering the process associated with the inbound shared signal whose values has changed. The event notification is scheduled for the next delta-cycle in the simulation. The *UpdateStatus()* routine cannot change directly the shared signal values because it is called when simulation is paused, and signal values can only be modified when simulation is running.

Appendix A presents code examples for the *Top Module Level* in *SystemC* and *System#*. Again, the system considered is the the *Squirrel-Cage* induction machine used for evaluation in Chapter 6.

5.2.2.3 The *Sub-Modules Level*

The co-simulation mechanism is completely transparent to the *Sub-Modules Level*. Thus, this level simply consists of *SystemC* modules/*System#* components. All the operations related with communication and synchronization between SDs are contained in *Top Module Level* and *Main Level* which were described in the two previous subsections.

Chapter 6

Evaluation and Results

After developing the work aiming at this Master dissertation main goals, the obtained results were evaluated. That is what this chapter addresses. Similarly to the implemented work, which was divided in two parts, the results and evaluation were also divided in two parts, each one corresponding to one of the implementation tasks performed: *SystemC* code generation from *System#* projects and *SystemC*/*System#* co-simulation mechanism implementation.

6.1 *SystemC* code generation from *System#* projects

Regarding to *SystemC* code generation from *System#* projects, the milestone initially defined was the correct generation of *SystemC* code for a set of *System#* example projects included in the *System#* available for download at [24]. The example projects consist of:

- a system which computes the greatest common divisor (GCD) of two numbers;
- a counter. Different versions of the counter were considered by varying the counter data type: *StdLogicVector*, arbitrary-length integers (*Signed* and *Unsigned*) and fixed-point data types (*SFix* and *UFix*);
- a bus arbitration system where multiple bus masters issue arbitration requests and hold the grant for a variable amount of time;
- a bit serializer/deserializer (Ser/Des) system. For this project, *SystemC* code was also generated by treating the system as a FSM (using the *System#* ability to automatically generate FSMs from Clocked Threads). So, two versions of the Ser/Des system were generated: *FSM* and *no-FSM* versions.
- a *Squirrel-Cage* induction machine, which is a rotor used in the most ordinary form of an AC induction motor. The model used consists on the induction machine considered in [56].

Some of these example systems have a low level of complexity. However, this milestone seemed to be reasonable taking into account the available time window to develop the Master dissertation, as well as the fact that *SystemC* code generation is not the only goal of this thesis.

Using the code generation engine developed in Chapter 4, *SystemC* code was generated for all target projects previously referred. The generated code was successfully compiled and run for all projects. Additionally, the simulation results obtained with the generated *SystemC* code were exactly the same results obtained with the original *System#* projects.

Focusing on simulation performance, *SystemC* generated code performance showed to be better than *System#* projects performance, for all targeted projects. Table 6.1 presents average execution time from 10 simulation runs for some previously referred projects, in both in *SystemC* and *System#*. Performance evaluation was checked for the following projects: *Bus Arbitration* system with 10 bus masters and *Ser/Des* system (no-FSM version).

Project	**simulation duration**	**clock cycle**	***System#* average execution time (ms)**	**Generated *SystemC* average execution time (ms)**
Bus Arbitration system	100μs	10ns	280.8	14.9
Ser/Des system	7500ns	10ns	37.3	1.2

Table 6.1: Comparison of average execution time in *System#* and generated *SystemC*

The generated code performance for the *Squirrel-Cage* project was also measured. However, unlike the previous *System#* projects, the *Squirrel-Cage* project does not execute RTL simulation of the intended system. It still produces an RTL specification of the system, but only performs algorithmic simulation in which communication and computation are approximate-timed. This speeds up simulation and makes the comparison with RTL *SystemC* simulation unfair. So, regarding to the *Squirrel-Cage* project, the simulation of generated *SystemC* code was compared with the simulation of generated *VHDL* code (also obtained through *System#* framework). Once again, ten simulation runs of the project were executed and average execution time computed. Table 6.2 presents the results.

Project	**simulation duration**	**clock cycle**	**Generated *VHDL* avr. execution time (s)**	**Generated *SystemC* avr. execution time (s)**
Squirrel-Cage induction machine	8μs	4ns	6.6	3.5

Table 6.2: Comparison of average execution time in *VHDL* and generated *SystemC*

In all situations tested, the simulation using the generated *SystemC* code showed to be faster than *VHDL* or *System#* RTL simulations. The *SystemC* code generation engine is integrated in *System#* framework (similarly to the *VHDL* code generation engine), rather than being an independent and separate body. This makes it easier to embed *SystemC* code generation into *System#* design flow and to use it for system simulation and verification.

Indeed, the targeted *System#* projects, in which *SystemC* code generation was tested, do not cover all *System#* syntax. So, once the milestone initially defined was achieved, an important question arose: *which System# constructs are/aren't covered by the SystemC code generation* *engine?*. To have an overview about the *SystemC* code generation comprehensiveness, *System#* modelling properties were divided in three domains and the ability to generate correct *SystemC* code for each domain was evaluated. The considered domains were:

- **Components, Channels and Ports:** this domain covers the basic modelling principle of separation between computation and communication. Components are correctly identified and translated to *SystemC*. Regarding to communication, *System#* built-in channels *Signal*, *SLSignal* and *SLVSignal* were correctly translated to *SystemC*. The concept of *port* and the *port binding* procedure are also covered by *SystemC* code generation engine;

- **Concurrent Behaviour:** the concurrent behaviour is represented by processes. Both *SystemC* and *System#* support three kinds of processes: triggered processes, threaded processes and clocked thread processes. All of these kinds of processes are supported by *SystemC* code generation engine. The only remark has to do with the *System#* internal representation for clocked threads, which affects the way this kind of processes is generated in *SystemC* (as stated in 4.2.4);

- **Data Types:** more emphasis within this domain was given to *Bit-Accurate* data types and resolved logic, once these data types are commonly used in the project of embedded system. *System#* resolved logic data types (*StdLogic* and *StdLogicVector*) and its operations are covered by the implemented code generation engine. Operations and data type conversions involving arbitrary-length integers and fixed-point values were tested in some counter versions and more exhaustively in the *Squirrel-Cage* project. The produced results were correct for these data types.

6.2 *SystemC/System#* co-simulation mechanism

The co-simulation mechanism implemented was tested in three different test systems. The first one, which is illustrated in Figure 6.1, is composed by three modules. Two modules belong to *SystemC* SD - (*Producer* and *Logger*) - and one belongs to *System#* SD - (*Op*). There are two signals shared across both SDs: *value* (connected to *Producer*'s output port and *Op*'s input port) and *result* (connected to *Op*'s output port and *Logger*'s input port).

