UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
BACHARELADO EM ENGENHARIA DE COMPUTAÇÃO

FELIPE DIENSTMANN MUSSE

SystemC-Unit: a unit testing framework for SystemC

Trabalho de graduação.

Trabalho realizado no Grenoble INP dentro do acordo de dupla diplomação UFRGS - Grenoble INP.

Orientadora brasileira:
Prof. Dra. Érika Cota

Orientador francês:
Prof. Dr. Matthieu Moy

Porto Alegre
2016

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Musse, Felipe Dienstman

SystemC-Unit: a unit testing framework for SystemC / Felipe Dienstmann Musse. – Porto Alegre: Engenharia de Computação da UFRGS, 2016.

50 f.: il.

Trabalho de conclusão (Bacharelado) – Universidade Federal do Rio Grande do Sul. Bacharelado em Engenharia de Computação, Porto Alegre, BR–RS, 2016. Orientadora brasileira: Érika Cota; Orientador francês: Matthieu Moy.

1. SystemC. 2. Teste unitário. 3. Modelização a nível transacional. 4. Teste de software. 5. Qualidade de software. I. Cota, Érika. II. Título.

RESUMO

A biblioteca de modelização SystemC é utilizada para construir modelos em software de sistemas de hardware complexos (tais como Systems on Chip), os quais são chamados protótipos virtuais. Na empresa STMicrolectronics, tais modelos são desenvolvidos com o auxílio de elementos reutilizáveis, os quais facilitam a representação de componentes comumente encontrado em diferentes sistemas de hardware. Garantir o bom funcionamento destes elementos é fundamental, uma vez que os modelos aos quais eles são integrados são utilizados em atividades importantes, tais como desenvolvimento de software embarcado e verificação funcional.

Este trabalho consiste no desenvolvimento de um framework de teste unitário para SystemC, o qual permite o teste destes elementos reutilizáveis. O contexto de desenvolvimento em que eles são utilizados é inicialmente analisado, assim como os problemas dos testes existentes. Em seguida, alguns frameworks de teste unitário de código livre são estudados para considerar sua aplicabilidade à solução destes problemas. Com base nos resultados destas análises, as características da solução proposta são definidas. Finalmente, o framework, o qual é denominado SystemC-Unit, é implementado, testado e validado.

Palavras-chave: SystemC, teste unitário, modelização a nível transacional, teste de software, qualidade de software.

Resumo estendido

Este é um resumo estendido em português para a Universidade Federal do Rio Grande do Sul do trabalho original que segue. O trabalho de conclusão original, em inglês, foi apresentado no Grenoble INP - Ensimag através do programa de dupla diplomação entre as duas universidades.

1 Introdução

1.1 Contexto

Um System on Chip (SoC) é um circuito integrado que contém, em um único chip, todos os componentes necessários à operação de um computador, tais como unidades centrais de processamento (CPUs), memórias, periféricos e interfaces. O desenvolvimento de um SoC envolve uma quantidade significativa de software embarcado, tais como firmware e drivers. No fluxo de projeto tradicional, este software apenas é desenvolvido após o design e fabricação do circuito. No entanto, a fim de manter competitividade e reduzir o *time-to-market* do chip, este software deve ser desenvolvido em paralelo com o hardware.

Uma solução para este problema é construir um modelo simples do SoC em software, o qual modeliza apenas o suficiente para desenvolver o software necessário. A técnica de modelização a nível transacional pode ser utilizada para obter tais modelos, os quais são chamados de **protótipos virtuais**. Por poderem ser escritos e simulados rapidamente, tais modelos são apropriados para desenvolver o software necessário antes da fabricação do circuito.

1.2 Exemplo de prototipagem virtual na indústria

Na **STMicroelectronics** (comumente chamada ST), uma multinacional franco-italiana fabricante de eletrônicos e semicondutores, protótipos virtuais são utilizados para antecipar o desenvolvimento do software embarcado para produtos como Systems on Chip. A equipe de Modelização a Nível de Sistema (SLM, de *System Level Modeling*) é a referência da empresa para as atividades relacionadas aos protótipos virtuais, os quais são construídos com **SystemC**, uma biblioteca de modelização para C++.

A fim de facilitar o desenvolvimento destes modelos, esta equipe desenvolve e mantém um conjunto de **elementos reutiliźaveis** para SystemC (também chamados de briques). Estes elementos representam entidades comumente encontradas em sistemas de hardware, tais como bancos de registradores e controladores de interrupções. Graças ao uso destes briques, o processo de escrever protótipos virtuais é acelerado de forma significativa; além disso, os modelos resultantes são mais homogêneos do que se cada equipe os tivesse desenvolvido do zero.

Garantir o funcionamento correto destes briques reutilizáveis é fundamental, uam vez que eles são integrados a modelos que são utilizados em atividades importantes, tal como o desenvolvimento de software embarcado. Um brique não-operacional pode causar atrasos na entrega de um produto ao mercado. Além disso, eles não são apenas utilizados pela equipe SLM, mas também por equipes de desenvolvimento de produtos da ST que desejam fazer uso de protótipos

virtuais em seus projetos.

1.3 Problema

Devido aos riscos de falha, uma certa quantidade de testes foi desenvolvida pra cada brique reutilizável. Geralmente, os testes relativos a uma funcionalidade de um brique são escritos por quem a implementou. Os testes existentes foram portanto desenvolvidos e modificados por diferentes pessoas, e isto foi feito de maneira *ad-hoc*. Como resultado, eles apresentam alguns problemas:

- Há muita repetição de código (tanto entre os testes de um único brique como entre os testes de diferentes briques), o que os faz desnecessariamente extensos e difíceis de entender, manter e modificar;
- Os testes não descrevem claramente quais aspectos de uma funcionalidade estão sendo verificados; como resultado, os testes recorrentemente se sobrepõem, enquanto outros aspectos não são testados;
- Testes similares são estruturados diferentemente, o que complica tanto o seu entendimento como a sua manutenção:
  - Operações comuns, tal como a comparação entre valores esperados e efetivos, são efetuadas assistematicamente;
  - Os resultados dos testes são heterogêneos e desorganizados; isto torna difícil a sua análise tanto em caso de regressão como de não-regressão.

1.4 Objetivos

A importância de testar os briques reutilizáveis em SystemC desenvolvidos pela equipe SLM foi estabelecida. Além disso, como mostrado acima, os testes existentes apresentam problemas significativos que dificultam sua compreensão, manutenção e escrita. O objetivo principal deste projeto é portanto resolver estas questões a fim de garantir a verificação correta do funcionamento destes briques, aumentando assim a produtividade em atividades de teste.

A solução proposta é desenvolver um framework de teste que:

- Proponha serviços simples que facilitem o trabalho dos desenvolvedores e os guie em direção a testes homogêneos e de qualidade: comparação de valores esperados, relatórios de erros, estrutura padrão de testes, e assim por diante;
- Forneça resultados de testes homogêneos e relevantes de forma a facilitar a sua análise para fins de regressão e não-regressão;

- Integre-se facilmente ao ambiente de desenvolvimento de protótipos virtuais em SystemC.

2 Visão geral do trabalho

Inicialmente, identificam-se requisitos que devem ser respeitados para o desenvolvimento da solução. Alguns deles são inerentes à utilização da biblioteca SystemC, enquanto outros são devidos ao ambiente de desenvolvimento da equipe SLM. Requisitos adicionais são extraídos através da análise dos problemas nos testes existentes de um exemplo de brique reutilizável.

Em seguida, alguns frameworks de teste de código aberto são estudados a fim de verificar sua aplicabilidade ao problema considerado e de extrair novos requisitos. Frameworks correspondendo a duas técnicas de testes são considerados: desenvolvimento guiado por compartamento (BDD, de *behavior driven development*), uma técnica relativamente moderna na qual os testes são baseados na descrição textual das funcionalidades do software, e teste unitário, uma abordagem clássica para o teste de software. Baseado nestas análises, estabelece-se uma lista definitiva de requisistos a serem considerados para o desenvolvimento do framework de teste. Além disso, conclui-se que os frameworks estudados não são adaptados para o teste de briques SystemC. Decide-se por desenvolver um framework do zero.

Por fim, propõe-se uma solução para os problemas apresentados, a qual é chamada **SystemC-Unit**. Primeiramente, estabelece-se uma lista de funcionalidades a serem fornecidas de forma a satisfazer os requisitos estabelecidos. Em seguida, o framework é implementado de forma incremental. Sua implementação também é testada a fim de verificar sua corretude. Por fim, exercícios de validação são efetuados, os quais permitem a identificação de funcionalidades ausentes e a verificação de que a solução atende às necessidades de seus usuários.

3 Resultados

As funcionalidades consideradas necessárias para a resolução do problema em questão foram integralmente implementadas e testadas. A implementação da solução proposta tem cerca de 4 mil linhas de código, bem menos do que os frameworks de código livre estudados. Atividades de validação permitiram constatar uma redução média de 21% no tamanho (em linhas de código) dos testes escritos com SystemC-Unit em comparação aos testes prévios. Uma primeira versão oficial do framework foi entregue à equipe SLM e está sendo atualmente utilizada para o teste de briques reutilizáveis.

Grenoble INP – ENSIMAG
École Nationale Supérieure d'Informatique et de Mathématiques Appliquées

End-of-Studies Project – Final Report

Undertaken at STMicroelectronics

SystemC-Unit: a unit testing framework for SystemC

Felipe DIENSTMANN MUSSE
3^{rd} Year – Embedded Systems and Software Option

February 15^{th} 2016 – July 15^{th} 2016

STMicroelectronics
12 rue Jules Horowitz
BP 217
38019 Grenoble Cedex

Internship supervisor
Jérôme CORNET
School mentor
Matthieu MOY

Abstract

The SystemC modelization library is used to build software models of complex hardware systems (such as Systems on Chip), which are called virtual prototypes. At STMicroelectronics, these models are developed with the aid of reusable elements which facilitate the representation of components commonly found in different hardware systems. Assuring the correct behavior of these elements is paramount, as models to which they are integrated are used in important activities, such as embedded software development and functional verification.

This work consists in the development of a unit testing framework for SystemC which allows the testing of these reusable elements. The developmment context in which they are used is initially analyzed, as well as the problems in existing tests. Next, some unit testing open-source frameworks are studied to consider their applicability to the solution of these problems. Based on the results of these analyses, the features of the proposed solution are defined. Finally, the framework, which is called SystemC-Unit, is implemented, tested and validated.

Keywords: SystemC, unit testing, transaction-level modeling, software testing, software quality.

Acknowledgments

First of all, I would like to thank STMicroelectronics for the opportunity to undertake this internship. My sincere thanks towards everybody with whom I worked during these five months for receiving me so well. I am very grateful to my team manager Laurent Maillet-Contoz, whose advice greatly helped me during this internship. I would also like to thank my school mentor Matthieu Moy for promptly answering my questions and for having prepared me so well for this project with his Transaction-Level Modeling courses. I am also very thankful to my second supervisor, Érika Cota, who helped to greatly improve this text with her many remarks and suggestions.

