Coverage-based validation of embedded systems

Maria Leonor Ferreira da Costa Cunha Bento

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Examination Committee

Chairperson:

Supervisor: Doutor José Carlos Campos Costa

Co-Supervisor: Doutor José Carlos Monteiro

Members of the Committee: Doutor Ricardo Jorge Fernandes Chaves

May 2015

Abstract

Embedded systems have become over time increasingly rooted in various fields of application and also more complex. An example is medical equipment that human lives depend on. Therefore, it is crucial to have a guarantee of proper operation of these systems prior to installation. The aim of this work is to implement a validation methodology for embedded systems based on a particular coverage.

Given a high-level description of the embedded system in SystemC and a quantitative level of coverage based on observability, specified by the user, the objective is to develop a tool that determines the input vectors exercising execution paths in order to obtain the specified level of coverage.

The proposed solution consists in initially computing a minimal set of paths that reach the specified level of coverage, and then determine which input vectors exercise each of these paths. The work will consist of the following recursive method: (a) Get the corresponding directed graph without cycles, from the SystemC description; (b) Using the graph, obtain the path that potentially covers more lines of code in the SystemC description; (c) Get the input vectors that exercise the path obtained; (d) Calculate the coverage obtained with these input vectors. (e) If the path does not achieve the desired coverage, repeat procedure.

From the methodology mention above, we successfully implemented the System C description interpretation and emulation in order to generate the graph. Still, we were unable to automate the conversion to SMTs to get the input vectors.

Keywords

SystemC; Verification; Validation; Embedded Systems

Resumo

Os sistemas embebidos tornaram-se ao longo do tempo cada vez mais enraizados em vários campos de aplicação e consecutivamente mais complexos e especializados. Um exemplo disto são os equipamentos médicos, dos quais dependem vidas humanas. Assim, é crucial ter uma garantia de funcionamento adequado destes sistemas antes da sua instalação. O objetivo deste trabalho é a implementação de uma metodologia de validação para sistemas embebidos baseados numa cobertura particular.

Dados uma descrição de alto nível do sistema embebido em SystemC e um nível quantitativo de cobertura com base em observabilidade, indicado pelo utilizador, o objetivo é desenvolver uma ferramenta que determina os vetores de entrada que exercem caminhos de execução, a fim de obter o nível especificado de cobertura.

A solução proposta consiste inicialmente na computação de um conjunto mínimo de caminhos que alcançam o nível especificado de cobertura e, em seguida, determinar quais vetores de entrada que exercitam cada um desses caminhos. O trabalho consistirá no seguinte método recursivo: (a) Obter o grafo dirigido, sem ciclos, a partir da descrição SystemC; (b) Usando esse grafo, obter o caminho potencialmente mais abrange; (c) Obter os vetores de entrada que exerce o caminho; (d) Calcular a cobertura obtida com estes vetores. (e) Se o caminho não alcançar a cobertura desejada, repetir o procedimento para outro caminho.

Da abordagem mencionada acima, conseguimos implementar com sucesso a interpretação, emulação da descrição SystemC, conseguindo assim gerar os grafos do sistema. No entanto, nesta fase a conversão de instruções para SMT ainda é feita manualmente.

Palavras-chave

SystemC; Verificação; Validação; Sistemas Embebidos

Acknowledgments

This project is part of the research project *PTDC/EEA-ELC/122756/2010 – Cervantes Co Validation Tool for Embedded Systems* in the Algos group of INESC-ID Lisbon.

Table of Contents

List of Figures

List of Tables

1. Introduction

An embedded system is an integrated computational system, composed by a software component and a hardware component. The software component confers flexibility to the system and can be easily changed depending on the application. The hardware component is responsible for performing specific tasks and specialized in subsystems used in more demanding application in terms of performance. Embedded systems are designed to perform certain functions and they require a higher degree of quality and reliability throughout the implementation process, when compared to other types of computational systems.

Our society does not realize the amount of embedded systems that surround us and that we interact with daily. Embedded systems are used in various fields such as telecommunications, consumer electronics, industrial, medical, avionics, military and others. Examples of embedded systems use are: the phone, printer, systems that integrate a car (like the engine control unit (ECU)), digital television, domotics applications, and vital signs monitors, among others.

Taking as an example the mobile phone, we can step back a mere 20 years to demonstrate the increasing complexity that this system has suffered. Mobile phones emerged with the sole purpose of being a way of mobile communication. Nowadays, due to the various technological developments, the phone went from a simple object to make calls to an authentic computer that combines several different machines like a game console, GPS, camera, among others. Today, thousands of new applications can be downloaded to these devices with the greatest of ease.

As mentioned earlier, the degree of reliability required in embedded systems is very high, but it varies depending on the field of application of the system. In areas of application where embedded systems are deployed in remote or difficult places to access, the reparation of a fault can be extremely expensive or impossible. The degree of reliability required is very high because they need to run smoothly for a long time. In these cases, it is crucial to ensure and guarantee the proper functionality of the system prior to its use.

Thus we require tools that help the industry to keep up with rapid market developments, so we need to meet the following requirements:

- reduce the production time of a product *i.e.*, its development should be faster resulting in a reduced time to market, With the decreasing prices on electronic products, consumers are always waiting for the latest novelty in stores. Whenever a new model comes from a certain company, it forces others to quickly inject new and more sophisticated models to market. To keep their market share, companies are forced to closely monitor the market trend, and constantly evolve;

- deal with the increased functionality required by the final consumer, these systems has been increasing its processing capacity. The increased number of transistors allows the increasing number of features on silicon.
- find appropriate mechanisms for embedded systems testing that have realistic and acceptable criterias for the goals of these systems

As a solution to the problems mentioned above, there were already proposed techniques based on formal verification of embedded systems, but they present serious limitations: can handle large programs, but only in cases where loops are non-existent; can handle loops, but it rises problems of scalability. Thus a better alternative for validating a system would be simulation.

The solution that we will present is based on simulation. It takes as input, a SystemC description of an embedded system. SystemC is a C++ library that also supports the emulation of hardware descriptions. This allows the development of hardware/software descriptions in parallel, since the early stages of the designing process. It also receives as input a quantitative coverage metric, based on observability. The tool will determine the input vectors that exercise the execution paths that reach the level of coverage defined by the user.

1.1. Objectives

The objective of this project was to develop a tool for co-validation of an embedded system in order to provide the engineer with an extra tool for validation in the early stages of the design process of an embedded system. It receives as inputs a description of the system and a coverage level that needs to be achieved.

The system description will be written in SystemC [1]. SystemC is a C++ library, which let us make use of all the faculties of C++ and adding the necessary components for hardware emulation. It allows us to describe the hardware/software specification of components, using a well-known language, the C++.

The other input required by the tool consists on a percentage value that will correspond to the minimum coverage of the program required. The subject to analyze is an embedded system, so the level of coverage will be a mixture of software and hardware concepts. The level of coverage will be based in statement coverage combined with observability coverage. This will produce a more realistic analysis of the program.

In statement coverage, achieving 100% coverage means that all instructions of the program were executed. Combining this approach with observability coverage means that we only take into account the instructions that produce some effect on the outputs. It is very difficult to achieve 100% coverage in a real program due to various reasons, such as the existence of “garbage code”, *i.e.*, there may be functions that are not called in the program.

The main task of this tool is to determine the input vectors needed for that particular execution path that is covered and the result produced by the execution path. If the tool is not able to find input vectors for a given execution path, then we say that the path is not feasible.

The main goal of this thesis is to develop the remaining components and link them, to unify them in a single tool.

Summarizing the goals of this thesis are:

1. Studying the existing tools for validation of an embedded system and related methods;
2. Search technologies compatible with SystemC;
3. Develop the component **SysCFG** to extract the CFG of each function of the source code from a SystemC description;
4. Develop the component **Extract Structure**, which will extract the structure of SystemC program: their modules, signals and the connection between them;
5. Develop the component **SysSMT** to convert SystemC instructions in SMTs and get the input vectors of a given path;
6. Develop a solution to integrate the components of the tool.
7. Test the solution with real programs, such as the echo cancelation, described in Section 6.

The goals 1 and 2 will be explained in tools and related work chapter that we will present in the Sections 2 and 3. Goals 3, 4 and 5 correspond to the implementation of the components **SysCFG** (see Section 5.1), **Extract Structure** (see Section 5.2) and **SysSMT** (see Section 5.3). These are the components developed in the scope of this thesis. Goal 6 is explained in Section 4. Finally, the goal 7 will present the results for the implemented components in this thesis. In the end of goal 7 we will have presented a viable solution for a tool for co-validation of embedded systems that receives as inputs a SystemC description and a coverage level that needs to be achieved. Also we will present in detail the implementation and validation of the components **SysCFG**, **Extract Structure** and a viable solution for the implementation of the **SysSMT** component.

1.2. Methodology

In this solution we will use a SystemC front-end called PinaVM (see Section 2.2.2) to generate a LLVM bytecode file (LLVM IR). This file contains the system represented in a format that can be easily handled by the LLVM [2] framework.

Then, during a recursive process the following tasks will be performed:

1. Extract from LLVM IR (see Section 2.3.2) the CFGs of the source code functions;
2. Convert the LLVM IR instructions to SMTs.
3. Extract the structure of the program described in SystemC, *i.e.*, the relationship between the various components, the modules and signals.
4. Build a Control Flow Graph (CFG) of the entire program from the previously extracted information.

5. Create a Directed Acyclic Graph (DAG) of our program, where it is stored information required to calculate the longest path;
6. Based on the coverage given, determine the longest path which achieves an equal or greater level of coverage compared with the specified by the user;
7. Calculate the input vector which exercises that path. Thus, it is necessary to use a solver, in this case, it will be used **Z3** [3].
8. If a path is found with the above conditions, return the path and the respective input vector, else return to point 5.

1.3. Document Structure

This thesis is organized as follows: Section 2 and 3 present the tools and the related work of our solution. Section 4 shows the architecture of the tool, the relation between the various components and a brief description of some of the approaches attempted, but that failed to fulfill the requirements. Section 5 presents the implementation details of the components that are part of the scope of our solution: (i) **SysCFG**; (ii) **Structure Extract**; (iii) **SysSMT**. Section 6 presents the results we achieved with the **SysCFG**. Finally, in Section 7 we present our conclusions and discuss future work.

2. Tools

In this Section we will talk about some tools related to our framework.

2.1. SystemC

SystemC is a C++ library, which allows modelling of hardware concepts, such as, hardware timing, concurrency, reactive behaviour, low-level communication, synchronization, higher-level communication (bus and network protocols), which are fundamental concepts that are scattered among several languages.

SystemC was designed for simulation, but it can be used for formal analysis.

SystemC allows the system to be modelled in different levels of abstraction, from the function description to the accuracy-cycle modelling. This brings a great an advantageous for systems with a high level of complexity.

SystemC accepts any type of representation of the system in a single simulation. It is sometimes mistaken as a language, but in reality, it is just a C++ library implemented through methodologies such as classes and macros. Due to this reasons, SystemC does not introduce any new syntax to C++, allowing the use of a wider range of tools of C++ with it.

To build a description of an embedded system in SystemC, it is possible to use C++ standard compilers such as GCC [4], LLVM [5], EDG [6] or Visual C++ [7], just by adding the SystemC libraries for generating an executable of the program.

SystemC had a wide acceptance by the scientific community. It has a slight learning curve since it is C++. It has been developed by the Open SystemC Initiative (OSCI) since 1999. Over the years, there have been several versions released and in 2006 it was approved as an IEEE standard. Today it is the standard specification, modelling, simulation and verification of electronic design Automation (ESL).

2.1.1. Components

SystemC provides several constructors that allow the modelling of hardware. Some of the most important ones are discussed below.

Figure 1 -illustrates a simplified view of a module. A module is the basic unit to describe a structure in SystemC. Modules contend ports, process, signals or other modules. The module, in Figure 1, shows various types of ports: input, output and inout which allows the module to communicate with other modules. In SystemC, it is also possible to construct other types of ports.

Another component is the process. The processes are used to describe the functionality of a module and implement a concurrent behavior. They can be compared to the methods of a class and are

sensible to changes in the signals and ports. There are several types of processes, for the modulation of different levels of abstraction. In Figure 1, are shown two SC_METHODS processes.

Signals are used to make the link between the processes and the modules. They are like wires carrying a value and are updated after a delta delay. There are two types of signals: resolved signals that may have multiple drivers (a bus) and unresolved signals that can only have one driver. Figure 1 shows that the links between the two processes and the child module are made by signals. In SystemC, signals are implemented as various types of variables. In this thesis, we will consider that the signals are represented by integer variables.

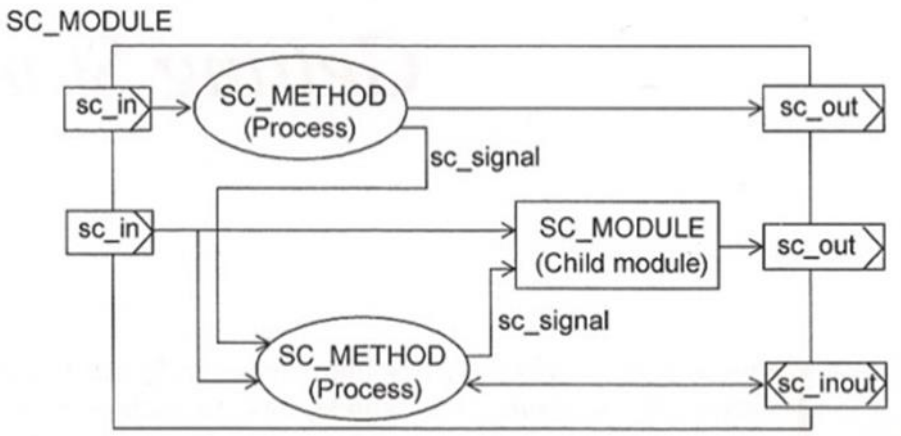

Figure 1 - A module, ports, processes and signals. Source [1].