The system operation is simple. Every clock cycle, the *Producer* generates a new value and passes it to the *Op*, which performs an operation on it. The result from this operation is, in turn, passed to the *Logger*. This module simply prints in the console the values received from the *Op*. In the whole co-simulation system, the only module which is fed by a clock signal is the *Producer*. The remaining modules react to changes on their inputs values.

The main purpose of this test system - called *Producer-Op-Logger* - was not stressing the co-simulation mechanism, in order to challenge its accuracy. Instead, the main concern was to inspect if signal values were correctly transferred and understood in the remote SD. This test system was co-simulated for different scenarios:

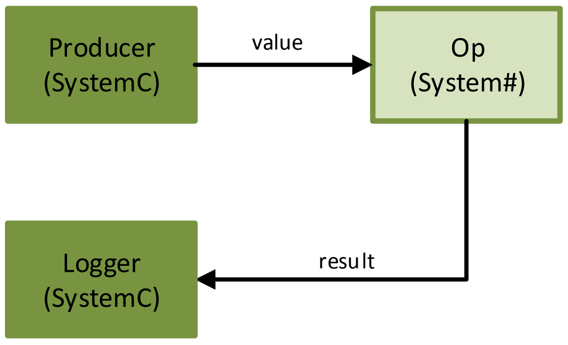


Figure 6.1: Scheme of test system 1: *Producer-Op-Logger*

1. the *Producer* generates a numeric value which is multiplied by 2 in *Op* and then passed to the *Logger*. The numeric value data type was varied: *int*, arbitrary-length integers(*sc_bigint*/*Signed*) and fixed-point values (*sc_fixed*/*SFix*);

2. the *Producer* generates a logic vector (*sc_lv*/*StdLogicVector*) and the *Op* module simply passes it to the *Logger*, without transforming it;

3. the *Producer* generates a logic value (*sc_logic*/*StdLogic*) which is logically inverted in *Op* and the passed to the *Logger*.

The second test system was a *Bus Arbitration* system with multiple bus masters shown is Figure 6.2. The *Arbiter* and the clock signal generator are modelled in *System#*, while the bus masters are implemented in *SystemC*. Regarding to shared signals and apart from *Clk* (all the bus masters are fed by the clock signal generated in *System#* domain), each bus master introduces more two shared signals: *Request_n* (connecting *Bus Master n*'s output port to one of the *Arbiter*'s input ports) and *Grant_n* (connecting one of the *Arbiter*'s outputs to a *Bus Master n*'s input). In all the shared signals, the transferred values are logic values (*sc_logic*/*StdLogic*).

The system operates as follows: the bus masters issue arbitration requests to the *Arbiter* and hold the received grant for a variable amount of time. Actually, this system is a replica of the bus arbitration system implemented in one of the *System#* example projects used for testing *SystemC* code generation. Furthermore, the code for the *SystemC* bus masters was obtained from the code generation engine. All modules in the system share the same clock signal and 10 bus master were considered in the design.

The third test system evaluated involved the *Squirrel-Cage* induction machine (Figure 6.3). Here, the system partition was the following: the *SystemC* SD comprises the whole rotor model, while the *System#* SD contains the clock signal generator and a *test-bench* which provides *stimuli* to the *Squirrel-Cage* model.

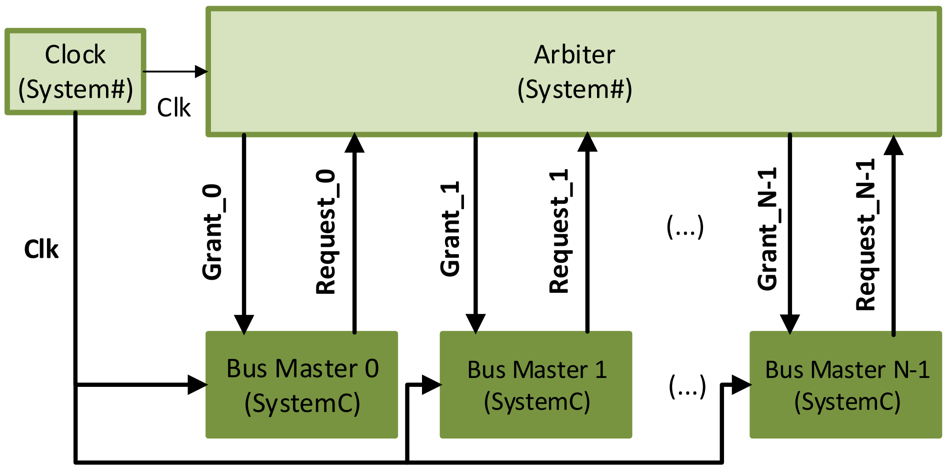


Figure 6.2: Scheme of test system 2: *Bus Arbitration*

This system partitioning tries to reflect a situation likely to happen in real systems design: develop a test-bench in a given language in order to stimulate a system/model projected in another language.

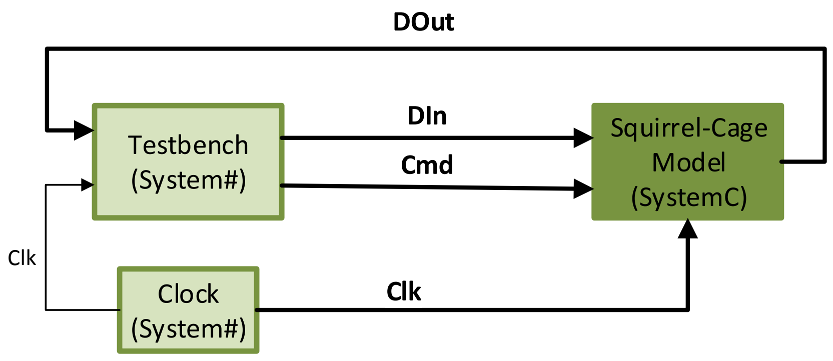


Figure 6.3: Scheme of test system 3: *Squirrel-Cage*

This way, the shared signals in the *System#* $\longrightarrow$ *SystemC* data flow are:

- **DIn:** *Squirrel-Cage* model input data. Corresponds to voltage values applied to the rotor. This data assumes the form of a logic vector with 56 elements;
- **Cmd:** a command which defines the action to perform by the rotor (define voltage values, fetch current values, etc.). The values of *Cmd* consist of enumeration values;

Project	simulation length	Clock Cycle	co-simulation average execution time (ms)
Bus Arbitration system	100μs	10ns	2014.4
Squirrel-Cage induction machine	8μs	4ns	4631.1

Table 6.3: Co-Simulation average execution time for *Bus Arbitration* and *Squirrel-Cage* systems

- **Clk:** the clock signal. Assumes a logic value.