I am especially thankful to my internship supervisor Jérôme Cornet. His guidance was the main reason for the success of this project. Were it not him for, this internship would not have been half as interesting and challenging as it was. I deeply thank him for his patient explanations when I did not easily accept his instructions and for sharing all his knowledge with me.

My sincere thanks towards both my universities, ENSIMAG (École nationale supérieure d'informatique et de mathématiques appliquées) and UFRGS (Universidade Federal do Rio Grande do Sul). I am very thankful to CAPES (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior) and to my exchange coordinator Claudio Geyer for providing me the opportunity to study in France and to undertake this internship.

Finally, I would like to fondly thank my parents – the older I get, the more I realize that everything I ever achieve will be thanks to them.

Contents

List of Figures

1. Introduction

1.1 Context

1.1.1 SoC and software development

A **System on Chip (SoC)** is an integrated circuit which contains, in a single chip, all the components required for the operation of a computer, such as central processing units (CPUs), memories, peripherals and interfaces. These elements commonly take the form of **IP (intellectual property) cores**, which are blocks of logic or data whose objective is to be reused for the development of different chips.

SoCs (and other electronic circuits) are commonly designed with **hardware description languages (HDL)**. They allow to describe the structure and behavior of a circuit with different levels of abstraction and the resulting models may be analysed and simulated. The two major hardware description languages are VHDL and Verilog. The most common way to represent a digital circuit with an HDL is with a **register-transfer level (RTL)** of abstraction, that is, in terms of the flow of data between registers and the operations performed on this data. After a RTL model has been developed, lower level descriptions may be derived from it, which are then used to manufacture the circuit.

SoCs involve a significant amount of software development, such as firmware and drivers. In the regular SoC design flow, this software is only developed after the circuit has been fully designed and manufactured. However, in order to maintain competitivity by reducing the circuit's **time-to-market**, the software should be developed in parallel with the hardware. This is complicated to achieve using the circuit's RTL representation: since the level of abstraction is rather low, simulating the hardware's operation with the software takes too much time. Moreover, ideally it should not be necessary to wait for the RTL description to be finished to start the software development.

A solution is to build a simpler model of the circuit with a higher level of abstraction, only modeling what is necessary to develop the required software. This is exactly the objective of the **transaction-level modeling (TLM)** approach. With this technique, models are commonly built using a general-purpose programming language. The idea is to separate the details of communication among the circuit's modules from the actual information exchange (transactions). Each of the hardware's subcomponents may hence be freely developed with the considered programming language without considering the circuit's clock (**cycle-less** model) as long as it respects the defined interface. As a result, when compared to RTL models, TLM ones are simpler to write and simulate faster, and therefore are suitable to develop the required software before the circuit is fabricated. For the same reasons, they are also useful for other activities, such as functional verification, architectural exploration and performance modeling.

In order to model hardware systems with a TLM level of abstraction, one may use a special-

ized software library. An example is **SystemC** [1], a standardized C++ library which is presented in Section 2.1 in more details.

1.1.2 Model terminology

TLM models which mainly aim to anticipate the embedded software development are generally called **virtual prototypes** (prototype, as it is an early model of a product that has not been fully designed; virtual, as it is a software model, not a physical one). A complete model of a circuit is composed of blocks[1] which are differently named according to their level in the system's hierarchy. Making the distinction between them is paramount to understand this document.

Throughout this text, the terminology illustrated in Figure 1.1 is used:

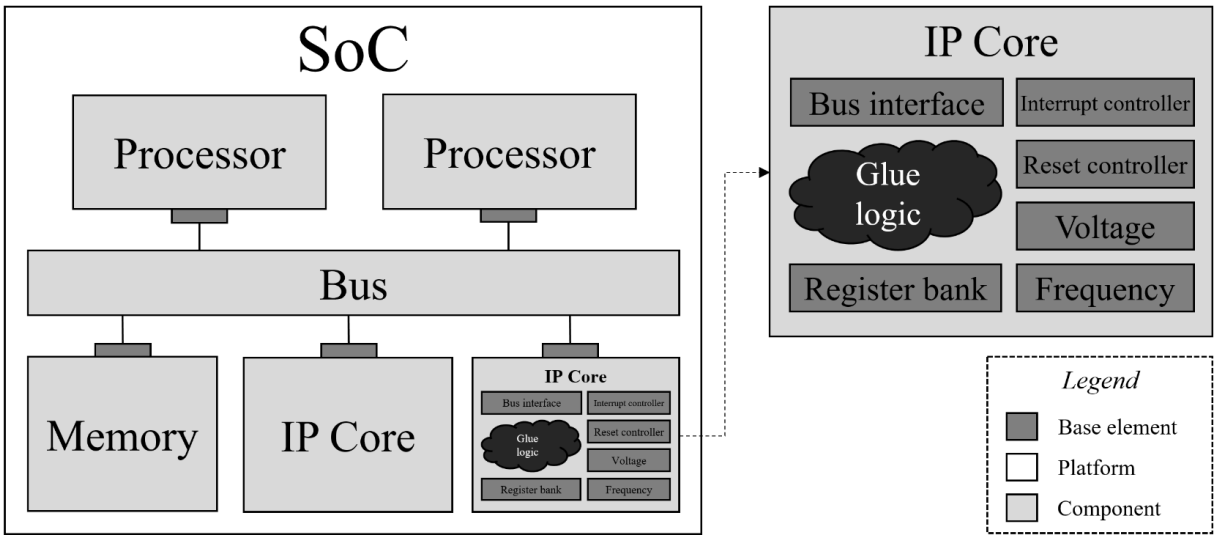

Figure 1.1 – Terminology for different entities of a virtual prototype

- a **virtual prototype** or **platform** corresponds to a complete and independent hardware system, such as a SoC;
- a **component** represents a rather complex and independent hardware subblock present in a platform, such as an IP core or a memory;
- a **base element** or **brick** is a "simple" block which represents an entity that is commonly found in more complex components and that are developed with reuse in mind. For instance, IP cores normally contain many registers, so a brick that models a register bank may be reused for the development of different components. A component is therefore composed of many reusable bricks glued together by the code which defines its specific behavior.

[1] A **block** or **module** simply refers to an entity in the hardware system's hierarchy, without regard to complexity.

1.1.3 Example of virtual prototyping in the industry

Within STMicroelectronics [2] (commonly called ST), a French-Italian multinational electronics and semiconductor manufacturer, virtual prototypes are used to anticipate the embedded software development for products such as Systems on Chip. The System Level Modeling (SLM) team is ST's main reference for activities related to virtual prototypes, which are commonly built with the SystemC library. The team's main responsibilities, illustrated by Figure 1.2, are:

- **Methodology:** providing training courses and support to other ST's divisions which use virtual prototypes to develop their products;
- **Direct model development:** when a group needs to develop a virtual prototype for one of its products but does not possess the required expertise, the SLM team may directly develop the model for them;
- **Standardisation:** the team is actively involved in the development of both SystemC and IP-XACT [3] (a specification format for electronic components) standards;
- **Reusable elements:** in order to facilitate the development of virtual prototypes with SystemC, the team develops and maintains a set of reusable SystemC bricks.

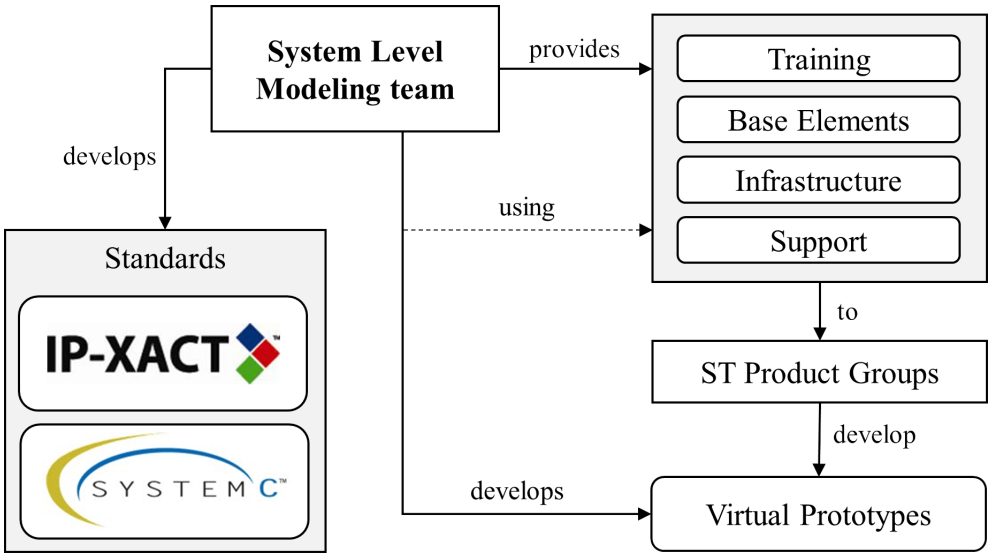

Figure 1.2 – System Level Modeling team activities

These reusable bricks are used internally by the SLM team and also provided to ST's product development divisions. Examples of these base elements are register banks, wires and voltage and frequency elements. As a result, the process of writing a virtual prototype is significantly accelerated. Moreover, the resulting models are more homogeneous than if each team had developed them from scratch.

Even though these base elements may seem simple at first glance, the fact that platforms will interact with them differently generates a need for them to be highly configurable. Moreover, when a product team requires a feature that is not present in a base element, it is requested to the SLM team, which subsequently implements it and releases the updated brick to all of ST. As a consequence, these elements are relatively complex.

Since these base elements are integrated into many components and platforms which are used in important activities such as embedded software development and functional verification, assuring their correct behavior is paramount. For instance, a platform which contains a bugged brick may not properly model the actual circuit; the software that has been developed with it may work with the virtual prototype, but not with the real system. It would be very hard to find the problem since, in the platform's developers vision, the circuit's specification was correctly translated into a virtual prototype – the problem is not in their model, but rather in the base elements of which they made use.

A non-operational brick may hence lead to very time-consuming and expensive troubleshooting. Moreover, frequent bugs would have an impact on the team's reputation, as these elements are provided to other divisions. Testing is therefore an essential activity within the SLM team – and not a trivial one, since the implementations of these bricks are certainly substantial.

1.2 Problem to solve

Due to the involved risks of failure, a number of tests is designed for each reusable bricks. Generally, the tests concerning a feature are written by whom implemented it. The existing tests have therefore been developed and modified by different people and this has been done in an *ad hoc* fashion. As a result, they are rather problematic:

- There is a lot of code repetition (between both tests for a single brick and tests for different bricks), which makes them unnecessarily large and difficult to understand, maintain and modify;
- The tests do not clearly describe which feature's aspects are being verified; as a result, they recurrently superpose themselves, whereas other functionalities are left untested;
- Similar tests are rather differently structured, and this complicates both their understanding and maintenance:
  - Common operations, such as comparisons between expected and actual values, are unsystematically performed;
  - The reporting of tests results is heterogeneous and disorganized: this makes it hard to analyze the tests results for both regression and non-regression purposes.