2.1.2. Data Types

As stated previously, SystemC is C++; therefore, various native data types of C++ are compatible with the description of SystemC, but with some restrictions. Pointers can be used but certain synthesis tools cannot deal with them.

SystemC library adds some new data types, such as sc_bit that only accepts the values '0 'and '1' and sc_logic that accepts the values '0 ', '1', 'Z' and 'X'. Any syntax regarding SystemC has the prefix "sc_".

2.2. SystemC Front-ends

In SystemC a simple C++ compiler is sufficient to generate an executable of the system. But sometimes, to generate an executable of the program is not the objective. There are applications that need to make changes, analyses or optimizations before reaching the executable, something impossible for a C++ compiler. In these cases it is necessary to use a front-end created especially to cope with SystemC.

A SystemC front-end may have several objectives: hardware synthesis, optimized compilation, symbolic formal verification, source code instrumentation, verification. There are several SystemC front-ends on the market [8], but all of them have different approaches, accompanied by limitations, difficult installations and hard usability.

The implementation of a SystemC front-end requires dealing with the complexity of C++, the system architecture and the linking of the two parts. Another decision to make is the format that you want for the intermediate representation (IR). The most common formats are the abstract syntax tree (AST), and control flow graph (CFG) but the decision must take into account the information required by the back-end.

To deal with all the C++ grammar, some implementations consider SystemC as a language and build from scratch a C++ parse and a SystemC parse. This approach makes the front-ends limited due to the complexity of the C++ language, because some aspects of the language are not fully implemented and some dialects of C++ can be created.

Other implementations use an existing C++ parse and only have to worry about building a SystemC parse. This approach yields better results since it is assured the support all the syntax of C++.

SystemC is a library that allows us to define architectures. For that, it uses macros and classes of C++ to define components and connectors which are instantiated at the beginning of the elaboration phase. The elaboration phase corresponds to the phase which lets you extract the architecture of the system at run-time, by running and linking the code to extract the SystemC constructors.

There are two ways to extract the SystemC architecture: the first is to execute the elaboration phase and extract the result. The second way is to perform a parse of the elaboration phase and infer the result without having to run it. [8]

2.2.1. Pinapa

PINAPA (**P**inapa **I**s **N**ot **A** **PA**rser) [9] is an open source SystemC front-end. Its objective is to perform simulation and has a different approach compared to the various existing front-ends at this time. It can extract accurate information from the system architecture and the behaviour of its components, without major limitations. However Pinapa presents some limitations when complex architectures are present, because unlike other approaches, it does not interpret the code. Instead, it compiles and executes. Pinapa does not require any manual changes. To deal with the extensive C++ language, Pinapa uses a well-known C++ front-end, the GCC.

Despite being able to generate an executable of a SystemC program with a simple C++ front-end, it is not sufficient to obtain all the necessary information. Figure 2 shows how each front-end treat the various types of information. A C++ front-end considers lexicography and syntax as static information to build the AST, and the architecture and behaviour as dynamic information visible at runtime. Pinapa takes a different approach, and considers architecture as static information that can be extracted from the memory at the end of the elaboration phase. Thus static information consists of the AST that is created by the C++ front-end and elaboration phase (ELAB) which stores information about the architecture.

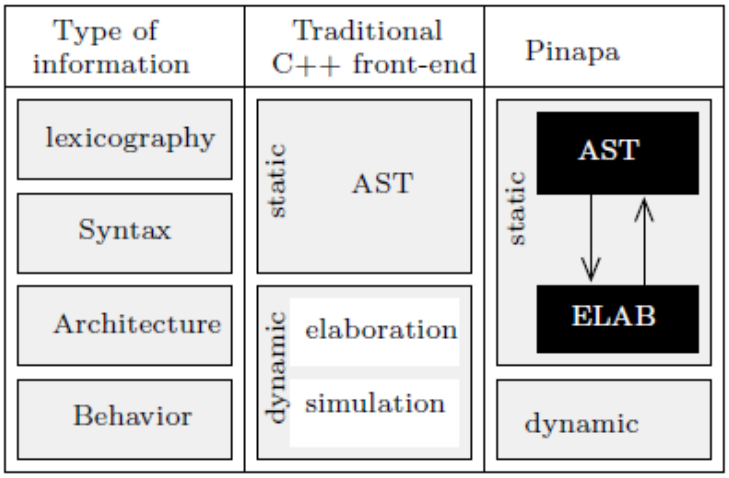

Figure 2 - Static and Dynamic information in a SystemC program. Source [9].

Although Pinapa does not have any limitations on the AST and ELAB, it presents some limitations regarding the fact that are unable to deal with pointers and references to SystemC objects and the use of templates can sometimes be problematic.

Pinapa is no longer an active project, but another project emerged following the same ideas, the PinaVM.

2.2.2. PinaVM

PinaVM [10] is an open source SystemC front-end, with the purpose of making formal and symbolic verification of SystemC programs. It can also be used for other applications quite easily. PinaVM was based on Pinapa innovative approach, reusing and introducing some new ideas to overcome the limitations that Pinapa presented.

To deal with C++, PinaVM reuses an existing C++ front-end, the LLVM, and executes the elaboration phase to obtain the system architecture, something made for the first time by Pinapa. To get rid of the limitations experienced by Pinapa, PinaVM uses the LLVM framework and its LLVM JIT compiler instead of the GCC. Therefore, PinaVM proved to be a tool for easy installation and use, supporting advanced C++ constructors that can deal with the complexity of the code in the elaboration phase. PinaVM became the first front-end to have as output a representation of the code in SSA form, the LLVM bytecode. At this time PinaVM is being update in order to be compatible with LLVM-3.2.

2.3. LLVM

Low Level Virtual Machine (LLVM) [5] is a compiler framework that aims to make multi-stage optimizations. It allows the realization of optimizations in compile-time, link-time, run-time and idle-time between runs [11]. It emerged in 2000 as part of Chris Lattner master thesis at the University of Illinois. Today, it has a very active community and, since 2005, Apple is his major sponsor. It is an open source framework, with a BSD-style license. This type of license allows a lot of freedom in its use, both in open source projects and in proprietary projects.

LLVM code is developed in C++. It is not only a compiler but a modular, flexible and reusable compiler infrastructure. This approach is quite different from traditional compilers. It allows the use of separate modules when constructing several other tools. Due to this fact, LLVM can be extensible and it is easy to incorporate with other tools. This innovative approach is faster than GCC and consumes less memory [12] [13].

2.3.1. Structure of LLVM

LLVM has a classical architecture of three-tier, as illustrated in Figure 3. It has several front-ends directed for several languages, one middle-end where code optimizations occur and multiple high performance back-ends (target-specific code generators).

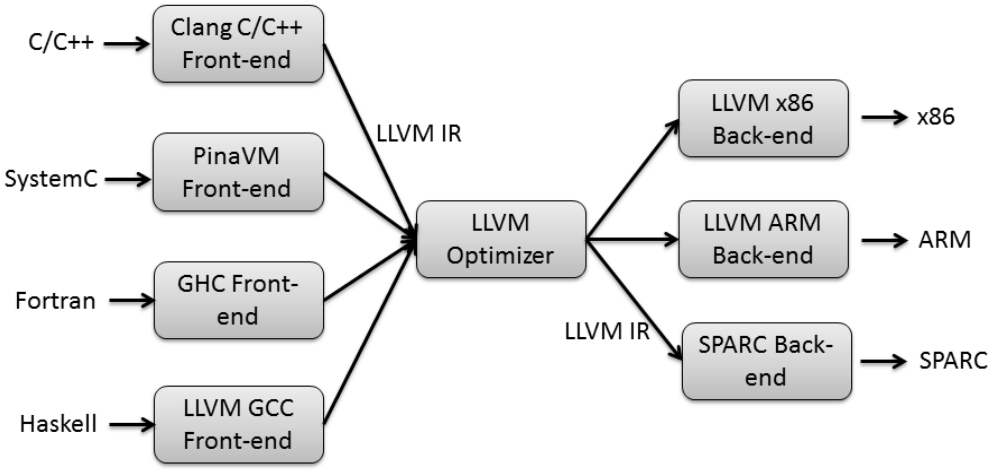

Figure 3 - Basic LLVM architecture.

LLVM per se does not include front-ends for source languages. All LLVM tools have as input, files of type .ll or .bc. Nevertheless LLVM development team has developed two frontends: LLVM-GCC and Clang. Due to its modular structure it is possible to use the middle-end as a single component.

2.3.2. LLVM IR

LLVM is an assembly language works as an intermediate representation (IR). LLVM IR is a portable low-level language that stores all the necessary information from the program. LLVM IR is fully independent of the source code [13].

LLVM IR bytecode is in static single assignment form (SSA). SSA form means that each variable is assigned only once to a virtual register, leaving immutable throughout the lifetime of the program. Whenever changes occur in the original variable already stored in a virtual registry, we will create new virtual registers that work as new versions of the initial variable.

SSA form, allow us to perform: standard scalar optimizations (e.g., dead code elimination, constant propagation), aggressive loop transformations (e.g., unrolling), and advanced inter-procedural optimizations (e.g., inlining).

LLVM IR can be represented in three equivalent forms: binary, in-memory and assembly form (text). Therefore we can store multiple formats on the disk, of the same code.

LLVM IR supports various data types: data types independent of the source language, like primitive types such as Integer, Floating Point, Void and Label; and derived types that let us implement high level structures when combined.

2.3.3. Structure of a Program

LLVM code is defined in a module. Modules have four distinct parts: meta information, external declarations, global variables and function definitions.

The meta information stores information regarding the alignment and size of various LLVM types to the target-architecture code. It also sets the endianness of the module. In global variables, it is stored in a set of pointers that point to data that has global scope. Functions use the prefix @ symbol. This allows us to distinguish them from local variables that use the prefix % symbol.

In function definitions, we have the structure of each function. Each function is composed by a set of basic-blocks. Each basic-block is set up with a list of sequential instructions and only one entry and exit point. Each start of a basic block is defined by a label and ends with a conditional instruction that contains the necessary information to know what is the next basic block or basic blocks to execute and under what conditions. As an example of conditional instruction, we have the branch (br) or a return statement (ret). The left part of each instructions indicates which virtual registers (examples: %res, %1) are stored. All instructions have information about their type, so they do not infer their type.

2.4. LLVM front-ends

As mentioned earlier, the LLVM project has two official front-ends: the LLVM-GCC [14] and Clang [15].

2.4.1. LLVM-GCC

LLVM-GCC front-end is designed to support the full range of languages and platforms of GCC. It supports languages such as C/C++. Objective-C, Ada and Fortran. It uses GCC has a front-end and compiles to LLVM bytecode or assembly language depending on the choice. It is outdated and has been replaced by Clang. LLVM-GCC derives from the GNU Compiler Collection, thus inheriting some less good specs of GCC [4]. Its antiquated approach makes it difficult to work because GCC have been coming deteriorate with each new version. It consumes a lot of memory and is becoming increasingly slower. It does not support modern techniques such as JIT compilation, cross file optimization, among others.

2.4.2. Clang

Clang [15] is a front-end for C/C++, Objective C/C++ and other variants, using the LLVM as a back-end. It generates LLVM bytecode as output. It was built from scratch by the LLVM development team,

APPLE and Google. It came to replace the LLVM-GCC, aiming to be fast, modular, reusable and implements various GNU language extensions. Their approach is similar to LLVM. It has a modular library-based architecture, and a good integration with IDEs. It has a BSD-style license which makes it open source. It was written in C++ and can retain more information during the compilation, that GCC. Clang can be faster than GCC and make less use of memory. Clang is a project that is under heavy development.

3. Related work

In this Section we will talk about the state of the art related to our tool.

3.1. Description of the System

Currently embedded systems have become more complex due to market demand and a continued decrease in time-to-market. The embedded system is composed by a software and hardware component.

Traditionally, an embedded system is built separately, *i.e.*, its two components are developed and tested separately during the early stages of development.

The software component is developed in high-level languages such as Java, C/C++, among others. The hardware component is developed in hardware description languages (HDLs) such as Verilog [16] and VHDL [17].

These languages have some limitations. They do not allow fast simulation, implementation of software algorithms or exploration of the system architecture.

The verification process of the embedded system is often done manually and, for this reason, some aspects are not detected at this stage. Only in the later stages of the process, the engineers have access to the embedded system as a whole, where its components truly interact with each other. In this stage, it is often to discover several errors and perform its corrections. For this reason, it might be necessary go back a few stages on the development process, to change the necessary components. The solution for this problem is to design the embedded system in a single language since the very beginning of the development process.

It has been emerging languages, like SpecC [18] and SystemC [1] [19], which allowed the co-development of the components of embedded systems. These languages are named system-level design languages and they extend C/C++ in order to allow the description of the hardware component. They have a good acceptance in the community, since they are based on languages widely used in the scientific community.

3.2. Software Testing

Software testing is the process of verification and validation of a particular software program. This process can be used for different purposes; the most common are fault detection software and quality control.

The verification process is designed to assess the technical features requested during the execution of the program, *i.e.*, to evaluate the behavior of the program during its execution. The validation process

enables us to know whether the program meets the initial requirements. These two processes are developed to discover different system failures.

Software is abstract, so it is difficult to get a sense of faults. The steps related to software testing are much neglected in terms of time, human and monetary resources. They are usually performed only in the final stages of the development process and often done manually. Normally it is a phase that often takes more time than the actual design and implementation.

Currently there has been an increased acceptance for testing as the software systems have increased their complexity and a demand of the market for quality has been increasing ever since.