In the opposite data flow direction, we have:

- **DOut:** *Squirrel-Cage* model output data. Consists of current values and others, produced by the rotor. Similarly to *DIn*, the values of this signal are logic vectors with 56 elements.

The code used for the *Squirrel-Cage* model in the *SystemC* SD was obtained through code generation engine previously developed.

The implemented co-simulation mechanism was employed to co-simulate the previously presented *SystemC/System#* heterogeneous systems. In all of them, the co-simulation results matched to the expected ones. The timing constraints inherent to systems operation were respected and both simulation domains showed to be correctly synchronized. In the case of the *Bus Arbitration* system and the *Squirrel-Cage* induction machine, the co-simulation results were the same as the ones obtained through the simulation of the equivalent *System#* project.

Comparing the implemented approach with the co-simulation mechanism features presented in 5.1, it is possible to state that the primary requirements defined were achieved. So, *the co-simulation results showed to be accurate and correct*, and the *modules/components written in one of the considered SLDLs need not to be changed in order to be used in co-simulation*. For example, the *SystemC* generated code used in the *Bus Arbitration* and *Squirrel-Cage* systems didn't suffer any change before being integrated in the test system.

Some performance measurements were also done for the *Bus Arbitration* and *Squirrel-Cage* systems. Co-simulation was run 10 times and the average execution time was computed. The execution time for a co-simulation run is the maximum between the execution times of *SystemC* and *System#* simulators. The results obtained from these measurements are shown in Table 6.3.

Comparing the execution times of co-simulation and single-engine simulation with *System#* and *SystemC* (Table 6.1) for the equivalent systems, it is possible to observe that co-simulation presents a substantially worse performance than the single-engine simulation. Some reasons may be cited as causes for the poor co-simulation mechanism performance:

- **high conservative nature of the synchronization algorithm implemented:** the introduction of *empty delta-cycles* is an example of extra computation added to the system due to high conservative synchronization;

- **communication operations overhead:** performance analysis shows that a higher percentage of execution time is associated with pipe read/write operations and between them, the *ReadFromPipe()* function is the one which absorbs a higher percentage of execution time (its *blocking* nature can contribute to this). Moreover, the null messages exchanged due to *empty delta-cycles* provoke the increasing of the amount of pipe operations performed;

- **tight coupling nature of some tested system:** the performance of distributed simulation or co-simulation mechanism is highly dependent on how the whole system is partitioned. If the system is appropriately partitioned, by assigning well balanced workloads to the different system parts, better performance is expected. In the case of the *Bus Arbitration* system tested, one SD (*System#*) comprises more modules and more scheduled activity than the other SD. Additionally, the system clock tree requires the clock signal to be spread across all SDs and components, introducing a signal data flow for every clock edge.

The implemented co-simulation mechanism requires considerable interaction with the user in order to configure the co-simulation environment in the desired way. From the three levels which constitute the co-simulation architecture, only the *Sub-Modules Level* does not require interaction with the user. In the Main Level, some named pipes and simulation parameters have to be manually defined. Port binding to shared signals, processes, methods and event definitions need to be configured by the user in the Top Module Level. Thus, the co-simulation mechanism has lack of transparency.

6.3 A design guideline: split the clock signal by both SDs

Recalling the co-simulation performance driving forces previously referred, a small design change was executed in the systems to co-simulate (*Bus Arbitration* and *Squirrel-Cage*). Instead of sharing the clock signal across both SDs, two instances of the clock signal were considered (one for each SD) - *split-clock*. To keep the coherency of the system, both clock signals must have the same behaviour. This design change aims to reduce the communication operations overhead in the co-simulation system, once it decreases, at least, the amount of data transferred between both SDs (one less shared signal in the system). At the first glance, this change seems to not represent a meaningful impact on the co-simulation performance because the main problem regarding to Communication overhead is not related with data throughput, but with the amount of calls to read/write operations on named pipes.

The *split-clock* approach was applied to the *Bus Arbitration* and *Squirrel-Cage* systems and the number of *null messages* and *non-null messages* sent by each SD was inspected. The obtained values were then compared to the ones from the co-simulation version where the clock-signal is shared among both SDs - *shared-clock* approach.

While co-simulating the *Bus Arbitration* system, the number of bus masters - N - was varied and the simulation length was adjusted in order to allow every bus master to acquire and release the bus once. The clock cycle (10ns) was kept constant while changing the number of bus masters.

N	simulation length		SD	*Shared-clock*	*Split-clock*
2	100ns	*# null msgs sent*	*SystemC*	106	54
			System#	85	53
		# non-null msgs sent	*SystemC*	4	4
			System#	25	5
3	150ns	*# null msgs sent*	*SystemC*	157	77
			System#	126	76
		# non-null msgs sent	*SystemC*	5	5
			System#	36	6
4	250ns	*# null msgs sent*	*SystemC*	258	126
			System#	205	123
		# non-null msgs sent	*SystemC*	6	6
			System#	59	9
5	300ns	*# null msgs sent*	*SystemC*	309	149
			System#	246	146
		# non-null msgs sent	*SystemC*	7	7
			System#	70	10
10	1000ns	*# null msgs sent*	*SystemC*	1019	475
			System#	810	466
		# non-null msgs sent	*SystemC*	17	17
			System#	226	26

Table 6.4: Messages flow for the *Bus Arbitration* system co-simulation

Some observations can be made from Table 6.4. In the *shared-clock* configuration, the number of *null messages* sent is bigger in the *SystemC* SD. However, this fact has to do with the system partitioning. In fact, the activity in *SystemC* domain highly depends on the clock signal that is generated in the *System#* domain. Thus, *SystemC* is often waiting for information about the clock signal in order to trigger its processes. While there is no triggered activity, *SystemC* SD simply sends *null messages*.

Comparing the messages flow for *shared-clock* and *split-clock* situations, one can observe that the amount of *null messages* sent decreases in both SDs. So, it means that the amount of pipe operations also decreases. Furthermore, in the *split-clock* case, the number of *non-null messages* sent remains the same in *SystemC* domain, but considerably decreases in the *System#* domain. The nature of *shared-clock* and *split-clock* configurations can help to understand this reduction on *non-null messages sent*. In *shared-clock* situation, the clock signal is generated in the *System#* SD and is treated as another shared signal. So, whenever there is a change in the clock signal, a non-null message containing the new clock value is sent. On the other hand, in the *split-clock* situation, the clock signal is not a shared signal and changes on its value are not transmitted. This way, the non-null messages which would be sent every clock edge, and whose content only consists of the clock signal value, do not exist any more in the co-simulation system.