1.3 Objectives

The importance of testing the SystemC reusable bricks developed by the System Level Modeling team has been established. Moreover, as shown above, the existing tests present some significant problems which encumber their comprehension, maintenance and writing. The main objective of this project is hence to solve these issues in order to assure the proper verification of these brick's behavior and to increase productivity in testing related activities.

The proposed solution is to develop a test framework which:

- Proposes simple services to facilitate the developers' work and steer them towards quality and homogeneous tests: expected value comparison, error reporting, standardized test structure, and so on;
- Provides homogeneous and relevant reporting of tests results in order to facilitate both regression and non-regression analysis;
- Easily integrates itself to the development environment of SystemC virtual prototypes.

Since the System Level Modeling team does not suffer from Not-Invented-Here syndrome[2], an auxiliary objective is to analyze the state of art of testing practices outside of ST. The suitability of a rather new methodology called Behavior Driven Development (BDD) for the testing of the SystemC bricks is evaluated. Moreover, the idea is not to blindly implement the framework from scratch, but rather to first analyze existing open-source testing libraries which may provide elements or ideas to facilitate its development. Both these analyses are carried out considering a possible extension of the framework in order to allow the testing of SystemC components (more complex blocks which are built upon reusable bricks).

1.4 Manuscript Organization

The text is organized as follows:

- Chapter 2 presents the technical background and concepts which are required for the understanding of this text.
- Next, Chapter 3 details the problems in existing tests and identifies the requirements to be respected by the proposed solution.
- Chapter 4 then analyses existing open-source test frameworks in order to verify their applicability to the presented problem.
- Afterwards, Chapter 5 presents the proposed solution and gives an overview of its development, validation and testing.
- Finally, Chapter 6 concludes with the results and perspectives of this work.

[2]Not invented here syndrome [4] is the tendency of both individual developers and entire organizations to reject suitable external solutions to software development problems in favor of internally developed solutions.

2. Background

This chapter reviews the main concepts required for the development of this work. First, some details of the SystemC virtual prototyping library are presented. Afterwards, the needed software testing concepts are shown and explained.

2.1 SystemC

SystemC [1] is a standardized C++ library which allows the modeling of both hardware and software of electronic systems at multiple levels of abstraction. It provides classes to represent, among others, the decomposition of a system into modules, the connectivity and communication between those modules, the passing of simulated time, the synchronization of concurrent modules and the data types commonly found in digital hardware. The library also includes a simulation kernel and offers TLM capabilities to abstract communication between modules. SystemC is defined and promoted by the **Accellera Systems Initiative (ASI)** [5], an organization of which most major EDA (electronic design automation) software vendors and users are participants (including ST). The library is defined by the IEEE (Institute of Electrical and Electronics Engineers) standard 1666-2011 and an open-source reference implementation is maintained by ASI.

A SystemC model is composed of different modules, which are instances of the `sc_module` class. These modules may contain **processes**, which are actually member functions that run concurrently and represent the circuit's parallel behavior. The model's execution occurs within the `sc_main()` function, which is defined by the user and called by SystemC's simulation kernel. A minimalistic example of such a function is shown in Listing 2.1. This execution consists of two phases:

Listing 2.1 – Minimalistic example of `sc_main()` function

```
1  int sc_main(int argc, char ** argv)
2  {
3      Cpu cpu("CPU");
4      Peripheral periph("Peripheral");
5
6      sc_signal<bool> irq;
7      periph.irq_out(irq);
8      cpu.irq_in(irq);
9
10     sc_start();
11     return 0;
12 }
```

- The **construction** is the definition of the model's architecture. Within the `sc_main()` function, the model's modules are instantiated (lines 3 and 4) and connected together (lines

6 to 8).

- The **simulation** starts with the call to `sc_start()` (line 10). No new modules may be added at this point. The control is given to SystemC's scheduler, which controls the passing of time and calls the modules' eligible processes one at a time to simulate the circuit's behavior. Processes may suspend themselves for a given amount of time with a call to `wait()` or by waiting for an event to be notified by another element of the model.

Figure 2.1 shows the interaction between the SystemC kernel and the `sc_main()` user-defined function and modules for the execution of a model.

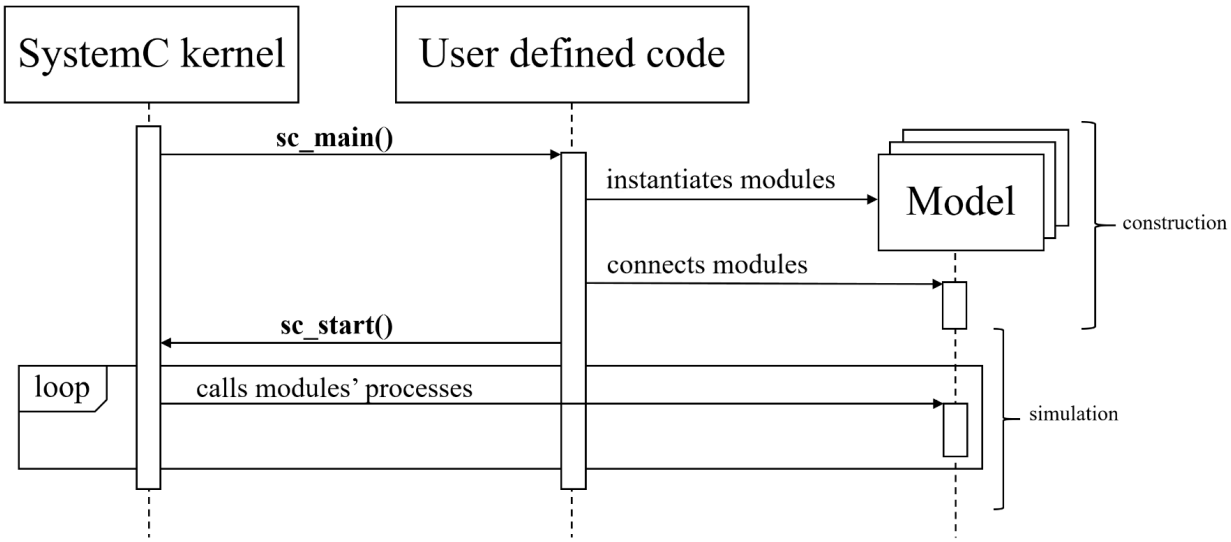

Figure 2.1 – Overview of the execution of a SystemC model

2.2 Software testing concepts

In simple terms, the objective of testing a software is verifying that it is correctly implemented, that is, whether there are problems in the expected functionalities. If it does not, another objective is finding what the problems are. Software tests are commonly divided in three different levels [6]:

- **Unit tests:** small units of the program are individually tested. In object-oriented programming, these units are commonly classes or methods, and in procedural programming, functions. Tests commonly focus on implementation faults, such as programming and logic errors.

- **Integration tests:** individual units are integrated to create composed elements and tested as a group. Tests in this level focus on the communication between modules.

- **System tests:** all the elements of the complete system are integrated and the software is tested as a whole. Tests in this level commonly focus on the overall functionality provided by the modules altogether.

Unit tests are automated through the use of unit testing frameworks, which provide support for developing and executing them. An example is the xUnit [7] family, a collection of unit testing frameworks that support different programming languages and share a similar test architecture. These frameworks provide generic test classes which are extended by the user in order to write tests. The tool may then execute all tests and indicate what the results were.

When using unit testing frameworks, tests take the form of what is called a test case in the xUnit architecture. A test case is a sequence of code which exercises a certain behavior of the element under test with a given stimuli and verifies its correct operation. A test case is composed of three parts:

1. **Setup:** the element under test is initialized to a known state, i.e., the preconditions of the test case. The setup code may also be called test fixture [8]. This setup code may be put directly in the test method (in-line setup) or separately so that it may be used by different test cases to avoid code repetition (delegated setup if invoked explicitly by the test code or implicitly by the framework);

2. **Exercise:** the test code interacts with the element under test, that is, the object or method to be tested is called (possibly with specific test data or inputs);

3. **Verification:** the results of the calls are compared to the expected ones. This is usually done through the use of assertions, which are functions or macros provided by the test framework to express logical conditions that are true if the element under test works properly.

Test cases are called independent if different features may be tested in isolation (i.e., they do not depend on results of previous test cases). Unit testing frameworks are commonly structured so that this is the case. For instance, different test cases for an object will work on separate instances of the class under test, which are usually instantiated at the start of the test case and destructed at its end.

A test suite is simply a set of test cases. Each test suite commonly tests a single feature of the element under test for different conditions (i.e. stimuli). This concept is used by unit testing frameworks in order to provide an organized manner of separating tests.

Tests may also be categorized in two types:

- In **white-box testing**, also known as structural or glass box testing, tests are written with knowledge of the internal logic or implementation of the element under test. White-box tests are commonly written by the developer of the software. This approach is commonly used in lower levels of testing, such as unit and integration tests.

- In **black-box testing**, also known as functional or behavioural testing, tests are written with no knowledge of how the element under test works internally. These tests are usually

written by dedicated testers. Black-box testing can be used in any level of testing, from system tests to unit ones.

The tests regarding the reusable SystemC bricks fall into the unit test level, as we are testing the object and its methods. Since they are written by their developers with knowledge of their implementation, they are considered white-box tests. A unit testing framework for SystemC does not currently exist.

3. Requirements

In order to solve the problems presented in Section 1.2, the initial approach is to identify the requirements and constraints to be respected for the development of the solution. Some constraints are already known as they are imposed by the SLM team's development context. Additional requirements are identified through the analysis of the existing tests of an example base element.

3.1 Initial constraints

Some constraints that should be respected by the solution to be developed are inherent to the usage of SystemC and to the established development environment of the SLM team. They are presented below:

- **Inversion of control:** SystemC follows an inversion of control design: unlike a regular reusable library, which is called by its user, it is SystemC that calls the code defined by the user (the `sc_main()` function). The solution do be developed must work properly within this context, that is, the framework must cooperate with SystemC or take it into account in order to execute the tests.

- **Lifetime of SystemC objects:** SystemC related objects (e.g. `sc_module`) must be instantiated during SystemC's construction phase, that is, within the `sc_main()` function and before the start of simulation (call to `sc_start()`). Moreover, after one such object has been instantiated, it may only be destructed after the end of simulation. This impacts the independence of test cases as the elements under test may not be instantiated and destructed at will. as When handling the instantiation and destruction of the objects required by the tests, the solution should take these constraints into account.

- **Testing environment:** the SLM team supports a rather large set of environments for the development of virtual prototypes: different operating systems (Windows and several Linux distributions, both 32 and 64 bit versions), compilers (GCC, Clang, Visual C++) and SystemC implementations. The provided base elements must work correctly with all of the supported environments. In order to verify that they do so, the team uses a tool which compiles and executes the tests of a base element for all possible execution configurations. Since the objective is to use the test framework to write these tests, the solution must be portable and simple to compile so that it works properly with all supported environments. Moreover, these constraints must also be respected by any external solution considered for direct use for the testing of base elements.

3.2 Additional Constraints

In order to better understand the usage and the subjacent test requirements of the base elements that the framework to be developed should test, we analyse a representative and significant example: the register bank brick. Complex systems contain hundreds of registers, and modeling them individually from scratch is very time-consuming, leads to unsystematic messages and makes it harder to correct bugs and to update the related communication protocols. The register bank base element has been created to solve these various problems which arise when modeling registers of a SystemC block.