3.2.1. Test Strategies

In this Section, we discuss techniques for verifying and validating a software system. Theoretically, there are techniques that can detect all existing errors but that would take several years. In practice there are approaches that allow us to ensure that the code is free of faults.

There are three possible approaches to demonstrate that the program is correct.

3.2.1.1. Structural Testing

These tests are of the white-box type. It means that we are interested in detecting system failures associated with aspects of system implementation. These are designed so that all paths of the program in analysis are executed at least once.

This approach is impractical in certain cases when the program has loops. The total number of paths in these cases is equal to the number of paths of the program multiplied by the number of times to unroll a loop. There may be loops with finite number too large or infinite number of times to unroll. In these cases it is impractical to cover all paths.

For this approach to work in these cases it is necessary to change the number of paths to a finite number, *i.e.* the number of times suitable for unrolling a loop. There are several proposes including one in which the loop should be unrolled zero, one or two times. This approach does not cover all paths of the program, making it impossible to detect all the existing failures. Yet this path analysis is a technique widely used.

3.2.1.2. Functional Testing

These tests are of the black-box type. Here the concern is with the behaviour of the program as a whole, *i.e.*, we are interested to know if the features of the program are meeting the requirements of the initial system. The implementation details are ignored.

To perform this type of testing, the program is subjected to all possible inputs that the program should accept. Then the output of the program is compared with the one that is the expected in that situation. This type of testing does not produce 100% coverage, because in some cases the number of inputs may be a very large number.

3.2.1.3. Correctness Profs

This approach consumes many computational resources and combines functional and structural aspects. It is used in systems where a high degree of reliability is required. Examples include communications, defense, aerospace and health care. Here the system requirements are translated into a formal language and passed as inputs to the system. To apply formal methods to the program it is necessary to convert it into a structure with formal semantics (mathematics), where the behaviour of the program is recorded in a precise structure. This form allows the use of precise mathematical techniques to evaluate the system specifications.

3.2.2. Code Coverage

Structural testing is one of the three approaches that present better performance. Despite not being able to ensure 100% coverage in some cases, it stays close. To perform this type of testing is necessary to choose which structure we want to represent the program and what is the metric that we will use.

Next, we will talk about the structure considered to this tool, a control flow graph, and some of the most popular metrics for code coverage.

```
void fibonacci(int num)
{
  int i;
  int F1, F2, Fn;
  if (num <= 0){
    printf("%i is not a valid number\n", num);
  }
  else if (num == 1 || num == 2){
    printf("F(%i) = 1\n", num);
  }
  else{
    F1 = 1.0;
    F2 = 1.0;
    for(i = 2; i < num; i++){
      Fn = F1 + F2;
      F1 = F2;
      F2 = Fn;
    }
    printf("F(%i) = %e\n", num, Fn);
  }
}
```

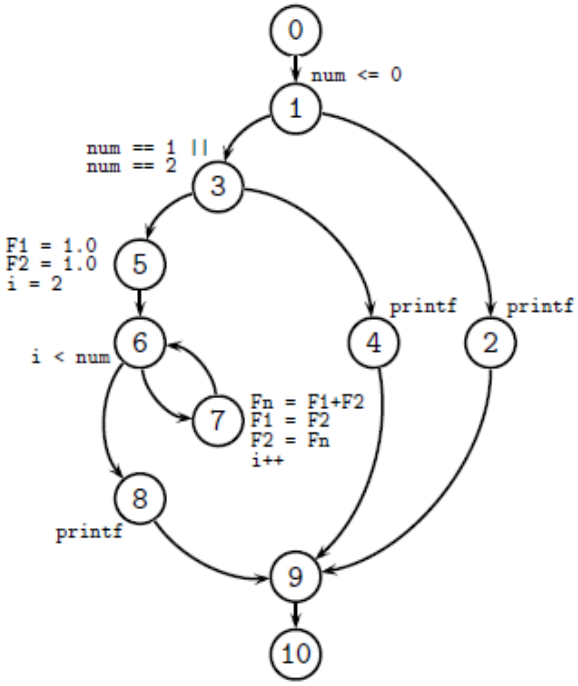

Figure 4 - C code for the Fibonacci program and its CFG. Source [20].

A control flow graph (CFG) [21] is a graphical representation of the program, frequently used for the analysis of programs as part of an intermediate representation. This abstract representation

demonstrates the behaviour of the program, making use of a directed graph where each vertex represents a basic-block or segment and edges or branches represent possible data transfers between the vertexes. Figure 4 shows an example of the Fibonacci program in C and its corresponding CFG. This CFG is composed of 10 vertexes and 13 edges.

A basic-block is composed of a set of non-conditional statements with a single entry and exit-point. Whenever a basic-block is called all his statements are executed. The last statement of a basic-block is a predicate, control loop, break, goto or exit method [21].

In LLVM IR the last instruction of a basic-block is a PHI function. PHI function stores information that allows us to know what is the next basic-block to be executed according to the information that comes to it.

Besides those elements already mentioned there are other elements in a CFG:

- Condition: is a Boolean operator in a predicate expression;
- Predicate: is the expression that contains the condition. It is used in control statements like if, for, case and others;
- Compound Predicate: is a predicate with multiple conditions.

3.2.2.1. Coverage Metrics

Several metrics are used in structural testing. The most used are path coverage, statement coverage and branch coverage.

Path coverage - this metric has as its minimum unit, the path. This coverage requires that from the entry-point to the exit-point of the program, all paths must be executed. This is the most complete metric of all structural testing method, but is often impractical to achieve 100% coverage. This is due to the large number of existing paths. Theoretically, this measure requires that the loops most be unrolled completely, which makes the number of paths too high to perform the test on them all. This approach also creates the possibility of creating unfeasible paths. In Figure 4, we can see that there are many paths due to the loop. The simplest paths are through the vertices 0, 1, 2, 9, 10 and 0,1,3,4,9,10. [20] The total number of path in this program is equal to the number of paths multiple by the times that the loop is unroll.

Statement coverage - this metric has as its minimum unit, the segment. This coverage is achieved when all the statements are executed at least once. This metric is rather weak, because even with 100 % statement coverage, there is no guarantee that the program is bug free.

In the case of loops, this metric can consider the loop completely covered with only one pass, without testing all the iterations that the loop can have.

The coverage of predicates is achieved without testing all the true/false combinations. Presents even a worst detection of errors when dealing with a compound predicate. In Figure 4 we can see that we

can achieve 100% of statement coverage, by just going through three paths: 0,1,2,9,10; 0,1,3,4,9,10 and 0,1,3,5,6,7,6,8,9,10 [20].

This coverage is also known as C1 coverage, line coverage, segment coverage, or basic-block coverage.

Branch coverage - this metric has as its minimum unit, the edge. It is achieved when every path from a node are executed at least once. This metric shows better results in finding bugs that statement coverage, because it requires that all branches be executed at least once.

It still does not fully cover the possibilities of a predicate. For a predicate with n conditions, their coverage will only be two (true/false evaluation). It handles a compound predicate as a simple predicate, thus ignoring all paths that result from it.

In the case of the program in Figure 4, we can achieve 100% of branch coverage by traversing the same three paths of the 100% statement coverage, because all edges were covered.

This coverage is also known as decision coverage, all-edges coverage, coverage or C2.

To overcome the blind spots of this coverage, there are variants of this coverage. Multi-condition coverage [22] requires that all true/false combinations of a simple predicate are covered at least once.

Loop coverage [23] requires that all loops be executed zero, one or two.

3.3. Observability Metrics

In software testing the techniques most commonly used are based on path testing [24]. This type of method is based on controllability metrics, *i.e.*, for a given input vector, there will be counted as covered all instructions of the path that was activated. This type of metric is not very reliable, because there may exist instructions on the path that produced no effect on the output.

Based on hardware testing techniques, we have metrics based on observability. For a given input vector, there are only counted as covered the instructions that influenced the output. This way, we can have a better sense of the coverage that when we only had a given input vector exercising the program.

There have been proposed several analyses [25] [26] using metrics based on observability. These analyse account for the instructions that have an effect on the output. For this, the internal variables are injected with errors and the program runs to whether such errors produce any effect on the output. This procedure is repeated for all internal variables and at the end, it is computed the coverage achieved.

The approach that we will follow in this project [20] [24] will not inject errors to the program and was based on observability analysis done in hardware models. The program is modified to be injected with functions that allow you to retrieve more information about the program. For each internal variable of the program it is kept a list of instructions that it depends. At runtime, for each variable, it is written to a

file the value assigned. With this information and with the help of a solver, it will be able to know which instructions covered. Those not registered are the ones that were not covered.

This approach is thus able to analyze a program taking into account the controllability because we know the path taken, and taking into account the observability because we know specifically what the internal variables produced an effect to the output. This approach is a good starting point for the designer chooses other input vectors that will achieve better coverage.

3.4. Validation of Embedded Software

We intend to build a tool for co-validation of software/hardware description. Several tools have been proposed [27] [28], but all have their origins in hardware or software techniques [29]. Due to this fact, they show limitations in dealing with the embedded system as one.

The most common techniques in software testing are based on path testing and have metrics based in controllability. In hardware testing, the metric is based on observability.

SystemC is a language that allows the description of an embedded system. And by using PinaVM, we can generate an intermediate representation of the source code without having to make major modifications in the source code.

This thesis solution combines hardware and software testing techniques and it is based on a validation method for embedded software described in C [20]. We extended the idea to embedded systems described in SystemC.

In [20], the solution was implemented in a framework written in C and it is composed by two components: a C parser and an Input Vector Generator (IVG).

The parser of C is based on C2C parser [30], which allows us to instrumentalize the source code, converting it to an easily manageable structure, abstract syntax tree (AST).

The instrumentalized code is passed to the next component to be manipulated by IVG.

The IVG is composed by three subcomponents:

- Longest path module, which will extract the longest path with the aid of a pseudo-boolean optimization (PBO) solver, the Bsolo [31].
- MILP module that extracts the input vectors of a given path using the lp_solve [32] a Mixed Integer Linear Programming (MILP) solver.
- The Coverage module inserts certain control functions along the path to be evaluated and execute the path with the input vectors generated previously. Then, the obtained coverage is evaluated from the generated output control functions.

The tool uses an iterative method with the following steps:

- Select the longest path;

- Determine the input vectors that allow the execution of the path. If is not possible to determine the input vectors, we know that the path is infeasible and we proceed to the next iteration;
- Execute the program using the input vectors and determine the coverage achieved (based in observability);
- If the cover observed, is not equal or higher than requested, the procedure starts again.

3.5. Verification Tools

In order to obtain the path with a given input parameters, we need to convert SystemC instructions in SMTs. It turns out that the conversion is not a linear problem, because we have to take into account the complexity of the language to convert, such as C/C++ and SystemC. In many areas there is a need to translate source code in other languages easier to manipulate.

This is the case with Bounded Model Checking (BMC), a formal verification technique commonly used to detect errors in real systems. There are several tools that implement BMC for C and C++ programs as the CBMC [33], SCOOT [34], LLBMC [35], ESBMC++ [36] and using SMT solvers.

CBMC [33] is a BMC for programs in C and C++ and supports SMT solvers, such as Z3 to validate programs. To support SystemC, CBMC uses SCOOT [34]. SCOOT is a model extractor for SystemC that can serve as a front-end for various tools such as CBMC. SCOOT uses a C++ front-end built especially for the task.

ESBMC++ [36] is a BMC for C++ programs built from the GNU C++ compiler. This way, it can take advantage of the GCC is ability to find most of syntax errors.

LLBMC [35] uses clang or llvm-gcc as front-ends to extract the LLVM IR program. Then it performs optimizations with the LLVM framework to generate a smaller and easier to handle LLVM IR. This representation goes through several transformations to be converted into SMTs. Due to the complexity and ambiguity of C/C++ language, using LLVM IR, it simplifies the program manipulation. At this stage, the C++ language is not fully supported. An example of this, are the operations with floating-point numbers.

In [37] the authors introduce a Model Checking tool for SystemC designs, which converts the source code to synchronous formalism SIGNAL [38] [39]. It uses GCC compiler to convert C/C++ to Static Single Assignment form (SSA) and then to SIGNAL.

In SMACK [40], we found a tool that converts optimized LLVM IR to Boolean intermediate verification language (IVL). SMACK allows the construction of verification algorithms based on IVL. This tool is very advantageous because it allows building tools without having to deal with the complexity of the source code. SMACK announces be compatible with various languages that are supported by LLVM front-ends, but unfortunately SystemC is not included.

All these tools generally follow the same approach: convert the source code to an intermediate representation using a front-end. Carry out, if necessary, some changes or enhancements in the

intermediate representation and finally convert the instructions for SMT or SAT format, depending on the solver that we will use.

3.6. Echo Cancellation

The system we used to test our tool is an echo cancellation system using an adaptive filter. This system can be used in a conference room to handle online communication. In a system with transmission and reception of audio the echo occurs when the received audio signal is added to the transmitted signal. The objective of the adaptive filter is to eliminate the replicas of the received signal without changing the transmitted signal. The system we used is based on the Least Mean Square algorithm to compute the filter coefficients at each moment. The system is implemented in SystemC and consists of nine modules. These modules handle the filter FIR, compute the filter coefficients and compute the filter error, among others.

3.7. Other Approaches

In this section we discuss the implementation of the developed solution in this thesis and, in addition we examine other attempted solutions that eventually led to the final solution.

Initially we dedicated our time to the study of software and hardware validation tools, syntax of SystemC and explore tools and technologies compatible with SystemC.

Following, we needed to discover a SystemC parser that would put the program in an easily manageable structure. We explored several options, but some of them didn't have any development activity in recent years or did not meet the requirements.