In the *Bus Arbitration* system, the *Request_N* and *Grant_N* signals only change their value on positive clock edges. Thereby, the write operations on these signals occur, at least, one *delta cycle* after the execution of the write operation on the clock signal (the new clock value is only

N	**simulation length**	**Shared-clock average execution time (ms)**	**Split-clock average execution time (ms)**	**Performance improvement**
2	100ns	1763.3	1115.1	36.8%
3	150ns	1777.1	1144.1	35.6%
4	250ns	1828.3	1165.2	36.3%
5	300ns	1875.1	1199.3	36.0%
10	1000ns	2003.4	1322.2	34.0%

Table 6.5: Co-Simulation average execution time for *Bus Arbitration*, considering the *shared-clock* and *split-clock* situations

available in the next *delta cycle*). As the messages exchange in the co-simulation mechanism take place after every *delta cycle* execution, the messages sent due to an event on the clock signal only contain information about the clock signal.

Actually, one can verify that the difference between the number of non-null messages sent by *System#* domain in *shared-clock* and *split-clock* situations is equal to the number of clock edges which occur within the simulation length. For example, considering the situation for N = 5:

Simulation length = 300ns; Clock cycle = 10ns
# clock edges = 2 * (Simulation length / Clock cycle) = 60
non-null messages reduction = 70 - 10 = 60

Having observed the impact on the amount of messages exchanged between both SDs decreases introduced by the *split-clock* situation, performance measurements were executed for the *Bus Arbitration* system co-simulation. Ten co-simulation runs were performed and the average execution time calculated. Table 6.5 present the obtained results.

A similar procedure regarding to the study of messages flow and performance (average execution time from 10 co-simulation runs) was done for the *Squirrel-Cage* system. Tables 6.6 and 6.7 present the results.

Simulation length		**SD**	***Shared-clock***	***Split-clock***
8μs	*# null msgs sent*	*SystemC*	26896	15038
		System#	22875	15017
	# non-null msgs sent	*SystemC*	54	54
		System#	4075	75

Table 6.6: Messages flow for the *Squirrel-Cage* system co-simulation

simulation length	**Shared-clock average execution time (ms)**	**Split-clock average execution time (ms)**	**Performance improvement**
8μs	4631.1	4255.4	8.1%

Table 6.7: Co-Simulation average execution time for *Squirrel-Cage*, considering the *shared-clock* and *split-clock* situations

The *split-clock* variation presented before cannot be viewed as a performance improvement to the co-simulation system, once the communication and synchronization mechanisms remain unchanged. Instead, it can be viewed as a design guideline: clock signals or signals whose behaviour is well known and defined from the beginning of the co-simulation procedure should not be shared across SDs.

Chapter 7

Conclusions and Outlook

The current chapter summarizes the main conclusions from the developed work, refers its contributions and point future work directions.

7.1 Conclusions

The work developed within this Master Dissertation is mainly related with two domains - *system level description languages* and *discrete-event (co-)simulation* - and the goals initially defined are deeply attached to those domains. *SystemC* and *System#* were the SLDLs considered in this work. The former is a popular and well-established hardware/software simulation tool and the later is a newcomer platform which extends its functionalities towards *high-level synthesis*, aiming the acceleration and automatization of the *Modelica-to-FPGA* design flow. The goal related with SLDL domain was to furnish *System#* with the ability to generate *SystemC* code for a given *System#* project. This feature was implemented and integrated in the *System#* framework and thus can be used during the *System#* design flow. Since performance measurements presented show the *SystemC* executes simulation faster than *System#*, the *SystemC* code generation feature can be used to address the problem related with the time-consuming nature of the simulation/verification loop done by *System#* prior to FPGA integration.

Being a recent design and simulation tool, *System#* still has a considerable improvement margin. In fact, at the time of this dissertation development, an *FZI* student was working on improving *System#* simulation performance using the new *C#* 5.0 features [57]. Nevertheless, the *SystemC* code generation is still an interesting feature which contributes to enlarge the *high-level synthesis* capabilities of *System#*.

The advent of *heterogeneous systems* came along with the increasing complexity of embedded systems. Consequently, the need for running a simulation procedure which in turn comprises simulators written in different languages arose. Ultimately, this is a problem of parallel/distributed *discrete-event simulation*. To face the challenge of simulating systems composed by elements designed in *SystemC* and *System#*, a co-simulation mechanism was implemented. The primary requirements of accuracy and modularity were successfully achieved by the implemented approach, through a high conservative synchronization algorithm and the existence of a *Top Module* level in each Simulation Domain which encapsulates the modules/components to co-simulate. Communication between SDs is performed using named pipes. The drawbacks of the implemented co-simulation mechanism are the lack of transparency for the user and the poor performance.

The co-simulation mechanism developed within this Master dissertation was the first attempt to co-simulate systems composed of *SystemC* and *System#* parts. However, it can be compared with some developed work in the field of parallel/distributed *SystemC*. The works from Trams [47], Cox [48] and Chopard *et. al* [49] are more related with the work developed within this project. In all of them, conservative synchronization is employed. In the *SystemC/System#* co-simulation, the synchronization algorithm implemented is similar to the ones used by Cox and Chopard *et. al*, but is different from Trams approach. Unlike Trams, the implemented synchronization algorithm is not constrained to systems with a regular and predictable behaviour. In his work, Cox actually modifies the *SystemC* kernel in order to produce a distributed version of it. On the other hand, Chopard *et. al* present a new *SystemC* construct used to partition the system to simulate. Instead, in this Master dissertation work, no changes were performed on *SystemC* kernel for sake of compatibility issues with future versions. The features added to *System#* kernel did not have a radical impact on the existing structure, but contributed to add some useful constructs related with single-step simulation. Regarding to communication between SDs, the implemented approach differs from the related referred work, once it employs named pipes instead of MPI or TCP/IP.

7.2 Future Work

The developed work has a considerable margin of progress. The *SystemC* code generation engine should be kept updated, following future new *SystemC* features. As referred in Chapter 6, more relevance was given to bit-accurate data types. So, the code generation engine should be exhaustively tested with a bigger variety of systems, dealing with different data types. An interesting contribution involving code generation would be the development of the ability to generate *SystemC* code at simulation time, seeking for possible benefits from convenient modelling and better simulation performance.

Regarding to *SystemC/System#* co-simulation mechanism the improvement window is wide. As referred before, the main drawbacks of the implemented approach are the lack of transparency and performance. Encapsulate the shared channels and all the communication and synchronization constructs in SLDLs constructs with a easy-to-use interface would clearly improve the usability and transparency of the whole co-simulation environment. From a user perspective, it would be convenient that the execution of a co-simulation run would look similar to a single-engine simulation. The development of a simulation platform, whose inputs would simply be the modules/components to co-simulate, and that would automatically set the co-simulation up and run it, would be a ultimate contribution for the simulation of heterogeneous systems and also for distributed/parallel simulation of homogeneous systems.