This brick's tests amount to 56 test suites with more than 27,000 lines of code. Analysing them allows not only to establish the tester needs in terms of operations and structure but also to further understand the problems which motivate the development of the test framework. Identifying the performed operations which repeat themselves between test suites makes it possible to evaluate which ones are more common and hence more important to provide.

These repeated patterns may be grouped into generic categories which represent aspects most likely to be required for the testing of other base elements. Based on the analysis of the register bank tests, we extracted 8 categories that are presented below and correspond to necessary features and important questions to be considered for the development of the test framework.

Assertions

Verifying that a number of a base element's values (commonly from the return of methods or functions) are as expected is the main manner of assuring its correctness. In current tests, assertions are carried out very unsystematically, and this complicates both their reading and writing.

In order to better understand this problem, consider the illustrative and simplified example shown in Listing 3.1. This code exemplifies a test case for the register bank. A `register_bank` object is assumed to be instantiated. Initially, stimuli (lines 2 to 4) are applied to it – values are written to three of its registers with the `write()` method. Four different manners of performing the required assertions (verifying that the actual return values of the `read()` method correspond to the expected ones) are then shown, all of which are problematic:

- **Option 1:** the actual return values of the `read()` methods are simply printed to the default output stream. No actual comparison is carried out in the code; the only possible verification is to manually read the tests output and then check that the printed values correspond to the expected ones.
- **Option 2:** the code actually compares the actual values to the expected ones, but failures have to be manually found in the tests output and there is no information about which of the `read()` calls yielded an error.
- **Option 3:** even though the comparisons are carried out separately, it presents the same problems of option 2.

- **Option 4:** each of the comparisons is performed separately and a rather relevant message is shown in case of failure. However, developers are still obliged to write the messages and the comparisons themselves.

Listing 3.1 – Examples of problems found for assertions in existing register bank tests

```
// stimuli application
register_bank.write(0x0, 0xA);
register_bank.write(0x4, 0xB);
register_bank.write(0x8, 0xC);

// assertions: option 1
std::cout << register_bank.read(0x0) << std::endl;
std::cout << register_bank.read(0x4) << std::endl;
std::cout << register_bank.read(0x8) << std::endl;

// assertions: option 2
if (register_bank.read(0x0) != 0xA ||
    register_bank.read(0x4) != 0xB ||
    register_bank.read(0x8) != 0xC)
        std::cerr << "error with read()!" << std::endl;

// assertions: option 3
bool passed = true;

if (register_bank.read(0x0) != 0xA)
    passed = false;
if (register_bank.read(0x4) != 0xB)
    passed = false;
if (register_bank.read(0x8) != 0xC)
    passed = false;

if (!passed)
    std::cerr << "error with read()!" << std::endl;

// assertions: option 4
uint32_t val;

if (read_val = register_bank.read(0x0) != 0xA)
    std::cerr << "reading 0x0: expected 0xA, got " << read_val << std::endl;

if (read_val = register_bank.read(0x4) != 0xB)
    std::cerr << "reading 0x4: expected 0xB, got " << read_val << std::endl;

if (read_val = register_bank.read(0x8) != 0xC)
    std::cerr << "reading 0x8: expected 0xC, got " << read_val << std::endl;
```

These problems are amplified when considering a large set of tests, since it is very impractical to write all comparisons and messages, to verify that there have been no errors and to find the relevant information for a given test among the execution's output. The test framework therefore

should provide the user a simple interface that facilitates the performing of asssertions. It should do so for all the necessary data types (not only for the primitive ones but also for the model's objects). Moreover, it is important to allow the tester to specify whether the failure of an assertion is fatal, that is, if the current test case should be aborted or continued if a value is not as expected.

Message assertions

A specific type of assertion which should also be supported is that of expecting messages from the model (for instance, an error message when an invalid operation is carried out). Within the SLM team, all SystemC models make use of the `tlm_message` development kit, which provides homogeneous instrumentation capabilities, such as info, error, warning and debug messages. The framework needs to communicate with it in order to obtain the required information about the messages generated by the tested objects.

Tests structure

Figure 3.1 shows a simplified version of the structure of current tests. For most of the test suites (which correspond to SystemC modules, as detailed below), all their test cases are present in a single method (or worse, in the module's constructor). As a result, different test cases are mixed, which makes it harder to maintain them; moreover, it is not possible to execute a given test separately. The test suites are executed by instantiating all the test suite modules and by calling their test methods (if the tests are not in the constructor). It is therefore also not possible to easily execute test suites independently. The test framework should solve these issues by providing a clear separation between tests and by allowing to choose which ones should be carried out.

```
int sc_main() {
  // tests are manually instantiated
  TestSuite1 testSuite1;
  TestSuite2 testSuite2; // constructor executes tests

  testSuite1.test(); // tests are manually called

  sc_start();
  return 0;
}
```

```
class TestSuite1 : public sc_module {
  // objects under test are mixed
  RegisterBank regBank1;
  RegisterBank regBank2;

  void test() {
    // test cases are mixed:
    // test case with regBank1…
    // test case with regBank2…
  }
};
```

```
class TestSuite2 : public sc_module {
  // objects under test are mixed
  RegisterBank regBank1;
  RegisterBank regBank2;

  TestSuite2() {
    // test cases are mixed:
    // test case with regBank1…
    // test case with regBank2…
  }
};
```

Figure 3.1 – Simplified structure of current tests

Instantiation of the module under test

When testing an object, it is rather reasonable to say that one or more instances of it will be required. The particularity here is that the register bank and most other base elements must be instantiated within a SystemC module. The current solution is to have one SystemC module for each test suite. This modules contains the required number of objects to test as attributes. As a result, the objects tested by each of the test cases for a given feature are mixed, also impacting the independence of test cases. The test framework should provide a simple way of declaring the object under test and of instantiating it for each of the test cases.

Construction and simulation

The correct operation of most aspects of a base element may be verified during SystemC's construction phase (see Section 2.1). However, some features may only be tested during simulation, such as timing related ones. For these tests, verifications must be carried out within a process of a SystemC module so that the scheduler may call it during simulation. In the existing tests, this is handled within `sc_main()` as follows (as shown in Figure 3.2):

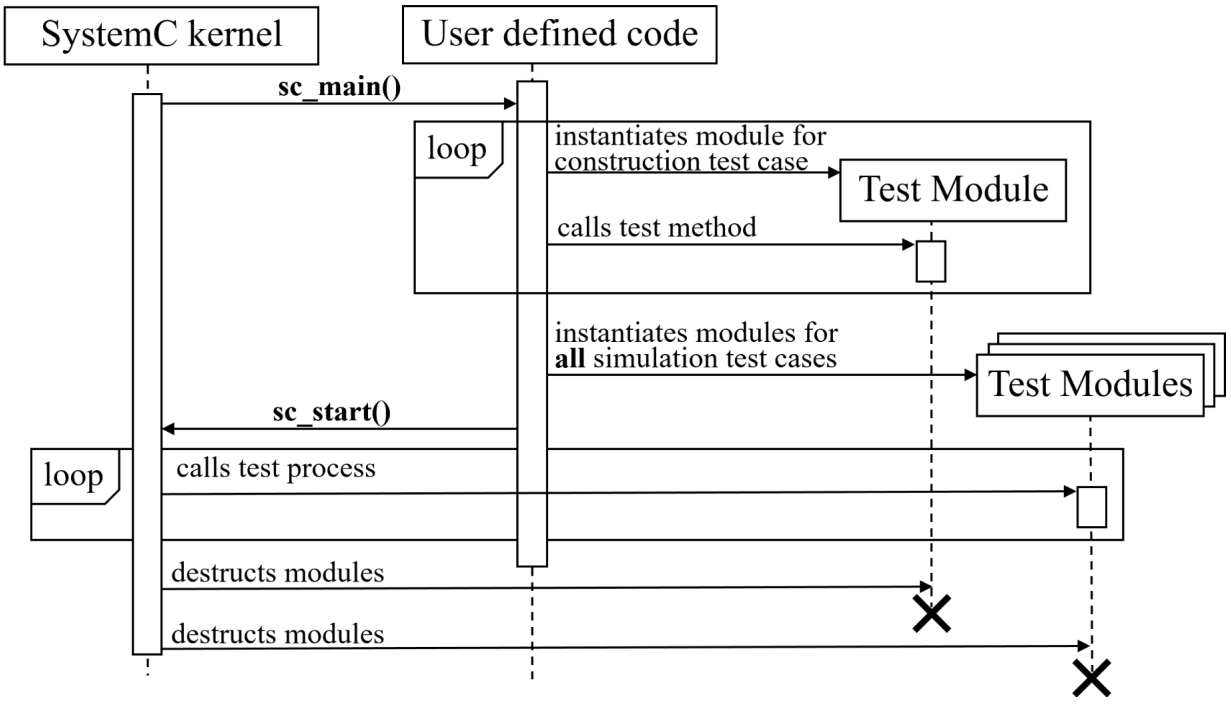

Figure 3.2 – Construction and destruction of test modules

1. The SystemC modules corresponding to all test suites are instantiated;
2. The test methods of the modules which correspond to features that should be tested in construction time are called;

3. Simulation is started and SystemC's scheduler calls the test processes of the modules corresponding to features that should be tested in simulation time;

4. After the end of simulation, all modules are destructed.

This method respects the lifetime constraints of SystemC objects. However, it is not achieved systematically, that is, the instantiation and destruction of modules and the calls to their methods are manually carried out. In order to avoid errors, this behavior should be handled by the test framework, and not by the actual tests. The solution should provide a manner of identifying which test should be executed during construction or simulation. Moreover, it should handle the instantiation and destruction of SystemC objects accordingly.

Temporal isolation of simulation tests

When simulation starts, all processes are eligible and the SystemC kernel executes them concurrently. Therefore, processes corresponding to different simulation tests may interfere with each other if they are not executed in temporal isolation. In current tests, this problem is solved with a parameter in the constructor of the modules containing processes which correspond to simulation tests. This parameter indicates at which time their processes should start, as exemplified in Listing 3.2. At the beginning of simulation, when a process is scheduled by SystemC's kernel, it calls SystemC's `sc_wait()` function with the argument received by its module's constructor. This call makes it sleep for the specified duration. When the waiting time is over, the process will be eligible again, and the simulation kernel will execute it (second loop in Figure 3.2).

Listing 3.2 – Example of how temporal isolation is achieved for existing simulation tests

```
1  int sc_main(int argc, char ** argv)
2  {
3      // TestSuite1 processes executed after 1 second of simulated time
4      TestSuite1 testSuite1("Test1", sc_time(1, SC_SEC));
5
6      // TestSuite2 processes executed after 2 seconds of simulated time
7      TestSuite2 testSuite1("Test2", sc_time(2, SC_SEC));
8
9      sc_start();
10     return 0;
11 }
```

In this code, this adjustment is manually done in the test code and is hence unpractical and error-prone. For example, it is rather easy to underestimate the required time for a given test, and tests would then overlap. The scheduling of simulation tests should therefore be handled transparently by the test framework so that they are executed in temporal isolation.