We tried to use the XOgastan tool, a project no longer available, which converts LLVM IR, from a C program, into XML. The installation of the tool was very time consuming and difficult. After having successfully installed XOgastan, the LLVM IR of a SystemC program was converted to a XML file. Unfortunately, the only output that we were able to retrieve was a XML file, where the XML nodes represented "undefined object type". We concluded that C++ and SystemC symbols were not recognized by the tool and to make it compatible with SystemC, it was necessary to build a tool from scratch, thus we put this solution aside.

After these failures, we tried to extract the AST using GCC, the GAWK and Graphviz. The result was an AST in a graphical format. This solution failed, because the SystemC symbols were not well interpreted.

In the end, we decided to use LLVM IR as a basis for our analysis framework. This approach corresponds to the solution described in this thesis.

For front-end, we chose PinaVM due to various factors, but especially because of the use of LLVM framework.

Installing PinaVM presented some challenges. The project was in a standstill and it was only compatible with older versions of SystemC, GCC and LLVM. It took us some time to discover the origin of certain errors. Most of those errors were problems related to version compatibility and certain deprecated libraries that were no longer used in Fedora 16.

The need to use older versions of libraries and LLVM framework formed a bottleneck of the project. To overcome this problem, we decided to make PinaVM compatible with LLVM-3.2, but meanwhile the project has been active again and it was ported by its own development team.

During the implementation we decided to change the version of LLVM, from version 2.7 to 3.2, because we were having several problems with version compatibility. This change also occurred because this version has improved debug information compared to the previews versions. The debug information can be used in the construction of tools for reconstructing the source code from the LLVM IR, among others.

4. Architecture

After introducing the study of the tools and algorithms related to this thesis, we will now present the developed solution. As noted in Section 1.1, the solution is made up of several components, some previously developed, other implemented as part of this thesis. In addition to the implementation of new components, this solution will also integrate all these components into a single tool, thus creating the *Cervantes Co-Validation Tool for Embedded Systems*.

In the following sections, we will present the tool, describing the various components and tasks.

4.1. Overview

This tool for co-validation will receive the following inputs:

- A description of the system to evaluate written in SystemC;
- A *Configuration file*, with a list of modules and its functions names. This list should be inserted in advance by the user;
- A quantitative metric referring the degree of coverage necessary to achieve. The metric is based on observability and will act as the minimum reference value to the program coverage.

The list contained in the *Configuration file* currently has to be manually inserted by the user but in the future it can be done automatically.

The tool will return coverage above or equal to this value. If it is not possible to achieve this coverage, the tool will return the higher coverage that it found.

The presented solution is based on the formulated strategy briefly described in Section 3.4. In order to apply the same strategy, we should bear in mind that in this project we want to validate embedded systems described in SystemC and not in C. To fulfill this requirement, we present a new methodology (see Section 1.2), where new components had to be implemented and incorporated with existing ones. Figure 5 shows the architecture of this project solution.

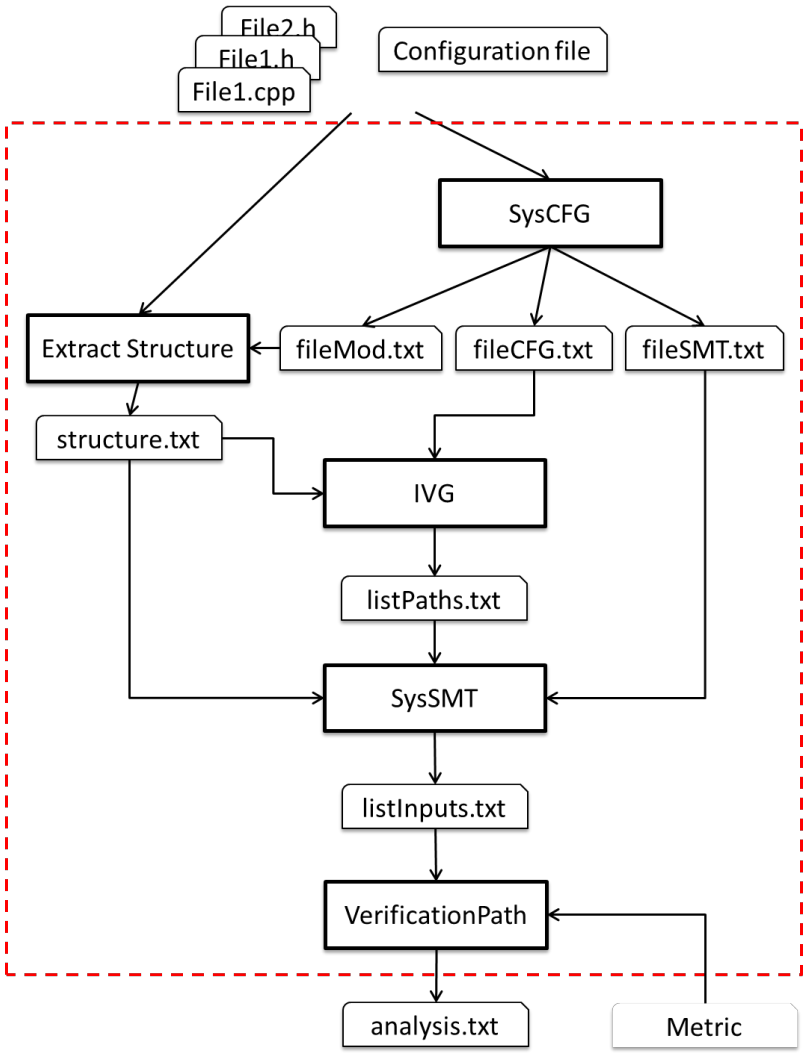

Figure 5 - General diagram of the co-validation tool.

The exchange of information between the components is done using text files with different formats. Next we will present the various components that compose this solution.

4.2. SysCFG

SysCFG (SystemC to CFG) aims to extract from a description in SystemC, the CFGs of the functions belonging to the source code. It takes as input parameters the description of an embedded system implemented in SystemC and a *Configuration File*. The information contained in the *Configuration File* will guide the **SysCFG** in generating the LLVM IR, from the SystemC description, and recognize which functions belong to the original source code.

SysCFG is composed of three components: **Front-end**, **Middle-end** and **Back-end**. In order to manipulate the source code, it is necessary to convert the source code to an intermediate representation which has mechanisms to incorporate extra information about the source code.

For that purpose, in the **Front-end**, the source code will feed PinaVM [10], a SystemC front-end, which produces the LLVM IR of the program. The LLVM IR is stored in a file "*file.bc*" that will serve as a starting point for our analysis of the source code.

In the **Middle-end**, we will perform a parse to the LLVM IR content, using the LLVM framework, in order to accomplish the following objectives:

- Catalog the instructions belonging to the source code (see Section 5.1.2);
- Convert the instructions of LLVM IR to SMTs (see Section 5.3.1).

By cataloging only the instructions of the source code, it will provide a basis to construct the CFG function and calculate the longest path. (see Section 4.4). The longest path is calculated by taking into account the number of instructions stored on each edge. At the same time, these instructions will allow to extract information to construct the SMTs (see Section 5.3.1).

Finally in the **Back-end** we will build the CFGs (see Section 3.2.2) related to the source code functions.

At the end, three separate files are generated with necessary information to feed different components. The "*fileCFG.txt*" stores the detailed structure of the extracted CFGs. The "*fileSMT.txt*" stores the SMT assertions extracted and organized by edges. Finally the "*fileMod.txt*" stores the list of functions extracted with their respective Identifiers (IDs).

4.3. Extract Structure

Assuming that previously we extracted the CFGs of the source code functions, the construction of the C/C++ program CFG is made starting by the main function of the CFG. Then it replaces the vertexes related to the source code function calls, by the corresponding CFG of the function. This substitution is made in the body of all functions of the source code.

However, our solution must deal with SystemC and the construction of the CFG from a SystemC program is not possible using the algorithm described above.

This is due to the structure of the SystemC library. A SystemC program also has a main function called sc_main, where are declared the system components, like, signals, modules and relations between signals and modules. In sc_main are also declared some instructions of the SystemC library to configure and initialize the system simulation. It turns out that the module functions that simulate the respective module behavior are not called directly in any part of the sc_main function of the program, unlike in C/C++. The SystemC contains a component called simulation kernel [19] that encapsulates part of the program where the signals are modified and where it is decided which module functions runs at a given time and in what order. Since we do not have access to the simulation kernel, we cannot get the necessary information to know when the module functions are called.

To work around this obstacle it was necessary to build the **Extract Structure** component that aims to extract the list of program signals, program modules and their relationships from the source code. At

the same time, it extracts the list of #define directives that it finds. This information will be passed to the **IVG** component with the intention of complementing the information already extracted by **SysCFG** so that the generation of the program CFG may be possible. To construct the CFG of a C/C++ program, the main function is the starting point and it guides the program flow. In our solution, the starting point is the sc_main function and the program flow is determined using the information extracted by the **Extract Structure**. It will be from the signals, modules and their relationship that we will determine the order in which the CFGs functions are called. For more details on CFG construction algorithm of a SystemC program, see Section 4.4.

If the constants defined in #define directives appear in the SMTs assertions, we will replace them with the associated value (see section 5.3). The extracted information is stored in a text file so that it can be passed to the **IVG** and **SysSMT**.

This component is part of the scope of this thesis and its implementation is described in detail in Section 5.2.

4.4. IVG

Input Vector Generator (**IVG**) is an imported component of the solution created by Costa at [20] and briefly described in Section 3.4. This component has been changed by its author, to make it possible to integrate with our solution.

Next we will briefly describe its new functionality. **IVG** now aims to generate a file containing a list of paths that cover the entire graph ordered by the longest path. Figure 6 shows the architecture of **IVG**. The **IVG** consists of three components: **GenerateCFG**, **GenerateDAG** and **GeneratePaths**.

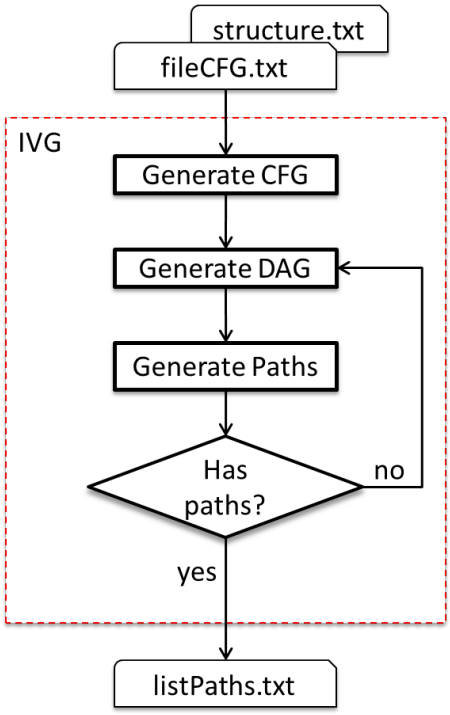

Figure 6 - IVG general diagram.

GenerateCFG receives from the **Extract Structure** a file containing the static structure of the program and another file containing the CFGs of the source code functions from **SysCFG**. With these information's the **GenerateCFG** will build the CFG of the program.

In order to generate the program CFG, we start by the sc_main function CFG and replace the vertexes regarding to the source code function calls with the corresponding CFG of the function called. The same approach is performed recursively to all functions called from sc_main.

In SystemC, the module functions are never called, only the SystemC simulation kernel knows at what time and when to call the module functions. Thus the methodology previously used (see Section 3.4) does not work for the module functions.

So to complete the CFG program, we need the information about the static structure of the program extracted by **Extract Structure** from the source code. This will allow inferring the relation between the modules through the interconnected signals. With this information, first we build a graph of the system. In this graph each node is a module and each edge is a signal between modules.

When searching for an execution path in a system we only want to execute an instruction after the variables that are used in that instruction were already assigned. Thus to execute a module in SystemC we need its input signals to be already assigned. This leads to the use of a topological sort in the graph of the entire system. The topological sort gives us a list of modules where for a certain module in the list, its input signals were all assigned in modules that are before in the list.

After obtaining the list of modules in a topological order we build a CFG where all module CFG's are connected in a sequential manner starting with the first module in the list and ending in the last one.

The purpose of this CFG is to have a graph that can be searched for execution paths of the entire system.

In the **GenerateDAG**, we want to extract one directed acyclic graph (DAG) [41] [42]. The difference between a CFG and a DAG is that a DAG does not admit cycles. Initially the loops are unrolled one time. There may be cases where, in advanced stages of the analysis, every path found is not feasible and/or with an acceptable coverage. In such cases, the loops must be unrolled again. This requires the DAG to be rebuilt.

For the construction of the DAG [43], the CFG will be traversed, and whenever a loop is found, they are unrolled once. This means doubling the vertices corresponding to the body of the loop. Thus we have the representation of the program in a DAG.

The **GeneratePath** component will be used to implement the algorithm to find a list with the longest paths. This algorithm [43] will be applied when this module is called. Each statement observed in the generated path, it is marked as analyzed in the DAG for future reference. This way the path found is the longest taking into account only the remaining part of the DAG that is not marked. A new path is generated taking into account only the part of the DAG that was not observed in previous paths.

The goal is to find a minimum set of paths that obtain the highest possible coverage of the program. The coverage attained is based on the observability and to increase the probability to find it, we consider the paths with the highest number of statements still to observe.

In extreme cases it may happen that after analyzing all the DAG, it has not yet been found a feasible path and/or achieved coverage. When all paths have been analyzed, this module will send information to the module **GenerateDAG** so a new DAG is generated taking into account the stored information of the previous and the loops are unrolled again. This approach requires a directed graph for the representation of the entire program.

At the end of the algorithm a file is created: "*listPaths.txt*", containing for each path found, their respective edge list.