The co-simulation performance is a target for future work. The causes of poor co-simulation performance were identified in Chapter 6. Two of them - *high conservative nature of synchronization algorithm* and *communication overhead* - are intrinsically related with the co-simulation architectures and operation. These two causes drive to two different directions on improving system performance:

- Adapt or change the synchronization algorithm in order to relax the synchronization without affecting the co-simulation accuracy;
- Optimize the communication between SDs, using different communication mechanisms or adapting existing ones to this particular application.

Appendix A

Appendix A - Examples of co-simulation code

The *Main Level* assumes the shape of a *.cpp* file in *SystemC* and a class in *System#.*. Here, the communication between Simulation Domains is established, simulation parameters are defined, the *Top Mudule* instantiated and the co-simulation cycle is implemented.

***SystemC Main Level* code**

```
#define SC_INCLUDE_FX
#include <systemc>
#include "cosim.h"
#include <stdlib.h>
#include <iostream>
#include "top.h"
#include "time.h"
using namespace sc_core;
using namespace sc_dt;
using namespace std;

int sc_main(int sc_argc, char* sc_argv[])
{
// Named Pipes IPC variables
HANDLE SyncPipe = INVALID_HANDLE_VALUE;
HANDLE InboundPipe = INVALID_HANDLE_VALUE;
HANDLE OutboundPipe = INVALID_HANDLE_VALUE;

BYTE* SyncBuffer = (BYTE*)calloc(BUFFER_SIZE, sizeof(BYTE));

LPCTSTR SyncPipeName = L"\\\\" SERVER_NAME L"\\pipe\\sync";
LPCTSTR InPipeName = L"\\\\" SERVER_NAME L"\\pipe\\pipe1";
LPCTSTR OutPipeName = L"\\\\" SERVER_NAME L"\\pipe\\pipe2";

// Synchronization variables
sc_time TNE;
sc_time sim_length = sc_time(8, SC_US);
sc_time tnle, tnre;

// Aux variables
BYTE* aux_buf = (BYTE*)calloc(BUFFER_SIZE+8, sizeof(BYTE));
bool begin = true;

clock_t start, end;

//// Create and Connect Named Pipes
if((SyncPipe = CreateAndConnectPipe(SyncPipeName, "InOut")) ==
    INVALID_HANDLE_VALUE)
        return -1;

if((InboundPipe = CreateAndConnectPipe(InPipeName, "InOut")) ==
    INVALID_HANDLE_VALUE)
        return -1;

if((OutboundPipe = CreateAndConnectPipe(OutPipeName, "InOut"))
    == INVALID_HANDLE_VALUE)
        return -1;

// Instantiate the project to simulate
top T("top");

sc_report_handler::set_actions (SC_WARNING, SC_DO_NOTHING);

start = clock();

// Co-simulation Cycle
while(sc_time_stamp() <= sim_length)
{

        if (begin)
        {
                begin = false;
                sc_start(SC_ZERO_TIME);
        }

        tnre = ExchangeTNE(SyncPipe, sc_time_to_pending_activity
            (), SyncBuffer);
        TNE = LookAhead(sc_time_to_pending_activity(), tnre);

        if(TNE > sc_time(0, SC_PS))
        {
                sc_start(TNE);  // Advance time
        }
        else
        {
                while(TNE == sc_time(0, SC_PS))
                {
                        sc_start(SC_ZERO_TIME);

                        // Outbound Sync
                        if(T.outbufsize > 0)
                        {
                                WriteToPipe(OutboundPipe, L"
                                    Outbound", T.OutboundBuffer,
                                    T.outbufsize);
                        }
                        else
                        {
                                aux_buf[0] = 0xff;
                                WriteToPipe(OutboundPipe, L"
                                    Outbound", aux_buf, 1);
                        }

                        // Inbound Sync
                        T.inbufsize = ReadFromPipe(InboundPipe,
                            L"Inbound", T.InboundBuffer);

                        // Update
                        if(T.inbufsize > 1)
                                T.UpdateStatus();

                        // Compute Lookahead
```

```
                              if(T.outbufsize == 0)
                                     tnle =
                                        sc_time_to_pending_activity()
                                        ;
                              else
                                     tnle = sc_time(0, SC_PS);

                              T.outbufsize = 0;
                              tnre = ExchangeTNE(SyncPipe, tnle,
                                 SyncBuffer);
                              TNE = LookAhead(tnle, tnre);
                       }
                }
         }

end = clock();

ClosePipe(SyncPipe, SyncBuffer);
ClosePipe(InboundPipe, T.InboundBuffer);
ClosePipe(OutboundPipe, T.OutboundBuffer);

cout << endl << "Execution␣Time␣(ms):␣" << (1000*(end-start))/
   CLOCKS_PER_SEC << endl;

return 0;
}
```