Compound and shared structures

Some aspects of a base element may not be verified within a single module; they require different modules to interact with each other (commonly during simulation). For example, in some of

the register bank tests, a separate modules needs to interact with the module which actually contains the register bank. The test framework should also cover this possibility and provide its users a way of describing the required structures for their tests. Moreover, different test cases may require the same structure, and in most cases, each of them requires a separate instance. However, for some structures needed by multiple tests cases, it is too expensive (memory-wise, for example) to instantiate them many times. The test cases must share a single instance of it and reset it to the inital condition after the execution of the required operations so that the remaining test cases may start from a known state. The framework should therefore also provide a manner of sharing the same instance of such a structure among many test cases.

Reporting of results

The tests' results take the form of textual information which describes which tests were carried out and which succeeded. Clear and well-organized test reports are paramount to verify that a modification has not introduced regressions and to find the problem if it did. In the existing tests, the reporting of assertion results is not homogeneous. Moreover, even though the assertion results from different test cases and test suites are separated by headers which indicate which test is being executed, this output is constructed manually in the test code. Since this code repetition could lead to errors, the reporting of results should be a responsibility of the test framework. It should generate adequate result messages for the assertions and provide a simple way of describing what is being verified by each test case. This information should then be assembled in an adequate and well-structured format. This could be achieved with the aid of the `tlm_message` instrumentation library.

4. Analysis of existing test frameworks

Some open-source testing frameworks are analyzed in order to not only verify their applicability regarding the already defined requirements but also to identify new ones. Frameworks corresponding to two different testing methods are considered: behavior-driven development[1], a rather modern technique in which tests are based on a textual description of the software's features, and unit testing, a classical approach for the test of software. Based on these analyses, a decision about whether to directly use these solutions is made. Moreover, the definitive list of the requirements and constraints to be taken into account for the development of the test framework is established.

4.1 Behavior Driven Development framework

Initially, the methodology itself is revised. Next, an existing open-source BDD framework is chosen and thoroughly studied in order to well understand whether it satisfies the previously established requirements. For the same reason, some existing tests for the register bank brick are rewritten with the chosen BDD framework. As a result of this analysis, we show that some requirements could not be satisfied. An attempt to modify the framework in order to solve these issues is presented and analyzed. Finally, based on the results of this study, a choice of whether to proceed with the use of BDD for the testing of SystemC base elements is made.

4.1.1 The BDD methodology

Behavior-driven development is a software development methodology whose main principle is to describe the software's expected behavior before its implementation. This description is written in a semi-formal format which uses natural language constructs and may therefore be shared between developers and non-technical personnel. The most commonly used format is a domain specific language which is very similar to user stories [9]. This behavioral description serves not only as specification but also as tests definition [10]. The tests are carried out with the aid of a BDD support tool which takes this description as input and executes the code associated with each sentence or clause (this association is defined by developers for each project). Since the tests are written before implementation, BDD is commonly considered an extension of test-driven development (TDD)[2].

[1] Even though this methodology is usually employed in levels of testing which are higher than that of the considered SystemC bricks (e.g. system testing), it is studied nonetheless due to the complexity of the tested objects.

[2] Test-driven development is a methodology which essentially states that for each feature of a software unit, a software developer must first define a test set for the feature, then implement it, and finally verify that the implementation makes the tests succeed.

The BDD methodology concurs with the test factorization effort addressed by this project. A simple and homogeneous way of specifying the SystemC bricks' behavior and a method of directly obtaining their tests from this description would certainly increase the quality of tests. Even though the software which would be tested by the framework to be developed already exists, a test framework based on BDD could still be suitable for it. Moreover, the framework could be used with total compliance to BDD for the specification of new features for a base element before their implementations.

4.1.2 The Cucumber framework

The analyzed behavior-driven development test framework is called Cucumber [11]. It was selected between other options as it is one of the most widely used BDD frameworks, with more than 100,000 downloads per week and more than a thousand contributors. Moreover, even though Cucumber was originally conceived with and for the Ruby programming language [12], it supports many other ones such as C++, which makes it more suitable to use with SystemC.

The so-called **feature files** describe the software's expected behavior (and hence define the test cases) and are written with a language called Gherkin (which contains natural language constructs). Listing 4.1 presents and example feature file in Gherkin. The object under test is a simple `Calculator`, which provides the `push()` and `divide()` methods. The example describes the `Calculator`'s expected behavior for the division feature.

Listing 4.1 – Feature file for the division feature of a Calculator

```
1  # language: en
2  Feature: Division
3    In order to avoid silly mistakes
4    Cashiers must be able to calculate a fraction
5
6    Scenario: Regular numbers
7      Given I have entered 3 into the calculator
8      And I have entered 2 into the calculator
9      When I press divide
10     Then the result should be 1.5 on the screen
```

Each feature file contains a number of scenarios (test cases), which are a sequence of steps. A **step** is a line of text which outlines a precondition (`Given`), an action (`When`) or an expected result (`Then`). Each step corresponds to a certain code snippet that should be executed, which is called **step definition**. Step definitions are implemented in a separate file using a regular programming language (Java or C++, for example).

In order to test software developed with many programming languages, Cucumber's feature file parser (written in Ruby) communicates with a separate executable which was written in the target language (in this case, C++) by the user. This executable contains the step definitions, which are compiled and linked to the C++ implementation of the framework. The parser reads the feature file and communicates with the target language executable to indicate which code should be executed to run each test.

Figure 4.1 illustrates the inner workings of the framework's C++ implementation [13]. The Ruby part receives a set of feature files as input; for our purposes, these files are a structured textual description of the tests. After the Ruby module parses the feature files, it knows which steps should be executed to carry out the tests.

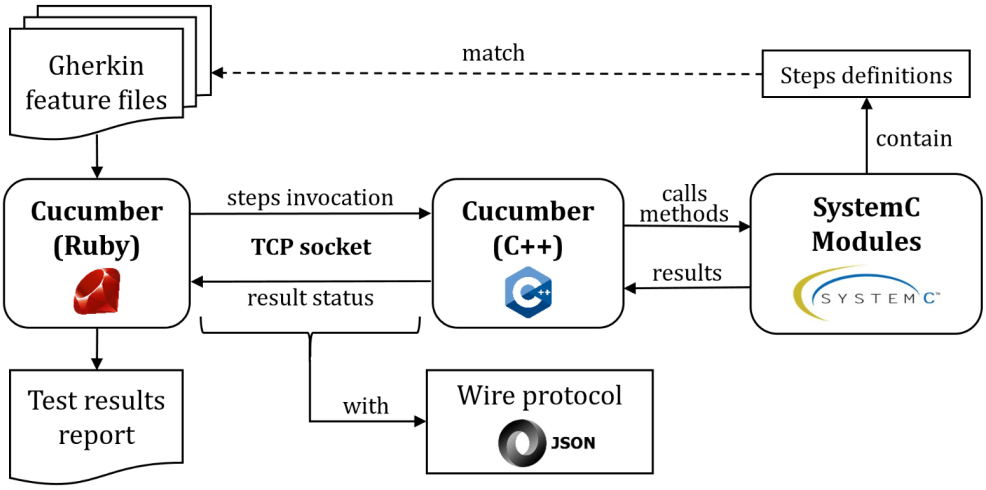

Figure 4.1 – Cucumber C++ inner workings and its integration to SystemC modules

It then establishes a communication with the C++ part of the framework (which corresponds to a different executable) through a TCP (Transmission Control Protocol) socket. This C++ element is a server (provided by the framework) which contains the definitions implemented in C++ of the steps present in the feature files (provided by the tester). Listing 4.2 contains the C++ definitions of the steps present in the `Calculator`'s feature file.

Listing 4.2 – Steps definitions for the division feature of the `Calculator`

```
using cucumber::ScenarioScope;
struct CalcCtx {
    Calculator calc;
    double result;
};
GIVEN("^I have entered (\\d+) into the calculator$") {
    REGEX_PARAM(double, n);
    ScenarioScope<CalcCtx> context;
    context->calc.push(n);
}
WHEN("^I press divide") {
    ScenarioScope<CalcCtx> context;
    context->result = context->calc.divide();
}
THEN("^the result should be (.*) on the screen$") {
    REGEX_PARAM(double, expected);
    ScenarioScope<CalcCtx> context;
    EXPECT_EQ(expected, context->result);
}
```

Through the exchange of JSON[3] messages which comply to Cucumber's so-called Wire protocol [14], the Ruby element requests the execution of a C++ step, one at a time. The C++ part executes the corresponding code and replies with the operation's status. For the considered SystemC bricks, this execution corresponds to calls to their methods, as shown in Figure 4.1. Listing 4.3 shows the JSON messages which are exchanged between the Ruby and C++ components of the framework to execute a single step of the `Calculator` example.

Listing 4.3 – Exchanged JSON messages for the execution of the tests

```
Scenario: Regular numbers # division.feature:6
> ["begin_scenario"]
< ["success"]
> ["step_matches",
    {
      "name_to_match":"I have entered 3 into the calculator"
    }
  ]
< ["success",
    [
      {
        "regexp":"^I have entered (\\d+) into the calculator$",
        "args":[{"val":"3","pos":15}],
        "id":"1",
        "source":"CalculatorSteps.cpp:12"
      }
    ]
  ]
> ["invoke",
    {
      "args":["3"],
      "id":"1"
    }
  ]
< ["success"]
 Given I have entered 3 into the calculator # CalculatorSteps.cpp:12
```

Through the exchange of these messages, The Ruby component knows which steps yielded the expected results and which tests were successful. It is then able to output a report with the tests results. Listing 4.4 contains the resulting output for the `Calculator` example.

Listing 4.4 – Execution of the tests for the division feature of the Calculator

```
Feature: Division
  In order to avoid silly mistakes
  Cashiers must be able to calculate a fraction

  Scenario: Regular numbers                     # division.feature:6
    Given I have entered 3 into the calculator  # CalculatorSteps.cpp:12
    And I have entered 2 into the calculator    # CalculatorSteps.cpp:12
    When I press divide                         # CalculatorSteps.cpp:28
    Then the result should be 1.5 on the screen # CalculatorSteps.cpp:23

1 scenario (1 passed)
4 steps (4 passed)
```

[3] JSON (JavaScript Object Notation) is a lightweight data-interchange format.

4.1.3 Adapting Cucumber to SystemC

In order to use the Cucumber C++ framework to test SystemC bricks, it is necessary to integrate it to the SLM team's development environment. The framework's compilation differs according to the used compiler and target operating system; moreover, it is also necessary to build its dependencies for each of these options. This is possibly a constraint for the utilization of the framework for the testing of SystemC bricks since all environments supported by the team would have to be considered, as explained in Section 3.1.