4.5. SysSMT

SysSMT aims to extract from a given path, its input list. For that, it receives three input files:

- "*structure.txt*" contains the #define directives that will be used to replace some of the constants that may appear in the SMT assertions by the corresponding value. This is used so the Z3 solver does not consider these constants as variables when trying to find the input parameters of a path. (see Section 5.3.2);
- "*listPaths.txt*" contains the paths that cover the entire DAG of the program, ordered by size, starting with the longest path. These paths are made of their edges list;
- "*listSMT.txt*" contains the list of all the edges of the program and their SMTs assertions.

In software testing, to generate a set of input vectors that exercise a particular path is a very complex problem. For the automatic generation of paths, there are methods based on symbolic execution [44], dynamic execution [45] or using a combination of both [46] [47]. Using these approaches, we can extract a lot of information from the program.

In our approach, the generation of the input vectors is handled by the **Z3** [3]. This is a solver based on Satisfiability Module Theories (SMT). It will allow the modelling of most of the expressions, without having to make major changes to the path under test. This approach will speed up the process in comparison with the approach in [20] and can deal with arrays indexed by variables.

A path is feasible if the solver used can produce input vectors to exercise that path. **Z3** will receive a list of instructions covered along the path and it will give as output a set of input vectors that exercise that path. If the **Z3** cannot produce the input vectors for a given path then we will know that this path is not feasible.

SysSMT was created within the scope of this thesis and its implementation is described in detail in Section 5.3.

4.6. Verification Path

When reaching the **Verification Path** component, we have a list of feasible paths with their respective input parameters and the coverage metrics to be achieved. This information comes from "*listInputs.txt*" file, created by **SysSMT** component and the coverage metrics is entered by the user. This component is intended to calculate for each feasible path, if it has an observability-based coverage [48] equal or greater than the requested. This methodology is vital with functional testing approaches in systems where requirements specify code coverage.

First it was created a library of functions written in SystemC. These functions will register the memory addresses of the instructions used during execution of the source code using static code instrumentation. The purpose is to monitor the use of all memory addresses and source code lines used during execution.

These library functions are added to the source code manually but in the future it may be done automatically, so that all the variables in source code become monitored. To test each discovered feasible path, we execute this new version of the source code using the input parameters produced by **Z3**.

At the end of each execution, we know which lines of source code were executed in order to know whether the path that we expect to analyze was in fact analyzed. In parallel we determine the coverage metrics from the list of memory addresses that were used during the execution of the source code. Finally, this module will generate a report specifying the details of each execution. Each report will contain the following information:

- Metrics achieved, taking into account the instructions of the travelled path (controllability);

- Metrics achieved, taking into account the instructions which caused some effect on the output (observability);
- Input vectors required to traverse the path.

Although it returns two metrics, the metric that will be considering as reference is the one based on observability.

5. Implementation

In this section we describe the implementation of the **SysCFG, Extract Structure** and **SysSMT** components.

5.1. SysCFG

In this section we will present the implementation of the **SysCFG** component. The **SysCFG** aims to extract the CFGs of the source code functions from a description in C/C ++ or SystemC. The **SysCFG** architecture is illustrated in Figure 7. To make **SysCFG** a modeling tool, we decided to implement the classic three-tier architecture inspired by the LLVM framework architecture. The **SysCFG** consists of three components, entitled **Front-end**, **Middle-end** and **Back-end**.

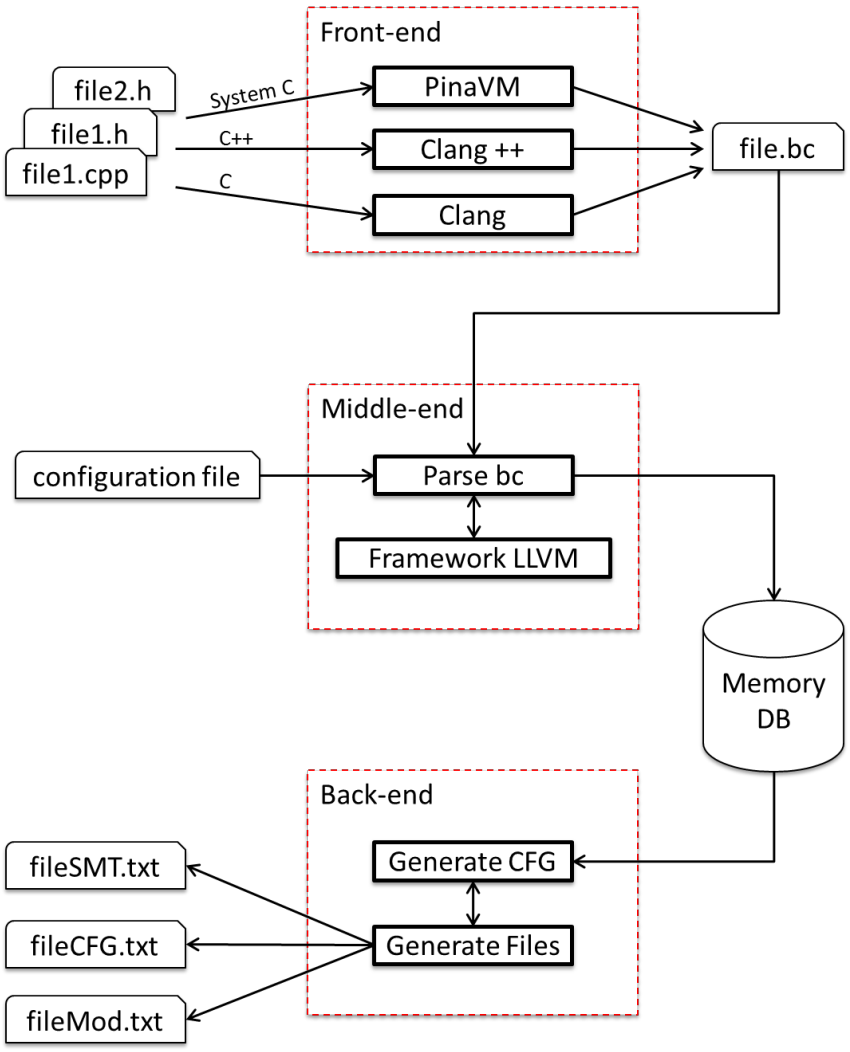

Figure 7 - General diagram of SysCFG.

The **Front-end** component has the ability to extract a LLVM IR from the source code of the program to be analyzed. LLVM IR is a low-level language similar to assembly. Depending on the source code

used as input, either C/C++ or SystemC, we can choose the appropriate front-end to convert the language to the same format, the LLVM IR. The LLVM IR will be stored in the file "*file.bc*" that will be the analysis target in the next component.

The **Middle-end** aims to parse the LLVM IR previously generated and extract all the information needed for the construction of CFGs related to the functions of the source code, using the LLVM Framework. At the same time, it translates instructions to SMTs. Due to some factors, currently, instructions in LLVM IR are stored in a strings. We will detail this issue later in Section 5.3.1.

Since our tool just want to rely on the functions and instructions of the source code, it is necessary for the **Middle-end** to learn to distinguish the source code functions from the C/C++ and SystemC library functions. This way it is necessary to the user to input a configuration file. This file has the name of the modules and their functions, which compose the program in analysis.

To explore the structure of the LLVM IR we used a Depth-first search algorithm. This component information will be stored in a format similar to the LLVM IR, being organized by: Module, Functions, Basic block and Instructions. The big difference is that instead of storing all instructions, we will only store the instructions belonging to the source code. This information will be passed to the next component via the Memory DB, which is a set of classes in memory that store information.

The **Back-end** component aims to make the parsed information stored on the Memory DB and change its structure to create the CFGs of the source code functions. To save this information in a structured form, we used the file format that IVG component accepts as input. This file is named *"fileCFG.txt*". Additionally, two other files were created: the **"***fileSMT.txt*" that stores all edges with the respective instructions in LLVM IR format and the "*fileMod.txt*", which stores the list of the functions corresponding to the source code and its respective IDs.

Next we will detail the operation of each of the three components. During the description of **SysCFG** we will rely on examples of C++ to demonstrate some features of our parse. We will avoid the SystemC examples because of the large proportions of the code size, since the examples would be too large only to explain some concepts. However, as SystemC it is a C++ library, the behavior is quite similar.

5.1.1. Front-end

First we must convert the source code in an easily manageable structure (intermediate representation) in order to extract the information easily. There are several types of tools for extracting information from a source code to an intermediate representation. For example, the BMC tools in Section 3.5.

The choice often depends upon the intermediate representation of the source code to be analyzed. In our case, SystemC narrows the choices, so we chose LLVM IR as the intermediate representation to be used in this project.

Depending on the language of the source code (C, C++ or SystemC), the front-end used is different. For the source code in C and C++ we use, respectively, the Clang or Clang++ as the front-end. For

SystemC used PinaVM. Initially it was not expected for SysCFG to be able to be used with C/C++, since the tool goal was to work only with SystemC, but due to the fact that the construction of SysCFG was made initially using examples of C and C++, before using SystemC, we decided to keep this feature. All front-ends mentioned use the LLVM framework as back-end.

The decision to extract the CFG from the LLVM IR brought to this project several advantages. Among them we highlight the access to the code to examine in LLVM IR and by using the LLVM framework, it allows us to access code in a structured form. The major disadvantage is that to extract exactly the same number of source code instructions, we had to create a complex parse to this purpose, which we will detail in the **Middle-end** component.

PinaVM was the SystemC front-end chosen. Based on our research, PinaVM is in fact the best SystemC front-end in the market. It has several advantages: it is open-source, it uses the LLVM back-end and it can overcome most of the limitations that other front-ends of SystemC present in [8].

The PinaVM is a project that is composed of a front-end and three back-ends, each with different goals. The PinaVM back-ends in the version used by us have some limitations when dealing with some source code features. Each back-end presents different limitations. Of the examples provided with the PinaVM, most runs successfully using the front-end and just a few run successfully in the back-ends.

The echo cancellation program where we intended to test the tool, it did not run successfully in the PinaVM back-ends. From this fact we decided to work with the LLVM IR generated by the front-end PinaVM.

When we started working with PinaVM, it was a stalled project and only compatible with older versions of LLVM and SystemC, something that caused some complications in its use and installation. When running the source code with the PinaVM, only its front-end worked. Again we could not find sufficient documentation on their back-ends. We suspect that is due to the way the echo cancellation system has been implemented. The implementation of a system in SystemC may be performed using different levels of abstraction. We suspect that the PinaVM back-ends were only prepared for some levels of abstraction and the implementation used in the echo cancellation was not recognized by the back-ends. So we decided to use the LLVM IR generated by PinaVM front-end as a starting point. This decision made it possible to use the SysCFG for C, C++ and SystemC, because the LLVM IR generated by the C/C++ and SystemC front-ends are similar in content and structure. Early in the implementation of our solution, the PinaVM became again an active project and suffered a lot of changes to be compatible with newer versions of LLVM and PinaVM.

Working with this version of LLVM IR has some advantages for our analysis. One of the main new features is the access to debug information that comes in LLVM IR. The debug information corresponds to the Metadata information that brings the necessary information to be able to extract the LLVM IR structure and the information of each source code instruction.

At the same time during the implementation of our approach, the big difference we found between generate LLVM IR from PinaVM or other front-ends, it was that, it didn't generate additional C++ and SystemC instructions, not directly linked to the source code instructions. We could not find documentation that would allow us to understand why they were being generated. We suspect that it is due the fact that the PinaVM interpret SystemC as a language and not as a C++ library. For that reason, it performs some optimizations to be able to generate a LLVM IR similar to the one coming from C or C++. These optimizations may have been made with the help of some existing parses in LLVM Framework or by some changes made by PinaVM.

Our developed parser works equally well for C/C++ and SystemC when we use the LLVM IR created from Clang, Clang++ and PinaVM respectively. If PinaVM was not used, the parse would have to be adapted specifically for SystemC and we would not ensure that we would be able to extract only the instructions from the source code, mostly because we would have to find some way to remove the C++ and SystemC instructions that PinaVM already eliminates.

5.1.2. Middle-end

Following we will describe in detail the **Middle-end** component. This component receives the "*file.bc*" in which is represented the LLVM IR corresponding to the source code. The objective of this component is to select and catalog the functions and instructions belonging to the source code. At the same time we intend to rebuild the instructions and move the contents of the instructions to a structure that allows in future converting each instruction into SMT format (see Section 5.3.1). To achieve these two objectives, we used the LLVM framework in order to access the module structure and get more details about the LLVM Metadata.

To further explain the process that occurs in the **Middle-end**, we will start by describing the control structures used to ensure proper extraction of LLVM information (see 5.1.2.1). Next, we describe how the data is stored in LLVM IR and what are the main features of the data structure we intend to build with information extracted from the LLVM IR (see 5.1.2.2). In 5.1.2.3 we present the instructions that our parse considers relevant in LLVM IR in order to be able to select the instructions of the source code. Then we describe the operation of *CompleteInstruction* function (see 5.1.2.4) that aims to iterate the LLVM IR instructions seeking information aimed at complementing the information stored in the relevant instructions. Finally, in 5.1.2.5, we will present the main algorithm of our parse where we explain how the concepts presented in the previous sections interact.

5.1.2.1. Control structures

For proper cataloging the source code instructions, we start by describing in this section the necessary control structures implemented to ensure correct implementation of our parse. These structures are designed to ensure that are analyzed:

- All functions of the source code;
- All basic blocks;

- All connections between basic blocks;
- All basic blocks instructions.

5.1.2.1.1. Parser Module

The first structure that was built was a list containing only the source code functions. Regarding the functions, in a LLVM IR referring to a SystemC program, we have *declare* and *define* functions. In *declare* functions we only have access to the function declaration. In *define* functions we have the corresponding function body. When the source code is in C, only the source code functions are *define* functions. As SystemC is a C++ library, in the case of C++ or SystemC source code some C++ and SystemC library functions also appear as *define* making them indistinguishable from the source code functions. This is one of the reasons why the LLVM IR becomes very large even for small programs in SystemC. But since we do not intend to explore the C++ and SystemC library functions, we only want to select the source code functions. An example of one of these functions is the *log* function. This function is shown in LLVM IR as a *define* function and through the list we built with the source code functions, we know which functions to ignore, such as the *log* function.