***System# Main Level* code**

```
class Program
{
static void Main(string[] args)
{
       // Named Pipes IPC variables
       NamedPipeClientStream SyncPipe;
       NamedPipeClientStream InboundPipe;
       NamedPipeClientStream OutboundPipe;

       byte[] SyncBuffer = new byte[CoSimulation.IPC.BufferSize
          ];
```

```
string SyncPipeName = "sync";
string OutPipeName = "pipe1";
string InPipeName = "pipe2";

// Synchronization variables
Time tnle, tnre;
Time TNE = new Time(0, ETimeUnit.ps);
Time sim_length = new Time(8, ETimeUnit.us);



// Aux variables
byte[] aux_buf = new byte[CoSimulation.IPC.BufferSize +
   8];
bool begin = true;

// Create and Connect Named Pipes
SyncPipe = CoSimulation.IPC.CreateAndConnectPipe(
   SyncPipeName, "InOut");
OutboundPipe = CoSimulation.IPC.CreateAndConnectPipe(
   OutPipeName, "InOut");
InboundPipe = CoSimulation.IPC.CreateAndConnectPipe(
   InPipeName, "InOut");
ProcessPool.MaxParallelProcessesPreset = 4;
FixedPointSettings.GlobalDefaultRadix = 10;

Top T = new Top(24, 32);

DesignContext.Instance.Elaborate();

long start = DateTime.Now.Ticks;

// Co-Simulation Cycle

while (DesignContext.Instance.CurTime <= sim_length)
{
          if (begin)
          {
                    begin = false;
                    DesignContext.Instance.Simulate(0);
          }
```

```
tnre = CoSimulation.Synchronization.ExchangeTNE(
    SyncPipe, DesignContext.Instance.
    TimeToPendingActivity, SyncBuffer);
TNE = CoSimulation.Synchronization.LookAhead(
    DesignContext.Instance.TimeToPendingActivity,
     tnre);

if (TNE.Value > 0)
{
          DesignContext.Instance.Simulate(TNE);
              // Advance time
}
else
{
          while (TNE.Value == 0)
          {
                    DesignContext.Instance.Simulate
                        (0);

                    // Outbound Sync
                    if (T.outbufsize > 0)
                    {
                              CoSimulation.IPC.
                                  WriteToPipe(
                                  OutboundPipe, "
                                  Outbound", T.
                                  OutboundBuffer, T.
                                  outbufsize);
                    }
                    else
                    {
                              aux_buf[0] = 0xff;
                              CoSimulation.IPC.
                                  WriteToPipe(
                                  OutboundPipe, "
                                  Outbound", aux_buf,
                                  1);
                    }
```

```
                                        // Inbound Sync
                                        T.InboundBuffer = CoSimulation.
                                           IPC.ReadFromPipe(InboundPipe,
                                            "Inbound");
                                        T.inbufsize = T.InboundBuffer.
                                           Length;

                                        // Update
                                        if (T.inbufsize > 1)
                                                T.UpdateStatus();

                                        // Compute Lookahead
                                        if (T.outbufsize == 0)
                                                tnle = DesignContext.
                                                   Instance.
                                                   TimeToPendingActivity
                                                   ;
                                        else
                                                tnle = new Time(0,
                                                   ETimeUnit.ps);

                                        T.outbufsize = 0;
                                        tnre = CoSimulation.
                                           Synchronization.ExchangeTNE(
                                           SyncPipe, tnle, SyncBuffer);
                                        TNE = CoSimulation.
                                           Synchronization.LookAhead(
                                           tnle, tnre);
                                }
                        }
                }

                long execTime = DateTime.Now.Ticks - start;

                CoSimulation.IPC.ClosePipe(SyncPipe);
                CoSimulation.IPC.ClosePipe(InboundPipe);
                CoSimulation.IPC.ClosePipe(OutboundPipe);

                Console.WriteLine();
```

```
            Console.WriteLine("Execution␣Time␣(ms):␣{0}", new
               TimeSpan(execTime).TotalMilliseconds.ToString());
}
}
```

The *Top Module Level* consists of a module/component in *SystemC*/*System#* which comprises the declaration of auxiliary variables, shared signals, synchronization events, instantiation of the modules belongign to the *Sub-Modules Level*, as well as processes and functions important during co-simulation.

***SystemC Top Module Level* code**

```
#define SC_INCLUDE_FX
#include <systemc>
#include "SystemSharp_DataTypes.h"
#include "SquirrelCage.h"
#include "System.h"
#include "m_clkGen.h"
#include "m_dut.h"
#include "cosim.h"
#include <iostream>
#include "sc_lv_add_ons.h"
#include "sc_logic_add_ons.h"
using namespace std;
using namespace sc_core;
using namespace sc_dt;

SC_MODULE(top)
{
// Co-Simulation aux signals variables
char clk_value;
int cmd_value;
char* din_value;

// Co-Simulation signals
sc_signal<sc_logic> _clk;
sc_signal<int> _cmd;
sc_signal<sc_lv<56>> _din;
sc_signal<sc_lv<56>> _dout;

// Synchronization events
sc_event e_clk, e_cmd, e_din;
```

```
BYTE* InboundBuffer;
BYTE* OutboundBuffer;

int inbufsize, outbufsize;

// System components/sub-modules
m_dut m_dut_;

void writer_clk()
{
        (_clk).write(sc_logic(clk_value));
}

void writer_cmd()
{
        (_cmd).write(cmd_value);
}

void writer_din()
{
        (_din).write(sc_lv<56>(din_value));
}

void out_dout()
{
        if((_dout).event())
        {
                sc_lvOut(0x04, &(m_dut_.DOut), OutboundBuffer, &
                   outbufsize);
        }
}

void UpdateStatus()
{
        int i = 0;

        if(inbufsize > 1)
        {
                while(i < inbufsize)
                {
                        switch (InboundBuffer[i])
                        {
                                case 0x01:
                                {
                                        UpdateSimpleType(
                                           InboundBuffer, &i, &
                                           clk_value, &e_clk);
                                        break;
                                }

                                case 0x02:
                                {
                                        UpdateSimpleType(
                                           InboundBuffer, &i, &
                                           cmd_value, &e_cmd);
                                        break;
                                }

                                case 0x03:
                                {
                                        UpdateBitAccType(
                                           InboundBuffer, &i,
                                           din_value, &e_din);
                                        break;
                                }
                        }
                }
        }

}

SC_CTOR(top): m_dut_("m_dut_")
{
        m_dut_.Clk(_clk);
        m_dut_.Cmd(_cmd);
        m_dut_.DIn(_din);
        m_dut_.DOut(_dout);

        SC_METHOD(writer_clk);
```

```
                dont_initialize();
                sensitive << e_clk;

        SC_METHOD(writer_cmd);
                dont_initialize();
                sensitive << e_cmd;

        SC_METHOD(writer_din);
                dont_initialize();
                sensitive << e_din;

        SC_METHOD(out_dout);
                sensitive << _dout;

        InboundBuffer = (BYTE*)calloc(BUFFER_SIZE, sizeof(BYTE))
            ;
        OutboundBuffer = (BYTE*)calloc(BUFFER_SIZE, sizeof(BYTE)
            );

        din_value = (char*)malloc((56+1)*sizeof(char));

        inbufsize = 0;
        outbufsize = 0;
}
};
```