Nonetheless, in order to further study the suitability of the framework, it is integrated to one of the supported environments. Next, a part of one of the register bank's test suites is rewritten in the form of a feature file. A SystemC module containing the corresponding steps definitions and instantiating the Cucumber server is built. The tests corresponding to this feature file are then executed. Even though they are successful, a few complications become clear:

- **No simulation tests:** the successfully rewritten tests are executed during SystemC's construction time. Writing tests that require simulation with Cucumber is much more complicated. In order to start the simulation, `sc_start()` must be called during the tests execution. However, the function that starts Cucumber's C++ server is called within `sc_main()` and it only returns after all tests have been carried out. The call to the function `sc_start()` must therefore be made within a scenario; nonetheless, since Cucumber does not guarantee an order of execution for the tests, it is not possible to do so, as construction tests must be carried out before simulation ones. The fact that `sc_module` objects required by all scenarios must be instantiated before the beginning of simulation is a problem for the same reason. Consequently, Cucumber does not support the writing of tests that require simulation.

- **Fatal failures:** if a test scenario fails in a manner considered fatal by SystemC, the C++ executable quits, and the Ruby component driving the tests execution is not be able to continue with the next scenario as it should. This problem may be solved to some extent through the modification of Cucumber's Wire protocol; nonetheless, maintaining a modified version of the framework would encumber even further the support of the required environments.

- **Compound and shared structures:** the rather rigid structure of Cucumber's step definitions makes it difficult to write complex tests which require compound structures. Moreover, there is no simple way of sharing an instance of an expensive object between many scenarios.

- **Base elements operations:** in order to test a SystemC brick with Cucumber, it is necessary to write one or more step definitions for each of the operations that may be carried out with it (most commonly, method calls). The initial idea was to provide and maintain these definitions for the base elements in order to accelerate the writing of tests. However, the interface of most base elements is rather extensive (i.e. the underlying objects provide a large number of methods) and changes rather frequently. When steps are not repeated

a considerable amount of times, one may conclude that the advantages brought by the methodology do not justify the overhead of linking each step to its definition. With regard to this aspect, BDD is therefore somewhat inadequate for the testing of SystemC bricks. It could, however, be more suitable for the testing of components, whose interfaces are generally simpler and more repetitive (e.g. read and write operations when connected to a communication bus).

4.1.4 Conclusions about BDD approach

After the analysis of Cucumber's inner working and the rewriting of some existing tests, several problems regarding its usage to test the SystemC bricks developed by the SLM team became clear, notably:

- the difficulty of integrating and maintaining the framework and its dependencies for each of the supported environments;
- the inadequacy of having to write and maintain step definitions for each of the many base elements' operations;
- the impossibility of executing tests during simulation;
- the complication of maintaining a modified version to support SystemC fatal failures;
- the difficulty to declare and share complex structures between tests.

It is therefore clear that Cucumber should not be directly used for the testing of SystemC bricks and that a more classical approach is to be considered. Nonetheless, a BDD framework which respects the SystemC and maintenance constraints above could very well be suitable for the testing of components – such a tool could be possibly developed in a future project, for example.

4.2 C++ unit testing solutions

Amongst the multitude of available C++ open-source unit testing frameworks, two of them are studied in order to better understand the practices of existing unit testing solutions. Their features are analyzed ir order to understand their suitableness with regard to the identified requirements. Next, some example existing tests are reimplemented with these frameworks to verify their applicability. As a result of this analysis, common features for this kind of tool are identified; moreover, an informed decision about whether to directly use them for the testing of SystemC bricks is made.

Table 4.1 – Feature comparison between Google Test and Boost.Test

Feature	**Google Test**	**Boost.Test**
Rich set of assertions	yes	yes
Custom assertions	yes	yes
Tests hierarchy	yes (tests and test cases)	yes (test cases and modules)
Choose tests to execute	yes	yes
Descriptive results reporting	yes	yes
Customizable reporting	yes (printers)	yes
Fixture (separate instance)	yes	yes
Fixture (shared instance)	yes (`SetUpTestCase()`)	yes (test suite level fixture)

4.2.1 Studied frameworks

The considered open-source unit testing frameworks are **Boost.Test** [15] and **Google Test** [16]. They are analyzed thanks to their portability, complete documentation and widespread use – for instance, Google Test is used in many notable projects, such as the LLVM compiler [17] and the OpenCV computer vision library [18]. They provide many features which could be leveraged to develop our framework, such as test results reporting, custom assertions and handling of fatal errors; most importantly, they are highly customizable.

As shown in Table 4.1, both frameworks provide a rather similar set of features:

- **Assertions:** they provide a rather complete set of assertions, which are macros that resemble function calls. Users may also define their own assertions.
- **Hierarchy:** assertions are called within unit tests, which are pseudo test functions that are also defined by macros (in Google Test, they are simply called tests; in Boost.Test, test cases). Unit tests are placed within a hierarchy: in Google Test, a test case is a set of tests; in Boost.Test, a test module may contain many test cases and also other test modules (and therefore allows a more complex test tree).
- **Choose tests to execute:** it is possible to select the tests to be executed. For example, with Google Test, we may choose to only run the tests which are in the test case that corresponds to a feature currently being modified. In the case of a regression, it is simple to execute only the failing tests for debug purposes.
- **Reporting:** The frameworks also provide clear reports of the test results. For example, they clearly show which test is being executed, which assertions have failed and for what reason. These outputs are both configurable and extendable.
- **Fixtures:** both frameworks provide a way of declaring setup objects (i.e. code and data) to be reused by different tests. Each unit test may use a separate fixture instance or a single instance may be shared by many tests.

4.2.2 Tests reimplementation

Both of the studied unit testing frameworks are integrated to one of the environments supported by the SLM team. Some of the existing tests for the register bank brick are then rewritten with both of them. Even though a few simple cases are successfully developed, some conflicts with previously defined constraints arise:

- **Lifetime of SystemC objects:** when reusing a fixture object for many tests, the frameworks makes an instance of it at the start of each unit test and destructs it when the test is over. In order to test SystemC bricks, most of these fixtures are or contain SystemC related objects (notably modules). As explained in Section 2.1, after being instantiated, these objects may only be destructed after the end of simulation.
- **Simulation:** in both frameworks, the invokation of tests is carried out with a function call which only returns after all tests have been executed. The call to `sc_start()` for the tests that should be carried out during SystemC simulation must therefore be done by one of the tests. Since tests are independent and there is no guaranteed order of execution, it is not possible to divide construction and simulation tests by simply using these frameworks out of the box. Even if this division were possible, all SystemC objects required by the tests would have to be instantiated before the start of simulation. As explained in the previous problem, the frameworks instantiate them on demand for each of the tests.
- **Modification and maintenance:** the problems above could possibly be solved by modifying the framework; however, after a brief analysis of their implementations, it is clear that this task would be far away from trivial. Moreover, maintaining such a modified version (either to correct bugs or to add features) would be a rather cumbersome responsibility, since it would require not only having good knowledge of both the original framework and the modifications that were made, but also handling it for all the supported environments.

4.2.3 Conclusion

After the analysis of both unit testing frameworks, many features that would be interesting for the testing of SystemC bricks have been identified. However, some of the constraints imposed by SystemC conflict with them and hence limit their off-the-shelf usage. Since it would be complicated to maintain a highly modified version of one of these frameworks, one may conclude that the studied solutions should not be used for the testing of SystemC bricks.

4.3 Results

The first result of the performed analyses is the identification of the requirements to be respected by the solution to be developed. They are summarized in Table 4.2, which also shows whether the studied frameworks satisfy each one of them.

The second result is the conclusion that the studied frameworks should not be used for the testing of SystemC bricks, as they do not satisfy some of the established requirements. Therefore,

Table 4.2 – Requirements for the unit testing framework for SystemC bricks

ID	Requirements	Cucumber	Google Test	Boost Test
0	set of assertions which is enough to verify the correct behavior of SystemC bricks	✓	✓	✓
1	verify that a message has been generated by the tested base element (i.e. `tlm_message` assertions)	✗	✗	✗
2	separate the tests which verify different features of the object under test	✓	✓	✓
3	choose which tests should be executed	✓	✓	✓
4	specify if each test should be carried out during construction or simulation and have its execution handled by the framework	✗	✗	✗
5	define test fixtures which may be reused and shared by different tests and which are instantiated and destructed correctly by the framework	✗	✗	✗
6	clear output of the tests results separating different tests and giving details about the failed assertions	✓	✓	✓
7	compile and execute the tests written with the framework in all environments supported by the SLM team	✗	✗	✗
8	extend the framework's output	✓	✓	✓
9	define custom assertions	✓	✓	✓

a unit testing framework shall be developed from scratch. Doing so does not mean developing a piece of software as complex as the analyzed open-source frameworks, but rather a minimal framework which respects the constraints above and which is easy to maintain.

5. Proposed unit testing framework for SystemC

Based on the results of the analysis presented in Chapter 4, the first step of the development of the testing framework for reusable SystemC bricks (henceforward called **SystemC-Unit**) is to establish a list of features whose objective is to satisfy the defined requirements. An initial architecture is then devised and its implementation is described. The development of the architecture and the implementation was performed incrementally with the addition of the additional features[1]. Some validation exercises are carried out with the framework's future users, which allow the identification of a few missing features and the verification that the developed solution meets its users' needs. The testing of the framework is also described.

5.1 Features

The following list of features fulfills the established requirements for SystemC-Unit:

- **Assertions**: the framework provides a set of assertions, which are function-like macros that allows its user to specify the expected behavior for the objects being tested. For example, with the equality assertion, it is possible to compare the return value of a tested object's method to the expected value. In order to specify whether the current test should continue if an assertion fail, each one of them has both fatal and non-fatal variants. The framework handles the identification of which tests failed or not according to their assertions and the systematic printing of the assertions results (Requirement 0).
  - **Custom assertions**: since it is impossible to anticipate all the assertions users will need, the framework provides a way of defining new assertions (Requirement 9).
- **`tlm_message` assertions**: as aforementioned, the SystemC bricks to be tested make use of the `tlm_message` devkit for instrumentation purposes, that is, the printing of messages with different types. The framework provides a manner of asserting that, for a given operation, a given message with a given type is generated. Moreover, the devkit allows error messages to be fatal and the execution of the program is hence aborted; this behavior is detected but bypassed by SystemC-Unit so that the remaining tests may continue (Requirement 1).
- **Test structure**: tests defined with the framework are organized into the hierarchized structure (Requirement 2) shown in Figure 5.1, which is as follows:

[1] For simplicity's sake, only the final list of features and architecture are presented in this report.