To select the functions we need to have other considerations in mind. Compilers use a technique called name mangling, which is used to add semantic information to the name of the structures, such as functions, classes and structures. This technique renames all objects names in the LLVM IR with semantic information encoded in the name of the objects.

When analyzing the LLVM IR we discovered a pattern in the names of the source code functions. The modules function names contain the original module and function name. The functions related to the constructor of the module contain the original module name and the "*sc_module_name*" string. Other functions related to parts of the module, contain the name of the original module.

In sc_main function, when a constructor of a module is called in LLVM IR that call is converted to a chain of function calls until reaching the function containing the constructor body. These functions that are called in chain follows the same naming scheme changing only the numeric value in the name, which means that they will be called in sequence.

In most cases the functions that are between the sc_main function and the function containing the constructor body include only two instructions: (i) the next function call; (ii) return statement of the function. For each module function that exists, there is one function in LLVM IR.

Taking the example of the echo cancellation system, it contains a module called memoria_hw and the module function is called access. The function related to the constructor call, called by sc_main, is called *_ZN10memoria_hwC**1**EN7sc_core14sc_module_nameEj* and it calls the function that contains the source code instructions, *_ZN10memoria_hwC**2**EN7sc_core14sc_module_nameEj*. The only difference in the name of these two functions is the numeric value of 1 to 2. The module function is called *_ZN10memoria_hw6accessEv* and in the examples tested by us there is only one LLVM function per function module.

So, to select the functions of the source code, we will use the contents of the Configuration File, which contains the name of all source code modules and the module functions name. This list does not need any prior sorting. Additionally to that list, the program also considers other two strings by default: "*main*" and "*sc_module_name*". The "*main*" string is used to choose the sc_main or main function and the "*sc_module_name*" to extract the functions related to the constructors of the modules.

The program will go through the LLVM IR and select source code functions containing the default names mentioned above. Then it generates a list containing only the name of the source code functions generated by the LLVM using name mangling.

5.1.2.1.2. Parse Functions

Thus having a list containing the names of functions to analyze, it is now necessary to analyze the structure of each function. A function contains several basic blocks, which contain sequential instructions and only one entry and exit point. The last instruction of each basic block corresponds to a conditional statement that contains the necessary information to know what is the next basic block or basic blocks to execute and under what conditions. In LLVM IR a conditional statement refers to a condition, cycle or the end of a function.

During our analysis to the body of functions it was necessary to ensure that all of its basic blocks and their connections were inspected while preventing the entrance in loop. During implementation we consider the function, in the LLVM IR structure, as a graph where basic blocks represent their vertexes and the links between them, the edges. The algorithm used to explore the structure of a function is based on depth-search algorithm.

It turns out that the source code may arise recursive cycles that influence this analysis. Take Figure 8 as a simplified graph of a while loop.

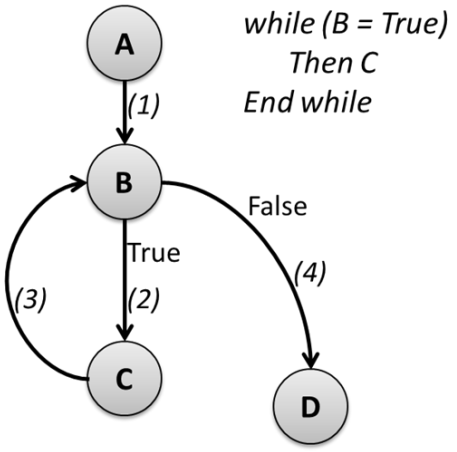

Figure 8 - Simplified while loop graph.

By analyzing the graph in Figure 8, we see that if we do not ensure some rules as stopping points, we will fall into an infinite loop during our analysis.

If, for example, we consider that we have no stopping point, while iterating through the graph we would get in an infinite loop between the vertexes B and C and our parse would only make the following analysis: A, B, C, B, C, B, (C, B…). It should be noted that when analyzing a conditional instruction, the parser always analyzes the vertex that corresponds to the true condition, and in this case, after the vertex B is the next C.

So we created two basic structures that prevent us to enter an infinite loop and ensure that each graph is fully covered (cover all vertexes and edges). The first data structure will store the basic block names which are being examined. The second stores the ordered pair of basic blocks. The registration of basic blocks will ensure that all vertexes are analyzed and registration of a pair of orderly basic blocks serves to ensure that all the edges are covered.

If, for example, we would only use the first structure that stores the name of basic blocks and iterate through the graph we would get a full coverage of vertexes: A, B, C, D, but we would not ensure the full coverage of edges. Upon arriving at the vertex C the parser would analyze the last instruction of C and checks that the next vertex is B. It considers the path as analyzed and from the vertex B, it determined that the next vertex to be analyzed is D. The problem is that the edge (3) is ignored and in the algorithm for construction the CFG function, this edge (3) is not registered.

So, before starting a scan of basic block content, we consult the data structure to see if the actual basic block was analyzed. If it was not, we analyze and store the name basic block into a list. Consequently we keep the ordered pair of basic blocks made by the previous basic block and basic block we have just analyzed. If the basic block has already been analyzed, we analyze whether the current pair of basic blocks has already been analyzed. If not, we record the edge.

This way, we will ensure coverage of all vertexes and of all edges: AB, BC, AC and BD. As we consider the edges as ordered pairs we must cover all the edges even if they are composed of vertexes already analyzed.

This data structure is stored only in memory during analysis of each function and it only serves to ensure full coverage of the function.

5.1.2.1.3. Parse of Basic block

During our algorithm to catalog the instructions contained in each basic block, we will see that it is necessary to make several parses of the same basic block to ensure that all LLVM IR instructions are covered following a certain order, as we will describe the further ahead.

Due to the recursive algorithm we have implemented to catalog and extract information of the instructions, we need to create a structure to store in an orderly manner the position of each statement that our parse covered, to keep the record of basic block instructions that were accessed during the parse and which weren't accessed. As the parsing is performed on the code several times, this structure prevents errors that may arise in the case that one or more instructions are covered more than once. At the same time we used the structure to know when all the basic block instructions were

cataloged, because as we know the total number of instructions in the basic block, in the end we will have the same number of positions in the list.

Take for example the Figure 9 and Figure 10. The Figure 9 shows a simple C++ code for the purpose of adding two values and Figure 10 represents the corresponding LLVM IR generated of that same function, containing only a single basic block. In this example we know that our parse finished its analysis when we have an array with 19 positions indicating that all LLVM IR instructions were covered.

```
1:  int main(int argc, char *argv[])
2:  {
3:          int a;
4:          int b = 10;
5:          int c = 12;
6:          a = b + c;
7:          return 0;
8:  }
```

Figure 9 - Adding two values, C++ code.

```
1:  %argc.addr = alloca i32, align 4
2:  %argv.addr = alloca i8**, align 4
3:  %a = alloca i32, align 4
4:  %b = alloca i32, align 4
5:  %c = alloca i32, align 4
6:  store i32 %argc, i32* %argc.addr, align 4
7:  call void @llvm.dbg.declare(metadata !{i32* %argc.addr}, metadata !910), !dbg
    !911
8:  store i8** %argv, i8*** %argv.addr, align 4
9:  call void @llvm.dbg.declare(metadata !{i8*** %argv.addr}, metadata !912),
    !dbg !911
10: call void @llvm.dbg.declare(metadata !{i32* %a}, metadata !913), !dbg !915
11: call void @llvm.dbg.declare(metadata !{i32* %b}, metadata !916), !dbg !917
12: store i32 10, i32* %b, align 4, !dbg !917
13: call void @llvm.dbg.declare(metadata !{i32* %c}, metadata !918), !dbg !919
14: store i32 12, i32* %c, align 4, !dbg !919
15: %0 = load i32* %b, align 4, !dbg !920
16: %1 = load i32* %c, align 4, !dbg !920
17: %add = add nsw i32 %0, %1, !dbg !920
```

```
18:  store i32 %add, i32* %a, align 4, !dbg !920
19:  ret i32 0, !dbg !921
```

Figure 10 - Adding two values, LLVM IR.

Therefore our parse is complete when all positions of the instructions are covered.

5.1.2.2. CFG structures

To better understand some of the decisions taken here, it is necessary to explain briefly how instructions are structured in LLVM IR and which format we want to get for each source code function.

As previously mentioned, the LLVM IR stores all information on a Module, which contains several data blocks, including the function definitions (see Section 2.3.3). Our parse will use the block of the function definitions to extract information relating to the functions. Alongside each instruction contains debug information that will help rebuild the source code instructions information.

As mentioned earlier, functions contain basic block and these basic blocks consist of sequential instructions where the last instruction corresponds to a conditional instruction, which contains enough information to know which the following basic blocks are. The conditional instruction, also classified as terminator instruction by LLVM, can be originated from one of the following instructions from the source code:

- Condition (if);
- Cycle (while, for);
- End of the function (return);

The functions in LLVM IR are organized in a graph. This graph has the following characteristics: each basic block corresponds to a vertex and stores a block of instructions and links between basic blocks represent the edges.

To build the CFGs necessary to the **IVG** component, we must follow some rules to be able to build the program CFG. The structure that we pretend to generate follows the rules defined in [20].

The new CFG require us to add another conditional instruction to the above list: the function calls. Function calls can refer to either the source code functions or C/C++/SystemC library functions. It is also necessary to add vertexes when a conditional statement appears or at the end of a condition, cycle and at the start of a function.

The construction of the program CFG is obtained covering the instructions and whenever we find a conditional instruction, we create one vertex. This vertex will be associated with a numerical identifier (ID) that represents the type of conditional instruction that led to it. Table 1 show for each type of conditional instruction, its value.

ID	***Vertex Type***
0	start of function

1	end of function
8	start of if
9	end of if
16	start of ifelse
17	end of ifelse
24	start of for
25	end of for
32	call to function
40	start of while
41	end of while

Table 1 - Conditional instructions.

The vertexes do not store any instruction, instead the sequential and the conditional instructions are stored in the previous edge going to the newly formed vertex.

5.1.2.3. Relevant instructions

When passing the SystemC source code to LLVM IR, some source code instructions are split into multiple LLVM instructions. In our approach, in order to try to get just the instructions in the source code, it was necessary to make a parse to the LLVM IR and search for a specific set of instructions and select them in a certain order. Unfortunately, our approach could not parse all the instructions that the LLVM library offers. So, we had to create a parse to guarantee the construction of a CFG per function, wherever the source code is in C, C++ or SystemC. So we focus our efforts in covering all conditional instructions referenced in Table 1 and ensure that all instructions in a function are covered, whether known or unknown. As we ensured the coverage of conditional instructions, we can say that the remaining instructions are only sequential instructions and they do not affect the structure of our CFG.

Our parse focuses on the selection of the following instructions:

- Variable declarations;
- Function calls;
- Store instructions (memory access);
- Terminator instructions;

A source code instruction may correspond to multiple instructions in LLVM IR. Thus from the instructions set above, it was necessary to inspect other instructions which might contain the remaining portions of the original source code instruction. To this end, we created *CompleteInstruction*

function. This recursive function extracts the rest of the instruction. We will describe its operation later (see 5.1.2.4).

Depending on the instruction that we have to analyze, our parser in some cases, not only covers the instruction, but rather a group of instructions marking their positions in the control structure described in (see 5.1.2.1.3). This prevents that in further parses we analyze the same instruction twice.

The declaration of variables is identified by the call to the function *llvm.dgb.declare*, which is composed by two arguments. The first indicates the type and name of the variable and the second the value of his initialization, if applicable. The variable initialization can be performed in several ways: (i) initialize a variable with a value, (ii) return value of a function call, (iii) result of an arithmetic operation and others. To identify the type of initialization, it is necessary use the *CompleteInstruction* function that will allow discover the value of the variable initialization by exploring other instructions.

The function calls are identified by the LLVM call instruction. This instruction allows us to locate the source code calls, the C/C++ or SystemC library calls and the instantiation of structures and classes. The call instruction contains as arguments the returned value, the calling function name and the input parameters. As the input parameters of a function are variables, we used the *CompleteInstruction* function that will recursively identify the content of each parameter.

There are various statements related to memory access. Our parse only considers as a priority the identification of the store instruction. This refers to instructions which store a value in a particular memory address, *i.e.*, a variable. The value may result from different operations such as arithmetic, equal to a value or the return of a function call, among others. Again, the identification of the value to be stored is done by the *CompleteInstruction* function.

Finally we have the terminator instructions. These instructions are classified by the LLVM as instructions that mark the end of basic blocks. In LLVM there are several types of terminator instructions that require different treatments of the information they store. We will only briefly describe the treatment of conditional branch, return and invoke instructions.

When conditional instructions, such as if, for or while are found in the source code, they are converted to the branch conditional instruction of LLVM. In these instructions we must extract two different objects: conditions and labels. The condition to be tested will allow us to decide what the next basic block is. To extract the condition to be tested it is necessary to use the function *CompleteInstruction* since the condition is always stored in another instruction or instructions. The labels indicate the name of the following basic block when the jump condition is true. If it is true, the next basic block is displayed on the first label. If condition is false, the second label indicates the following basic block.

The return instruction means that there are no further basic blocks to consider in this route. This instruction marks the end of a function. We just have to extract the argument returned by the statement using the *CompleteInstruction* function.