***System# Top Module Level* code**

```
class Top : Component
{
public static readonly Time ClockCycle = new Time(4.0, ETimeUnit
    .ns);

// Co-Simulation signals
public SLSignal _clk = new SLSignal();
public Signal<Testbench.ECmd> _cmd = new Signal<Testbench.ECmd
    >();
public SLVSignal _din;
public SLVSignal _dout;

// Co-Simulation aux variables
```

```
StdLogicVector _dout_value;

private Clock _clkgen;
private TestbenchSFix tb;

// Boundary buffers
public byte[] OutboundBuffer;
public byte[] InboundBuffer;
public int inbufsize, outbufsize;

// Synchronization Events
Event e_dout;

public Top(int iw, int fw)
{
    _din = new SLVSignal(iw + fw);
    _dout = new SLVSignal(iw + fw);

    _din.InitialValue = new string('U', iw + fw);
    _dout.InitialValue = new string('U', iw + fw);
    _cmd.InitialValue = Testbench.ECmd.Nop;
    _clk.InitialValue = 'U';

    _clkgen = new Clock(ClockCycle)
    {
        Clk = _clk
    };

    tb = new TestbenchSFix(iw, fw)
    {
        Clk = _clk,
        Cmd = _cmd,
        DIn = _din,
        DOut = _dout
    };

    InboundBuffer = new byte[CoSimulation.IPC.BufferSize];
    OutboundBuffer = new byte[CoSimulation.IPC.BufferSize];

    inbufsize = 0;
    outbufsize = 0;

    _dout_value = new string('U', tb.IW + tb.FW);

    e_dout = new Event(this);
}

private void writer_dout()
{
    _dout.Next = _dout_value;
}


private void out_clk()
{
    CoSimulation.BufferOperations.StdLogicOut(0x01, _clk, ref
       OutboundBuffer, ref outbufsize);
}

private void out_cmd()
{
    CoSimulation.BufferOperations.CppTypeOut(0x02, sizeof(int),
       ref OutboundBuffer, outbufsize);
    Array.Copy(BitConverter.GetBytes((int)_cmd.Cur), 0,
       OutboundBuffer, outbufsize + 2, sizeof(int));
    outbufsize += 2 + sizeof(int);
}

private void out_din()
{
    CoSimulation.BufferOperations.StdLVOut(0x03, _din, ref
       OutboundBuffer, ref outbufsize);
}

public void UpdateStatus()
{
    int i = 0;

    while (i < inbufsize)
    {
        switch (InboundBuffer[i])
        {
            case 0x04:
                {
                    CoSimulation.BufferOperations.
                       UpdateBitAccType(InboundBuffer, ref i,
                       ref _dout_value, e_dout);
                    break;
                }
        }
    }
}

protected override void Initialize()
{
    AddProcess(out_clk, _clk.ChangedEvent);
    AddProcess(out_cmd, _cmd.ChangedEvent);
    AddProcess(out_din, _din.ChangedEvent);

    AddProcess(writer_dout, e_dout);
}
}
```

References

[1] Blochwitz T. Koellner, C. and T. Hodrius. Translating modelica to hdl: An automated design flow for fpga-based real-time hardware-in-the-loop simulations. *Proceedings of the 9th International Modelica Conference*, 2012.

[2] Systemc resources- oops, October 2012. URL: `http://oopsproject.info/systemc.php`.

[3] Alois Ferscha. Parallel and distributed simulation of discrete event systems. 1995.

[4] C. Kollner, F. Mendoza, and K.D. Muller-Glaser. Modeling for synthesis with system#. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 470 –476, may 2012.

[5] M. Bacic. On hardware-in-the-loop simulation. In *CDC-ECC '05. 44th IEEE Conference on Decision and Control 2005 and 2005 European Control Conference.*, pages 3194 – 3198, dec. 2005. `doi:10.1109/CDC.2005.1582653`.

[6] Modelica and the modelica association - modelica association, October 2012. URL: `https://www.modelica.org/`.

[7] Set gmbh | e-motor emulatoren - motoremulation - testsysteme motorsteuerung - nachbildung des verhaltens von elektromotoren auf basis mathematischer modelle, October 2012. URL: `http://www.smart-e-tech.de/produkte/e-motor-emulatoren.html`.

[8] H. Elmqvist, M. Otter, and F. E. Cellier. Inline integration: A new mixed symbolic/numeric approach for solving differential-algebraic equation systems. In *Proceedings of the European Simulation Multiconference (ESM), 1995*, 1995.

[9] M. Dubois, El Mostapha Aboulhamid, and F. Rousseau. Towards an efficient simulation of multi-language descriptions of heterogeneous systems. In *APCCAS 2006. IEEE Asia Pacific Conference on Circuits and Systems, 2006*, pages 538 –541, dec. 2006. `doi:10.1109/APCCAS.2006.342527`.

[10] M. Dubois, E.-M. Aboulhamid, and F. Rousseau. Acceleration for heterogeneous systems cosimulation. In *ICECS 2007. 14th IEEE International Conference on Electronics, Circuits and Systems, 2007.*, pages 294 –297, dec. 2007. `doi:10.1109/ICECS.2007.4510988`.

[11] A. Amory, F. Moraes, L. Oliveira, N. Calazans, and F. Hessel. A heterogeneous and distributed co-simulation environment [hardware/software]. In *Proceedings. 15th Symposium on Integrated Circuits and Systems Design, 2002.*, pages 115 – 120, 2002. `doi:10.1109/SBCCI.2002.1137646`.

[12] The specc system, October 2012. URL: `http://www.cecs.uci.edu/~specc/`.

[13] Iec standard for systemverilog - unified hardware design, specification, and verification language (adoption of ieee std 1800-2005). *IEC 62530:2007 (E)*, 2007. `doi:10.1109/IEEESTD.2007.4410440`.

[14] Openvera website, October 2012. URL: `http://www.open-vera.com/`.

[15] Webhome < p1647 < twiki, October 2012. URL: `http://www.eda.org/twiki/bin/view.cgi/P1647/WebHome`.

[16] Brian Bailey and Grant Martin. *ESL Models and their Application: Electronic System Level Design and Verification in Practice*. Springer Publishing Company, Incorporated, 1st edition, 2009.

[17] .net downloads, developer resources & case studies | microsoft .net framework, October 2012. URL: `http://www.microsoft.com/net`.

[18] E.M. Aboulhamid and J. Lapalme. Recent trends in hardware/software description languages. In *Proceedings of the 15th International Conference on Microelectronics, 2003. ICM 2003.*, pages 185 – 188, dec. 2003. `doi:10.1109/ICM.2003.1287761`.

[19] J. Lapalme, E. M. Aboulhamid, G. Nicolescu, L. Charest, F. R. Boyer, J. P. David, and G. Bois. Esys.net: a new solution for embedded systems modeling and simulation. *SIGPLAN Not.*, 39(7):107–114, June 2004.

[20] Ieee standard for standard systemc language reference manual. *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, 9 2012. `doi:10.1109/IEEESTD.2012.6134619`.

[21] Doulos. Systemc in europe - current usage and future requirements. Technical report, 2003. URL: `www.doulos.com`.

[22] L. Cai and D. Gajski. Transaction level modeling: an overview. In *First IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, 2003.*, pages 19 –24, oct. 2003. `doi:10.1109/CODESS.2003.1275250`.

[23] Doulos. *SystemC Golden Reference Guide*. 2006.

[24] systemsharp - a c#-based system-level design framework with vhdl output - google project hosting, October 2012. URL: `http://code.google.com/p/systemsharp/`.

[25] R. M. Fujimoto. Parallel discrete event simulation. *Commun. ACM*, 33(10):30–53, October 1990. `doi:10.1145/84537.84545`.

[26] Rainer Dömer, Weiwei Chen, Xu Han, and Andreas Gerstlauer. Multi-core parallel simulation of system-level description languages. In *Proceedings of the 16th Asia and South Pacific Design Automation Conference*, ASPDAC '11, pages 311–316, Piscataway, NJ, USA, 2011. IEEE Press.