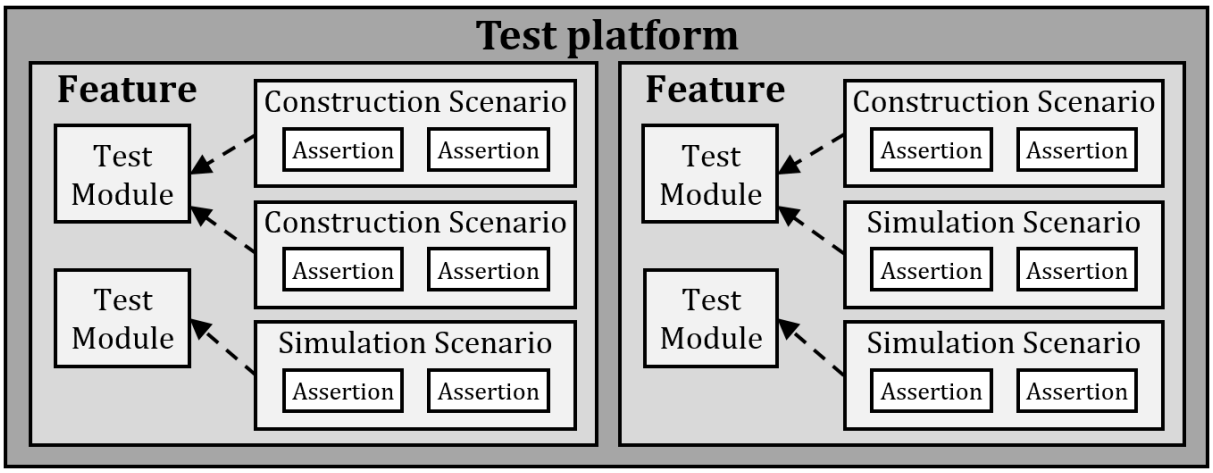

Figure 5.1 – Diagram exemplifying structure of tests developed with the framework

  - A **scenario** corresponds to the actual test code. It verifies the tested object's correct behavior by applying stimuli to it and by subsequently performing assertions. There are two types of scenario: **construction scenarios** and **simulation scenarios**. Each scenario's type indicate in which SystemC phase it should be executed and the framework guarantees a correct execution order (Requirement 4). Each scenario is linked to exactly one **test module** (see below), which contains the objects over which the verifications are carried out. Simulation scenarios also specify in which of its test module's processes they should be executed.
  - A **test module** is a SystemC module which is the test fixture (i.e.a class containing setup objects and code) required by one or more scenarios. Each scenario is linked to exactly one test module, but many scenarios may require the same test module. The framework handles their instantiation and destruction according to which scenarios will be executed. By default, each scenario receives an independent instance of the test module; since some test modules may contain objects whose instantiation is expensive, it is also possible to share an instance of a test module between many scenarios (Requirement 5). Test modules also contain the processes in which the linked simulation scenarios will be executed.
  - A **feature** is simply a set of scenarios which test the same brick's feature.

- **Choose tests to execute:** The structure presented in Figure 5.1 allows the framework's users to choose which tests they wish to execute by indicating the name of features or scenarios through command line options (Requirement 3).

- **Clear, homogeneous and useful reporting**: the framework provides a useful overview of the tests results, showing which scenarios and features have been executed, which ones were successful and which ones failed. Moreover, it gives details about the assertions that failed, such as where in the code it was carried out and what were the expected and actual

results (Requirement 6). The default output is somewhat light, since the most common objective of the tests' execution is simply to check that there have been no regressions. A more verbose output is also available and may be enabled with a command line option.

  - **Extendable reporting**: The default output may be replaced or extended with the addition of a user-provided object, which must implement the printer interface defined by the framework (Requirement 8). For instance, this feature may be used to generate a PDF or XML document with the tests results.

5.2 Implementation overview

The framework's architecture is mainly divided in four packages:

- **Core:** the framework's main entry point, it handles its configuration (e.g. choice of tests to execute) and extension (e.g. addition of a printer) by the user and the tests execution;
- **Assertions:** defines the classes corresponding to the different assertion types the users may carry out, as well as the class from which they should inherit in order to define their own;
- **Structure:** defines the classes from which the users should inherit in order to define their tests and handles their registration with the framework's core.
- **Printing:** defines the interface expected from printer objects, provides its default implementation (which generates the output with the tests results) and handles the existing printer objects.

Figure 5.2 represents the interactions between the framework's packages and the user-defined tests for a typical execution of tests, which goes as follows (the numbers below correspond to those of the figure):

1. The user develops tests by writing classes with inherit from classes provided by the framework;
2. When the execution starts, the user-defined tests statically self-register with the framework's core;
3. The user calls the framework's initialization and the core parses the user's command-line options (for example, indicating that a single test case should be executed and specifying which one);
4. The framework's core iterates over the tests hierarchy and the tests chosen by the user are executed (first, all construction scenarios, and then, the simulations ones):
   (a) Construction scenarios: the required test modules are instantiated on-demand and the test method is called;

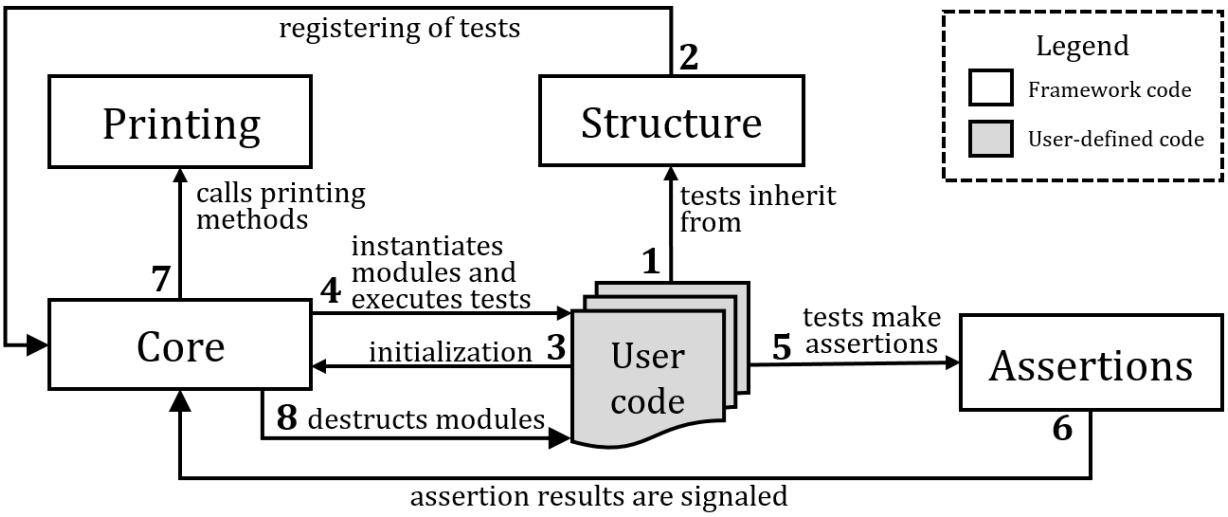

Figure 5.2 – Interactions between the framework's packages and the user-defined tests

(b) Simulation scenarios: the required test modules are all instantiated before the beginning of simulation; a SystemC scheduler module which controls the execution order of the simulation tests is instantiated; the simulation is started; finally, each test method is executed within the indicated test module's process in temporal isolation.

5. The test methods for both types of scenario make assertions;
6. These assertions are signaled to the framework's core so that it may stop the current test in case of failure for a fatal assertion;
7. The assertions are also transmitted to the printing package to output the tests results;
8. After the end of simulation, all objects instantiated by framework (SystemC related or not) are destructed.

In order to actually develop tests, the user writes classes which correspond to the elements of the framework's test structure: scenarios, test modules and features. These classes are defined with macros provided by the framework. The tests are organized in an nested fashion: a feature class contains multiple test modules classes, and a test module class contains multiple scenario classes. Each feature class is written in a separate file. Within one feature class, the user defines different test modules classes. Construction and simulation scenarios classes are then defined within the test modules containing the objects they test.

5.3 Example Test

Listing 5.1 shows an example of tests written with the developed framework for a ficticious calculator brick. We assume that it only handles unsigned integers and that its bit size is configurable. Moreover, overflow may be disabled (i.e. an error message is generated if it occurs),

but there is a bug in this functionality (no message is actually yielded). The following points are worth mentioning:

- The test module class defines the configuration of objects which is reused by multiple scenarios (in this example, a single instance of `Calculator` on line 10);
- Since it is the framework that makes instances of the test module, the user provides a static method which gives the name to be be used to instantiate it (line 8) and the constructor (lines 12 to 15) which initializes its objects as it should;
- Each scenario class defines a test method, which applies stimuli to and makes assertion over the objects (the `Calculator` object) contained in the instance of the test module provided as parameter by the framework.

Listing 5.1 – Example of test written with the developed framework

```
1  FEATURE(Overflow) {
2      std::string name() { return "Overflow"; }
3
4      TEST_MODULE(Module8BitCalculator) {
5      public:
6          static std::string module_name() { return "MODULE_8BIT_CALC"; }
7
8          Calculator calculator;
9
10         Module8BitCalculator(sc_module_name name) : TestModule(name)
11         {
12             calculator.setBitSize(8);
13         }
14
15         CONSTRUCTION_SCENARIO(OverflowDisabled) {
16             std::string name() { return "Overflow Disabled"; }
17
18             void test(Module8BitCalculator & module)
19             {
20                 module.calculator.disableOverflow();
21                 ASSERT_ERROR(m_calculator.add(128, 128), "Overflow");
22             }
23         };
24
25         CONSTRUCTION_SCENARIO(OverflowEnabled) {
26             std::string name() { return "Overflow Enabled"; }
27
28             void test(Module8BitCalculator & module)
29             {
30                 ASSERT_EQUAL(module.calculator.add(128, 128), 0);
31             }
32         };
33     };
34 };
```

Listing 5.2 shows the output generated by the framework when executing the example above. Since only failed assertions are shown, it is rather easy to identify the problems (even for examples which are not as minimalistic as this one).

Listing 5.2 – Example output

```
Executing 1 features
============================================================
Executing 1 construction features
============================================================
Feature: Overflow
============================================================
Executing 2 scenarios
------------------------------------------------------------
Scenario Overflow Disabled...
    overflow.cpp:24: expected module.calculator.add(128,
      128) to yield error message containing "Overflow",
      but no error occured                                  [ERROR]
------------------------------------------------------------
Scenario Overflow Enabled...                                [ OK  ]
------------------------------------------------------------
Feature Overflow: finished (1/2 successful scenarios)       [ERROR]
============================================================
Finished all construction features (0/1 successful)         [ERROR]
============================================================
Finished all features (0/1 successful)                      [ERROR]
```

5.4 Implementation details

SystemC-Unit's implementation amount to around 4 thousand lines of code. In comparison to the analyzed open-source solutions, it is way simpler and may therefore be more easily maintained (for instance, Google Test consists of around 32 thousand lines of code).

Many technical challenges were surpassed in order to achieve the established requirements for the framework, such as:

- the cooperation with the different devkits commonly used by the SLM team to develop virtual prototypes, such as the `tlm_message` instrumentation devkit;
- the integration of the framework with SystemC, regarding both the correct execution order for construction and simulation tests and the instantiation and destruction of the SystemC modules required by the tests;
- the self-registration of test classes, so that the framework may implicitly know about the tests written by the user;
- the temporal isolation of simulation tests, which is transparent to the user and guarantees that processes corresponding to different tests don't interfere with each other.

A complete description of the framework's implementation may not be provided due to the industrial context of this work. Nonetheless, this section presents one of the interesting challenges which have been encountered: the self-registration of user-defined test classes.