The invoke instruction appeared during our analysis of the LLVM IR originated from SystemC. Theoretically to declare a signal in SystemC is like declaring a variable with a given type. It turns out

that in LLVM IR the declaration and use of variable types derived from SystemC library in LLVM IR are converted to the invoke instruction. The invoke instruction is a complex instruction, with different arguments. This instruction contains a call statement to a function and two labels. The first label called “normal label” corresponds to the basic block name to execute, if the called function returns to its normal execution flow. If the function returns any execution that occurred during its execution, then the program flow follows the second label, called “exception label”. This will direct the program flow to several basic blocks with instructions outside the source code.

This flow may be driven by a basic block sequence of SystemC library code and in our analysis we were unable to discern any pattern that would allow the automatic return to the normal flow manner. These exceptions are exceptions of SystemC library. The exceptions of the syntax "try ... catch" do not appear in these situations as it appears that its flow keeps in the program source code. Since the aim is to consider only the source code content, we simplify the analysis of the invoke instruction. Our parse considered that this statement consists in a call instruction and the “normal label”. So we ignored the “exception label” as this accounts for instructions of the SystemC library and we just want to parse the source code instructions.

Thus, the parse unfolds the instruction invoke in a call statement and a statement that marks the end of basic block (normal label). This last statement is then discarded by the **Back-end** component as it is only interested in the source code instructions.

5.1.2.4. CompleteInstruction function

In this section we describe the function which was built to look for the rest of the instruction and the information referring to the source code. As described in the previous section this function is called at different times and has the ability to complete the instruction requested and therefore to reconstruct the source code instruction.

As mentioned earlier, we could not cover in detail all LLVM instructions. Instead we only cover those that were appearing in the examples tested in C, C++ and SystemC. We focused our efforts so that all instructions in the echo cancellation system (see Section 3.6) were covered.

The instructions in LLVM are similar to the instructions in assembly. The LLVM instructions contain a lot of information stored thanks to the debug information that is associated. Simply put, the LLVM instructions consist of an opcode, a code that identifies the type of instruction, and two values. These values can represent different types of information from information encoded as debug information, constants or labels [49]. To make it possible the usage of the function in various situations and easily extendable if new instructions appear, we implemented a recursive function that takes a value and will first examine if the value refers to an instruction, constant or unknown type instruction.

5.1.2.4.1. Instruction

If the value refers to an instruction, we convert the value to an instruction and access its opcode. From its opcode we know what kind of instruction it refers. With a simple if/else chained statement we

redirect the instruction to the corresponding option. Each instruction receives a different data processing.

In the case of the call instruction, the process to be performed is the same as mentioned above. The call statement indicates the type of data that the function returns, the function name and the call function parameters. For each parameter, the *CompleteInstruction* function is called passing the value of the parameter.

Another example is the binary operation where it is necessary to know the values associated with each parcel and which operations. Each parcel can be a function or a constant and thus becomes necessary to use the *CompleteInstruction* function to find each parcel.

Since the *CompleteInstruction* function does not cover all the existing instructions in LLVM, the function body is prepared so that when any new instruction appears, its treatment is easily configurable. Currently when the parse finds a reference to an untreated instruction, the instruction is recorded by the parse as sequential instruction and the parse will indicate to the user that it doesn't know the instruction and prints the instruction in LLVM format.

5.1.2.4.2. Constant

In LLVM there are various types and subtypes of constants. In the examples we tested we just had to analyze constants of type Integer, Float and Double. The constants Bool and Char are also analyzed as they are always converted to Integer in LLVM IR. If it finds a kind of constant unanalyzed, the parse will behave the same way as in the cases of the instructions by indicating to the user that it doesn't know the constant and prints the constant in LLVM format.

5.1.2.4.3. Other instructions

The value in LLVM may refer to various types and subtypes of objects. As mentioned before if it finds a new object, the parser process as the object as unknown and it prints it in LLVM format.

Even if there is the risk that an instruction does not fall in these identifiably categories, our parse can extract the source code instructions even though some are consider as unknown. CFGs do not suffer this limitation, since the necessary instructions for the construction of CFG are recorded, that is, the conditional instructions. The unknown are classified as sequential instructions. So far we had only one example of behavior changing when we construct the CFGs caused by this limitation. At this stage we are only cataloging the instructions and then generate the CFG, unknown instructions in this context only serve to count the instructions found.

5.1.2.5. Algorithm cataloging instructions

The purpose of this stage was to select and catalog the functions and instructions belonging to the source code and allow us to build in a later stage the corresponding CFG of the program. For the construction of the approach presented here it was necessary to analyze examples of LLVM IR resulting from C/C++ and SystemC. Several approaches have been implemented, tested and evolved

until we get to this approach that showed us the best results. This approach thus resulted from the combination of several approaches.

5.1.2.5.1. Complete Algorithm

So far we have introduced several mechanisms to consider for the correct execution of the parse. We will now describe how all these mechanisms interact with each other to extract the necessary information. Figure 11 shows a simplified pseudo-code of our algorithm.

```
1:   List functionProgram;
2:   for each Function *f in Module *M
3:       save_in_my_Structure M;
4:       if f belongs to functionProgram
5:           save_in_my_Structure f
6:           for each BasicBlock *bb in f
7:               save_in_my_Structure bb
8:               for each Instruction *ii in bb
9:                   if ii is a declaration or a call to a class
10:                      save_in_my_Structure ii
11:                      parse bb to retrieve all information about this ii
12:                      mark the instructions that had his information
13:                      end if
14:                  end for
15:              for each Instruction *ii in bb
16:                  if ii is a store or terminator instruction
17:                      save_in_my_Structure ii
18:                      parse bb to retrieve all information about this ii
19:                      mark the instructions that had his information
20:                  end if
21:              end for
22:              for each Instruction *ii in bb
23:                  if ii is a call instruction or refers to a constant
24:                      save_in_my_Structure ii
25:                      parse bb to retrieve all information about this ii
26:                      mark the instructions that had his information
27:                  end if
28:              end for
29:          end for
30:      end if
```

```
31:  end for
```

Figure 11 - Oversimplified algorithm for Middle-end component.

The parser created is applied to all functions that have been selected using the control structure mentioned in Section 5.1.2.1.1. For the correct analysis of all basic blocks in a function we use control structures described in Section 5.1.2.1.2 to ensure that all basic blocks and their connections are analyzed and recorded. We started the parse of the function by the *entryblock* representing the first basic block of the function.

The parse searches only the relevant instructions (see Section 5.1.2.3) for the selection of the source code instructions from all LLVM instructions. To complete the information stored in each selected instruction, the *CompleteInstruction* function is used (see Section 5.1.2.4).

In this approach it was not enough to find out which instructions to select. There was also the need to extract them in a certain order to avoid interference between their information stored in each instruction.

To correctly extract the instructions it is necessary to perform three parses for each basic block so we can ensure that the contents of the instructions are successfully removed. During the parses to a basic block we resort to the body inspection mentioned in Parse of Basic block section so that each time an instruction is analyzed, this would have its position stored in the control structure. This allows that among parses, it is not possible to use it again, thus avoiding misinterpretations by the algorithm. Also whenever all existing positions in a basic block are cataloged the control structure server has a stopping point.

In a first parse, the goal is to extract declaration of variables and class calls. In the second we extract the stores and terminator instructions. In the third and final parse, we cataloged separately calls to internal and external functions and all unknown instructions (Figure 11).

Each parse only considers the instructions that its position has not been set in the control structure (see Section 5.1.2.1.3).

```
13:  int a;                  call void @llvm.dbg.declare(metadata !{i32* %a}, metadata
                           !913), !dbg !915
14:  int b = 10;             call void @llvm.dbg.declare(metadata !{i32* %b}, metadata
                           !916), !dbg !917
15:                          store i32 10, i32* %b, align 4, !dbg !917
16:  int c = 12;             call void @llvm.dbg.declare(metadata !{i32* %c}, metadata
                           !918), !dbg !919
17:                          store i32 12, i32* %c, align 4, !dbg !919
18:                          %0 = load i32* %b, align 4, !dbg !920
19:                          %1 = load i32* %c, align 4, !dbg !920
20:                          %add = add nsw i32 %0, %1, !dbg !920
```

```
21:  a = b + c;        store i32 %add, i32* %a, align 4, !dbg !920
22:  return 0;         ret i32 0, !dbg !921
```

Figure 12 - Section of adding two values, LLVM IR.

Taking as an example the section described in Figure 12, on the first pass, the parser begins to look for variable declarations and class calls. In this example the first instance is found in position 13. When calling the *CompleteInstruction* function we will not get any instructions since it is only the variable declaration. After it is found position 14 due to declaration of a variable and the *CompleteInstruction* function finds position 15 where the initialization of the variable b is stored. The same applies to position 16 and 17. In the second parse, we look for store instructions and then by the terminator instructions. Positions 13 to 17 have already been marked, so the stores have already been analyzed and therefore they will not be reviewed in this second parse.

Thus, position 21 is the first to be found. With the help of *CompleteInstruction* function positions 20, 18 and 19 are also found. The last free position 22 refers to a terminator statement where the *CompleteInstruction* function does not find any instructions for the return value, because it is an integer constant.

5.1.3. Back-end

At this stage of **SysCFG** all information regarding instructions has been extracted and stored in a structure similar to LLVM IR detailed in the previous section. The **Back-end** component aims to change the structure in which the instructions are organized and store the information of this new structure in text files.

We recall that in the previous stage there weren't any changes in organization of the instructions, except in the selection of them. At the end we intend to organize each function in a CFG with specific characteristics. To this end, we will rearrange the instructions stored in the format Module, Function, BasicBlock and Instruction similar to LLVM IR into Graph, Edge and Vertex format.

The structure that comes from the **Middle-end** component contains the instructions stored in the same way that an LLVM IR. So here the parse that will take place will find the same obstacles that the one in the iteration of the structure from the previous phase. In order to achieve the coverage of all basic blocks and register all links between them, it is necessary to use two control structures that were also used in the LLVM IR parse. One will keep track of the analyzed basic blocks and the second will register the ordered pairs of basic blocks, in order to save the links between them (see Section 5.1.2.1.2).

This new structure will have three main components: (i) graph, (ii) vertex, (iii) edge. The graph is the structure corresponding to the functions, *i.e.*, a graph will store all information related to a function of the source code. In the graph is stored a list of vertexes and edges. A vertex is formed when a conditional statement is found. In Table 1, we refer which instructions of our structure are considered as conditional.

Upon iterating through the structure, when a statement with the ID listed in Table 1 arises, one vertex is created and it is stored the ID instruction that led to it. The conditional instruction is stored in the edge of where the data stream comes and that ended in the newly formed vertex. It is also stored in this edge the block of sequential instructions preceding the conditional instruction. The edges will represent the directed connections between the vertexes, helping to define the data stream.

From this parse each function will generate one CFG, which will split the function structure in more vertexes, because at this stage we consider all functions calls as conditional statements.

Figure 13 shows an oversimplified pseudo-code version of our algorithm for the **Back-end** component.

```
1:  MyStructure _myStr = getMyStructureFrom_Middle-end;
2:  MyCFG _cfg;
3:  Create a list with all MyCFG = _listCFG;
4:  for each Function *f in _myStr
5:      create list to save instructions to edges = _listInst;
6:      save every entry vertex of f in _cfg;
7:      for each BasicBlock *bb in f
8:          for each Instruction *ii in bb
9:              if ii.ID == relevantInstruction
10:                 save ii in _listInst and save _listInst in a new edge;
11:                 save a new vertex in _cfg with the type of ii;
12:                 save the new edge in _cfg and
13:                 connect the previous vertex with the new vertex;
14:             else
15:                 save ii in _listInst;
16:             end if
17:         end for
18:     end for
19:     save exit vertex of f in _cfg;
20:     save _cfg to _listCFG and empty _cfg;
21: end for
22: _listCFG = sort vertexes, starting each function with the entry vertex;
23: save _listCFG in a .txt file creating out final CFG;
```

Figure 13 - Oversimplified algorithm for Back-end component.

The **Back-end** component of **SysCFG** was designed to allow the **SysCFG** to works as a back-end of LLVM. At this stage the **Back-end** stores the information in a file format specifically designed to be compatible with the **IVG** component. The **Back-end** can be easily modified to generate another file format with the information. There are several types of files standards used by various visualization

and manipulation of graphs or verification tools such as GML [50] or XML [51]. This comes from the fact that the extracted structure of the **Middle-end** component keeps structure similar to LLVM IR, a familiar structure for many.

Thus at this stage, three files were generated to carry information to various components. The file "*fileCFGs.txt*" contains all the information about the structure of each CFG. The information is organized as follows:

- **General info**: contains the number of functions, vertex and edges found.
- **List of flows**: contains the list of the functions found. For each function, we have stores the ID, LLVM function name, the first vertex ID and the last vertex ID.
- **List of vertexes**: contains the list of all generated vertexes. Each vertex will have an associated ID, the ID of the conditional statement type that led to it (see Table 1), the number of edges that comes from this vertex and the number of edges arriving at the vertex.
- **List of edges**: contains the list of all edges generated. Each edge contains its ID, the ID of the initial vertex and the ID of the final vertex, the number of stored instructions and conditional value. The conditional value can be 0 or 1, in case of the edge comes from a condition and the value -1 otherwise.

The "*fileSMT.txt*" file contains the list of all instructions found associated with the edge ID where they are stored. The file "*listMod.txt*" contains a copy of List of flows from file "*fileCFGs.txt*".

5.2. Extract Structure

In this section we will detail the implementation of the **Extract Structure** component. As mentioned above (see Section 4.3), this module aims to extract from the source code a list of: signals, modules and the relationship between them. This information will be used as an input of the **IVG** component to be used for the construction of CFG of the program in analysis. At the same time, it will extract the list of #define directives, which are useful in the creation of SMTs assertions later on.