[27] J. Liu. Parallel discrete-event simulation. 2009.

[28] Jayadev Misra. Distributed discrete-event simulation. *ACM Comput. Surv.*, 18(1):39–65, March 1986. `doi:10.1145/6462.6485`.

[29] R. Fujimoto. *Parallel and Distributed Simulation Systems*. Wiley Interscience, 2000.

[30] Sherman R. Chandy, K.M. Space-time and simulation. *Proceedings of the 1989 SCS Multiconference on Distributed Simulation*, March 1989.

[31] K.M. Chandy and J. Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Transactions on Software Engineering*, SE-5(5):440 – 452, sept. 1979.

[32] R. E. Bryant. Simulation of packet communication architecture computer systems. Technical report, Cambridge, MA, USA, 1977.

[33] K. M. Chandy and J. Misra. Asynchronous distributed simulation via a sequence of parallel computations. *Commun. ACM*, 24(4):198–206. `doi:10.1145/358598.358613`.

[34] David M. Nicol. Principles of conservative parallel simulation. In *Proceedings of the 28th conference on Winter simulation*, WSC '96, pages 128–135, Washington, DC, USA, 1996. IEEE Computer Society. `doi:10.1145/256562.256591`.

[35] D. M. Nicol and P.F. Reynolds. Problem oriented protocol design. In *Proceedings of the 16th conference on Winter simulation*, WSC '84, pages 470–474, Piscataway, NJ, USA, 1984. IEEE Press.

[36] D. M. Nicol. Parallel discrete-event simulation of fcfs stochastic queuing networks. *SIGPLAN Not.*, 23(9):124–137, January 1988. `doi:10.1145/62116.62128`.

[37] Tropper C. Groselj, B. The time of next event algorithm. *Proceedings of the 1988 Conference on Parallel and Distributed Simulation*, pages 25–29.

[38] B. D. Lubachevsky. Efficient distributed event-driven simulations of multiple-loop networks. *Commun. ACM*, 32(1):111–123, January 1989. `doi:10.1145/63238.63247`.

[39] Sherman R. Chandy, K.M. The conditional-event approach to distributed simulation. *Proceedings of the 1989 Conference on Parallel and Distributed Simulation*, pages 93–99, 1989.

[40] D. Jefferson and H Sowizral. Fast concurrent simulation using the time warp mechanism. 1982.

[41] David R. Jefferson. Virtual time. *ACM Trans. Program. Lang. Syst.*, 7(3):404–425, July 1985.

[42] F. Bouchhima, M. Briere, G. Nicolescu, M. Abid, and E.M. Aboulhamid. A systemc/simulink co-simulation framework for continuous/discrete-events simulation. In *Proceedings of the 2006 IEEE International Behavioral Modeling and Simulation Workshop*, pages 1 –6, sept. 2006. `doi:10.1109/BMAS.2006.283461`.

[43] F. Mendoza, C. Kollner, J. Becker, and K.D. Muller-Glaser. An automated approach to systemc/simulink co-simulation. In *2011 22nd IEEE International Symposium on Rapid System Prototyping (RSP)*, pages 135 –141, may 2011. `doi:10.1109/RSP.2011.5929987`.

[44] M. Bombana and F. Bruschi. Systemc-vhdl co-simulation and synthesis in the hw domain. In *Design, Automation and Test in Europe Conference and Exhibition, 2003*, pages 101 – 105 suppl., 2003. `doi:10.1109/DATE.2003.1186679`.

[45] L. Benini, D. Bertozzi, D. Bruni, N. Drago, F. Fummi, and M. Poncino. Systemc cosimulation and emulation of multiprocessor soc designs. *Computer*, 36(4):53 – 59, april 2003. `doi:10.1109/MC.2003.1193229`.

[46] M. Dubois and E.M. Aboulhamid. Techniques to improve cosimulation and interoperability of heterogeneous models. In *ICECS 2005. 12th IEEE International Conference on Electronics, Circuits and Systems, 2005.*, pages 1 –4, dec. 2005. doi:10.1109/ICECS.2005.4633510.

[47] M. Trams. Conservative distributed discrete event simulation with systemc using explicit lookahead. *Digital Force White Paper*, February 2004. URL: http://www.digital-force.net.

[48] D.R. Cox. *RITSim: Distributed SystemC Simulation*. PhD thesis, Rochester Institute of Technology, August 2005.

[49] B. Chopard, P. Combes, and J. Zory. A conservative approach to systemc parallelization. In *Proceedings of the 6th international conference on Computational Science - Volume Part IV*, ICCS'06, pages 653–660, Berlin, Heidelberg, 2006. Springer-Verlag. URL: http://dx.doi.org/10.1007/11758549_89, doi:10.1007/11758549_89.

[50] P. Combes, E. Caron, F. Desprez, B. Chopard, and J. Zory. Relaxing synchronization in a parallel systemc kernel. In *ISPA '08. International Symposium on Parallel and Distributed Processing with Applications, 2008.*, pages 180–187, 2008. doi:10.1109/ISPA.2008.124.

[51] Christoph Roth, Oliver Sander, Matthias Kühnle, and Jürgen Becker. Hla-based simulation environment for distributed systemc simulation. In *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques*, SIMUTools '11, pages 108–114. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2011. URL: http://dl.acm.org/citation.cfm?id=2151054.2151078.

[52] Ieee standard for modeling and simulation (m amp;s) high level architecture (hla)– federate interface specification - redline. *IEEE Std 1516.1-2010 (Revision of IEEE Std 1516.1-2000) - Redline*, pages 1 –378, 18 2010. doi:10.1109/IEEESTD.2010.5954120.

[53] Michael C. McFarland, A.C. Parker, and R. Camposano. The high-level synthesis of digital systems. *Proceedings of the IEEE*, 78(2):301–318, Feb. doi:10.1109/5.52214.

[54] Edsger W. Dijkstra. Letters to the editor: go to statement considered harmful. *Commun. ACM*, 11(3):147–148, March 1968. doi:10.1145/362929.362947.

[55] Ieee standard computer dictionary. a compilation of ieee standard computer glossaries. *IEEE Std 610*, page 114, 1990.

[56] Christian Köllner, Nico Adler, and Klaus D. Müller-Glaser. System#: High-level synthesis of physical simulations for fpga-based real-time execution. In *Proceedings of the FPL, IEEE International Conference on Field-Programmable Logic and Applications*, pages 731–734, 2012.

[57] David Hlavac. Modellierung, simulation und analyse ereignisdiskreter hardware/software-systeme in c# 5.0. Diplomarbeit, Karlsruher Institut für Technologie, Institut für Technik der Informationsverarbeitung, 2013.