Self-registering tests

When using the developed solution, writing tests simply consists in defining classes – the framework *automagically* knows about them. In order to understand how this behavior is achieved, let us first explain what does it mean for a class to be self-registering, then consider what criteria are required for it to be possible, and finally analyze how it is actually done in the framework.

In practice, saying the framework "knows" the user-defined classes means it contains (a pointer or a reference to) an instance of each of them. This could be achieved manually, that is, the user could define a class, instantiate it and add the object to the framework. However, it is more practical and less error-prone for this registration to occur automatically. The objective is hence to have a class that adds an instance of itself to the framework by simply being defined.

In order to do so, two points must be considered:

- The instantiation of these classes must not be carried out in the user's code, but rather in the framework itself. As a result, the information about the type of the user-defined class must be informed to the framework so that it may create an instance of it;
- This instantiation must occur automatically, that is, the simple fact of declaring the class means it will be instantiated.

An initial solution is to make use of rather long macros for the definition of the user's test classes. However, since one of the objectives of developing the framework from scratch is for it to be maintainable, this option is not suitable. The employed solution is compact, but quite complex. In order to explain how it works, the definition of feature classes is analyzed as an example.

Listing 5.3 shows a simplified version of the framework's code which handles the self-registration of features. Users define features with the `FEATURE` macro, whose definition is shown on lines 2 and 3. Notice that `Feature` is not a class, but rather a class template[2]. Each user-defined feature class inherits from a instance of the `Feature` class template (lines 6 and 7). The template argument for the instantiation is the derived class itself [3]. For example, if a feature class is called `Multiplication`, it should inherit from `Feature<Multiplication>`. As a result, we have an instance of the class template for each user-defined feature. Each of these instances has the knowledge of the real type of the derived class. By factorising the instantiation code in the class template, it is therefore possible to instantiate the user-defined classes.

[2]A class template is a specification for generating classes based on parameters (in our case, a generic type). The actual classes generated from a class template are called instances of it. They are implicitly instantiated when the class template is referred to with a given set of template arguments (in our case, the type of the derived class).

[3]This C++ idiom in which a class X derives from a class template instantiation using X itself as template argument is called Curiously Recurring Template Pattern (CRTP) [19].

Listing 5.3 – Code that handles the self-registration of feature classes

```
1  // feature macro definition
2  #define FEATURE(feature_name) \
3      class feature_name : public Feature<feature_name>
4
5  // feature class template definition
6  template <typename T>
7  class Feature : public FeatureBase {
8      // initialization of this dummy static variable will add an instance
9      // of the user-defined feature class to the framework's core
10     static bool dummy_init;
11     // forces initialization of the static variable above
12     typedef StructWithNonTypeParameter<bool &, dummy_init> dummy_type;
13 };
14
15 // definition of the dummy static variable
16 template <typename T>
17 bool Feature<T>::dummy_init = Core::addFeature<T>();
18
19 // initialization function
20 template <typename T>
21 bool Core::addFeature() {
22     // instantiates feature
23     FeatureBase * feature = new T();
24     // adds it to the vector of features
25     m_features.push_back(feature);
26     return true;
27 }
28
29 // struct template with a non-type parameter,instantiating it forces
30 // the initialization of the non-type argument
31 template <typename T, T>
32 struct StructWithNonTypeParameter {
33     /* nothing */
34 };
```

After having solved the first point, the next step is to make these instantiations occur automatically. To solve this problem, each instance of the `Feature` class template contains an static dummy boolean variable (line 11). The definition of each of these variables (line 18) will call an initialization function (line 22) which will create an object of the derived class (line 25) and add it to the framework (line 27). However, according to the C++ standard, the implicit instantiation of a class template does not cause any of its static data members to be implicitly instantiated, and therefore the initialization function is not called by simply writing the feature class. To address this issue, each class template instance makes a type definition containing a dummy struct template (lines 33 to 37) which takes a non-type template parameter, and the dummy variable is passed as argument (line 13). This type definition implicitly makes an instance of the struct template and this instantiation finally forces the initialization of the dummy boolean.

5.5 Testing

In order to verify the implementation of SystemC-Unit, a tool like itself is ideally required. One might consider using one of the studied open-source testing framework to write these tests. However, this option is not suitable for one of the same reasons for which these solutions are not directly used to test SystemC bricks (i.e. the effort of providing and maintaining them for all supported environments). SystemC-Unit's tests are hence developed as a plain C++ program. Nonetheless, a small set of testing functions is used in order to facilitate the framework's tests implementation and debug and to reduce some of the problems it aims itself to avoid.

The testing approaches for some of the framework's features are presented below:

- **Assertions:** for all types of assertion provided by the framework, they are instantiated with typical and possibly problematic parameters. The tests then verify if the assertions were correctly created; for example, the actual assertion result and message are compared to the expected ones.
- **Printing of values:** the decision of how different datatypes should be shown in the tests results is factorized in a single class of the framework. In order to test it, the results (string object) it yields for variables of commonly used and custom (i.e. unknown to the framework) datatypes are compared to the expected ones.
- **Self-registration:** the correct self-registration of tests is verified by adding known side effects to their constructors; for example, they could add their names to a known vector. The test code then checks if this vector contains the names of all defined tests.
- **Test execution:** after the definition of some tests, their correct execution is verified by adding a custom printer to the framework which knows about them and verifies that they are actually executed and yield the expected result.
- **Results reporting:** for a given set of tests, it is verified that the framework correctly reports their results by redirecting its output to a file or string and by comparing it to the expected output.

5.6 Documentation

The framework's implementation is documented with the Doxygen tool [20], which allows the automatic generation of documentation consistent with the source code. A user guide showcases the framework's features, explains how to configure, customize and execute it, and provides guidelines about the writing of quality tests with it. Finally, a maintenance guide clarifies some complicated sections of the implementation that are hard to explain solely with text as source code's commentary and which could be important to the framework's modification.

5.7 Validation

In order to evaluate if the developed framework meets its users' requirements, the following validation activities were carried out:

- Throughout the implementation, some of the existing tests for the initially analyzed base element (the register bank) have been rewritten with the framework. Its correct operation has therefore been verified by checking that the new tests yield the same results as the previous ones.

- During the framework's development, when the feature list was somewhat mature, a meeting was held with its future users. The objective was to show an intermediate version of the solution in order to obtain feedback about whether it met their needs. Moreover, the intended features yet to be developed were also presented, so that developers could validate that the development was going in the right direction. The reactions were mostly positive, as there were not many reproaches and most comments concerned the addition of non-critical features.

- Some existing tests for the SystemC bricks other than the initially analyzed ones were rewritten. This not only validates the framework's operation as above but also verifies that all the required common features were successfully extracted from the analysis of the register bank's tests.

- A developer used the framework to implement a few tests for different SystemC bricks. With his remarks, it was possible to validate the framework's impact in performance: the average size reduction in lines of code for the rewritten tests was 21%. Moreover, a few unclear interfaces and documentation were identified and fixed.

6. Concluding Remarks

The objective of this work was to solve the problem of testing reusable SystemC elements, which are used to construct virtual prototypes of hardware systems. Initially, an analysis of the SLM team's development environment and of the existing tests of an example base element yielded a set of constraints to be respected for the testing of these elements. Next, the analysis of some open-source testing frameworks allowed not only the verification of their applicability regarding the already defined requirements, but also the identification of new ones.

Since it was concluded that the studied frameworks are not suitable for the testing of SystemC elements, a solution had to be developed from scratch. The proposed unit testing framework for SystemC elements was called SystemC-Unit. A set of features to satisfy the established requirements was defined and fully implemented. Its implementation amounts to around 4 thousand lines of code, way less than the studied open-source frameworks. Furthermore, SystemC-Unit was fully documented, tested and validated. Tests written with SystemC-Unit are, in average, 21% smaller (in lines of code) than the previously existing tests.

A first official version of the test framework has been released for the System Level Modeling team. Its maintenance will be responsibility of this project's supervisor; this is a logical choice since he followed and guided the project's development from the beginning.

In time to come, the framework may be extended in order to support the testing of more complex SystemC components (e.g. a complete model of an IP block), which are built upon the reusable bricks that the developed tool aims to test. Even though the requirements for these models are different than those for the aforementioned bricks, many aspects of the framework may surely be leveraged.

Bibliography

[1] Accellera Systems Initiative. The SystemC standard. http://www.accellera.org/downloads/standards/systemc. Accessed: 25 March, 2016.

[2] STMicroelectronics. ST company information. http://www.st.com/web/en/about_st/st_company_information.html. Accessed: 25 March, 2016.

[3] Accellera Systems Initiative. The IP-XACT standard. http://www.accellera.org/downloads/standards/ip-xact. Accessed: 25 March, 2016.

[4] Michael Nash. Overcoming not invented here syndrome. http://www.developer.com/design/article.php/3338791/Overcoming-quotNot-Invented-Herequot-Syndrome.htm. Accessed: 25 March, 2016.

[5] Accellera Systems Initiative. Accellera Systems Initiative presentation. http://www.accellera.org/about. Accessed: 25 March, 2016.

[6] Ian Sommerville. *Engenharia de software*. Pearson Brasil, 2011.

[7] Gerard Meszaros. *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley Signature Series (Fowler). Pearson Education, 2007.

[8] Michaela Greiler, Andy Zaidman, Arie van Deursen, and Margaret-Anne Storey. Strategies for avoiding text fixture smells during software evolution. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 387–396. IEEE Press, 2013.

[9] Cucumber Limited. Gherkin. https://github.com/cucumber/cucumber/wiki/Gherkin. Accessed: 1 December, 2016.

[10] J.F. Smart. *BDD in Action: Behavior-Driven Development for the Whole Software Lifecycle*. Manning Publications Company, 2014.

[11] Cucumber Limited. The Cucumber framework. https://cucumber.io/. Accessed: 25 March, 2016.

[12] Ruby community. The Ruby programming language. https://www.ruby-lang.org/en/about/. Accessed: 7 June, 2016.

[13] Cucumber Limited. The Cucumber C++ implementation. https://github.com/cucumber/cucumber-cpp. Accessed: 25 March, 2016.

[14] Cucumber Limited. The Cucumber Wire protocol. https://github.com/cucumber/cucumber/wiki/Wire-Protocol. Accessed: 25 March, 2016.

[15] Boost.Test contributors. The Boost.Test library. http://www.boost.org/doc/libs/release/libs/test/. Accessed: 25 March, 2016.

[16] Google Inc. The Google Test framework. https://github.com/google/googletest. Accessed: 25 March, 2016.

[17] LLVM contributors. LLVM unit tests. http://llvm.org/svn/llvm-project/llvm/trunk/unittests/. Accessed: 4 November, 2016.

[18] OpenCV contributors. OpenCV: C++ tests. http://code.opencv.org/projects/opencv/wiki/QA_in_OpenCV#C-tests. Accessed: 4 November, 2016.

[19] Wikimedia Foundation Inc. More C++ idioms: Curiously Recurring Template Pattern. https://en.wikibooks.org/wiki/More_C++_Idioms/Curiously_Recurring_Template_Pattern. Accessed: 25 May, 2016.

[20] Dimitri van Heesch. Doxygen. http://www.doxygen.org/. Accessed: 1 December, 2016.