To better describe the operation of this component, first we will describe and explain what information is extracted from each file of the source code, using to that end a sample program of SystemC (see Section 5.2.1). Finally, we will describe how we store the previously collected information in a text file (see Section 5.2.2).

5.2.1. Methodology

To demonstrate the **Extract Structure** operation, we will use a simple program in SystemC, containing only one module. This program aims to make the sum of three values. The implementation of a SystemC program does not necessarily have to follow the methodology that will be described in this section. We evaluate this methodology because it was used in the echo cancellation system. Annex A shows the sc_main function of the program.

The starting point for program execution is the sc_main function where the signals, that connect the system modules, are defined. Initialization of the modules instances, using their constructors and set which signals are connected to each instance ports. Additionally, SystemC functions to configure and start the simulation are also called.

From the sc_main function, the parse will extract the list of declared signals. The declaration of each signal contains the name and the type of data that signal carries. In the case of the reset signal (Annex A), it can only carry boolean data. The clock signal allows, during simulation time, the synchronization of events between SystemC modules. In this case, the parse will only extract the signal name: clk (Annex A).

In SystemC, a module can be instantiated multiple times and each instance can have different signals associated. The parse considers that each instance of a module corresponds to a different object. In the Annex A, the somador16b module is instantiated twice (som1 and som2) and each associated with different signals. We associate each instance found with the module that originated it and then we extract the name of the instance ports and associate each port to a signal.

The Annex B shows the module structure somador16b. The modules in SystemC are described as follows:

- Declaration of module ports, which can be input, output or input/output;
- Declaration of internal variables of the module;
- Module constructor declaration;
- Declaration of module functions implemented through a basic C or C++ function;

To complete the information of each port, we perform a parse of every file where the modules are defined, Annex B. In this parse we complement the information corresponding to the module ports. This is where we can know the type of each port, which allows us to define the flow of the data.
At the end we will have sufficient information to infer the static structure of the program. In the case described in Annex A and Annex B, the corresponding structure is described in Figure 14. This structure serves as map for the organization of the module functions CFGs to construction the program CFG.

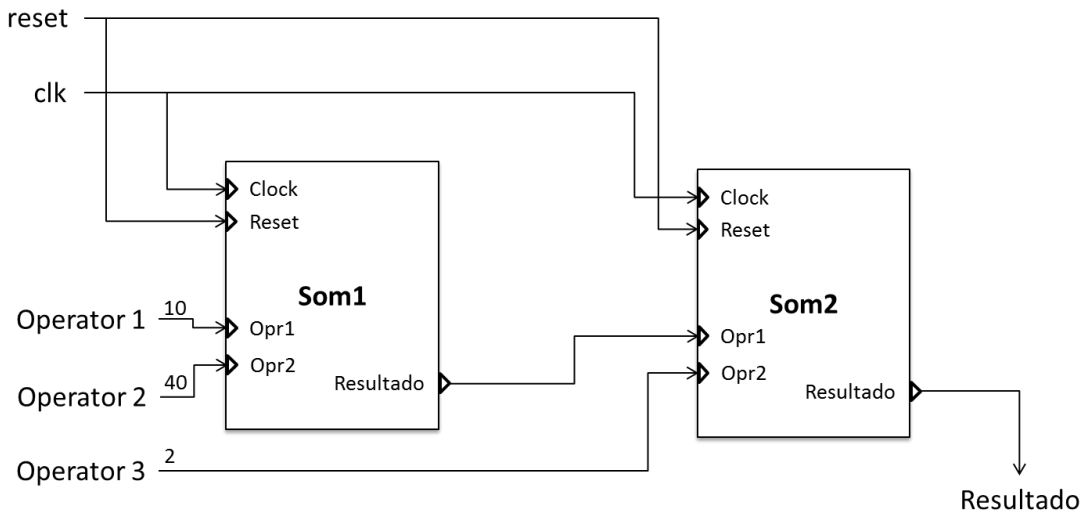

Figure 14 - Static structure of the program.

5.2.2. Output

In this section we will explain how the previously collected information is stored and related to the information extracted in the **SysCFG**. To associate the CFGs of the source code functions extracted in **SysCFG** component with the extracted information of **Extract Structure**, we import the file "*listMod.txt*" which contains the CFGs list of the functions belonging to the source code and its respective ID. The file generated by the **Extract Structure** will be named "*structure.txt*" and contains the following information:

- **Module list**: contains the name of each source code module and its ID;
- **Instance list**: contains all instances declared in sc_main function. For each instance, it has its ID, the ID of the module that instantiated it and its name;
- **CFG list**: contains a copy of the content of the file "*listMod.txt*";
- **CFG per Module**: contains the list of IDs of CFGs associated by Module, except the CFGs of module functions;
- **CFG functions per Module**: contains the list of IDs corresponding to the CFGs module functions, associated by Module;
- **Edge list**: contains the links between instances of the modules in the program. Each edge has an ID, the ID of the input instance and the ID of the output instance. Using information collected about signals and the respective connections to the ports of each instance, we can discover the edges of the entire system.
- **Signals list**: contains the signals list, where each signal contains an ID, its name and the type of object that it carries (Boolean, Integer, …);
- **Ports list**: contains the list of ports, where each port has an ID, instance name that it belongs to, port name, port type and the ID of the signal that it connects;
- **Define list**: contains the list of #define directives that may exist;

To discover the edges of the system, we begin to discover for each port of each module instance verify what signal is connected. Then, checking the port type (input, output or input/output), we analyze if the signal in question was using a port of a different instance. In Figure 14, by treating the port "*resultado*" of "*som1*" instance, we can see that it is an output port and that it was connected to the signal "*resultado*". So we go through the remaining input ports to see if the "*resultado*" signal was used on any other port. In this case it was found that the "*resultado*" signal was connected to "*opr1*" port of "*som2*" instance. In this association there may be other cases, such as a signal of an output port that is connected to more than one input port, or a signal that does not connect anywhere. If we do not connect it anywhere, such as the signal "*resultado2*", we give this edge a value of -6 as the ID the output instance. If the signal does not start from an instance, the input instance value ID is -7.
In the case of clock and reset signals, these two signals have no input instance ID value, since these two signals do not come from any module. As the input instance ID, we use the value -1 and -2 respectively for clock and reset signals. Annex C contains the generated file to the program described in Annex A and Annex B.

5.3. SysSMT

In this section we describe the proposed approach to extract the input parameters of a given path. The procedure described in here is performed on a path and repeated for all existing paths in "*filePaths.txt*" file, which comes from the **IVG** component. To extract the input parameters of a path, this component is divided into three sub-components:

- **LLVM to SMT**;
- **Generate Path**;
- **Z3**;

In **LLVM to SMT** subcomponent the goal is to convert all LLVM instructions regarding the source code instructions to SMT assertions and store this information in the "*fileSMT.txt*" file. In **Generate Path** subcomponent, it aims to generate files with extension "*smt2*" for each path. Each file will contain the SMT assertions for each path that was found by the **IVG** component. These files will be built based on the SMT assertions generated in subcomponent **LLVM to SMT** that were stored in the "*fileSMT.txt*" file. The **Z3** subcomponent goal is to generate the input parameters of each path using the Z3 solver. As input for each path, it uses the files of the paths generated in the **Generate Path** subcomponent.

Part of the subcomponent **LLVM to SMT** is implemented in the **SysCFG** to minimize the amount of required parses to the LLVM IR. In **SysCFG** there were no mentions to **SysSMT** for topic separation. The remaining sub-components are implemented in **SysSMT**.

5.3.1. LLVM to SMT

At this stage the objective is to convert LLVM instructions from the source code to SMT assertions so that the solver Z3 is able to generate the input parameters for each previously found path.

To successfully implement this solution, it will be necessary to initially select and store the contents of LLVM instructions regarding the source code to an intermediate structure. The chosen structure was the Abstract Syntax Tree (AST), a structure often used for these situations. In a second phase it aims to generate SMT assertions from the information stored in the intermediate structure and generate the "*fileSMT.txt*" containing the SMT assertions of the source code.

5.3.1.1. LLVM to AST

In this first phase, it is necessary to review some concepts referred to in Section 5.1.2.3. As mentioned earlier in LLVM IR some source code instructions are split into multiple LLVM instructions. When this happens **SysCFG** swipes the remaining LLVM instructions to know if there are any other LLVM instructions corresponding to the same source code instruction. To this end, we use the *CompleteInstruction* function. This allowed us to complete the information about an instruction from the source code iterating through other LLVM instructions. At this stage the goal is just to iterate through the instructions, marking the instructions found in a control structure (see Section 5.1.2.1.3) to prevent errors associated with the analysis of instructions between the various parses made to the LLVM IR. The information of the instructions content is not saved.

To avoid repetitive parses to the LLVM IR and improve the tool performance, we decided that the extraction of the information from the LLVM instructions to an intermediate structure should be done by the **SysCFG** (see Section 6.1.2)

The structure chosen as intermediate structure was the Abstract Syntax Tree (AST). This structure allows storing abstract information of each type of instruction. We use a solution similar to the one in the tutorial [52] which exemplifies the best way to store information from LLVM instructions in an AST. In this tutorial there is a description of how the data structures must be implemented in order to extract correctly the information of each LLVM instruction type thanks to the *opcode* that allows distinguishing what types of instructions we are dealing with.

To integrate with our existing solution, we added the AST data structure and populate the structure during the instructions iteration.

When the parser finds any unknown instructions, they are stored in a string using the LLVM format, so that the user can change, at the end of the subcomponent **LLVM to SMT**, the "*fileSMT.txt*" that contains the SMT assertions and insert the assertions manually.

At the end when all instructions are stored in the ASTs, we will generate SMT assertions.

5.3.1.2. AST to SMT

At the beginning of this phase all source code instructions are stored in the AST data structure.

This phase aims to generate the "*fileSMT.txt*" containing the SMT assertions of the source code instructions. In this file each SMT assertion stays associated with the edge ID where the corresponding LLVM instruction is stored in the CFG of the function (see Section 5.1.3).

To make sure that the Z3 solver receives the right information about each path, it is necessary that only the variables from the source code reach the solver. This way we do not make the problem more complex for the solver to solve. Similarly, some instructions that do not affect the input and output parameters are discarded. This is the case of data output instructions to the screen, as we will talk later on.

We will not cover here all instructions and instructions particularities that may appear in source code, instead we will focus our attention on a set of instructions:

- Variable declarations;
- Store instructions (store a value in a variable);
- Function calls;
- Cycles, conditions and function return;
- Unknown instructions;

In the same way that one source code instruction can unfold to various LLVM IR instructions, the same can happen with SMT assertions, *i.e.*, one LLVM IR instruction can unfold in various SMT assertions. Following we will describe briefly how each of the instructions listed above is treated.

5.3.1.2.1. Variable declarations and store instructions

As mentioned before (see Section 5.1.2.3) variable declarations may appear in various ways, with or without variable initialization. In SMT it is necessary to separate the variable declaration from its startup value by creating several assertions. Before using a variable in the solver, we must declare it. We require only the name and the variable type to construct the SMT assertion.

The variable initialization receives the same treatment as the instructions that store value in the variable (LLVM store [53]). Both instructions are intended to bind a value to a variable. In a variable we can assign a value. Depending on the type of association one or a set of assertions are created.

5.3.1.2.2. Function call

In the case of function calls, the treatment will be different depending on the function called. We consider the following two cases: (i) C/C++ and SystemC library function calls; (ii) source code function calls.

In the event that the called function refers to a function of C/C++ or SystemC library, there may be cases where this function call has no direct correspondence to SMT assertions. Then it is necessary to create SMT assertions that permit the linearization of the function, as it is the case of the logarithmic function. In other cases, some functions do not affect the discovery of the input parameters. We have for example the functions that are used to write and read from the screen (example: function *std::cin* e *std::cout* in C++). These instructions will be ignored and no assertion is created. In the case of the data read function, there may be cases where a variable is dependent on the value that is inserted by the user. At this stage the variables are declared, but it will be the Z3 solver to give the values to those variables.

For source code function calls there is a different treatment. As mentioned previously (see Section 4.4) for the discovery of path to be analyzed, it is necessary to construct the program CFG. The construction of the CFG was performed starting with the sc_main source code function and in some cases replacing the function calls with the corresponding CFG of the called function, while also adding other CFGs in a particular order. Due to this methodology we will have a problem with the variable's name. Take for example the following situation: the function sc_main has the variable "*result*" declared and it is initialized with a certain type of value. Another function of the source code contains a variable named "*result*". To build the file with the assertions of a certain path, we would have two variables declared with the same name, which will generate error.

To solve this problem we decided to change all variable's name of each instance of the source code functions. When we find a call instruction to a source code function, it will create a variable for each parameter containing in the name a concatenation of two strings: the parameter name and a randomly generated string with a length of 5 characters. The same string is applied to all the variables contained in this function's instance. So the function call is replaced by a set of assertions containing the variable declaration for each parameter and assertions regarding the variables initialization with the value assigned to each parameter. This way, for each function instance the name of the variables will be different thus avoiding errors.

During the construction of the files for each path besides putting the assertions that were created from the call instruction of the function, we need to replace the name of the variables that make up the function body called by the name created earlier.

5.3.1.2.3. Cycles, conditions and function return

We move now to the case of the LLVM terminator instructions, like the conditional branch, invoke and return instructions, given that these correspond to cycles, conditions and function return.

As mentioned earlier the conditional branch instruction corresponds to the conditional statements or cycles found in the source code. These instructions contain a condition that determines the next basic block to execute. Take for example the condition "*if*" and the cycle "*for*". In the case of "*if*" condition it is necessary to store the condition contained in "*if*" in two different forms in assertions. The first form corresponds to the assertions of the true condition, while the other to the assertions of the negated condition. In "*fileSMT.txt*" there is an ID to indicate that both assertions groups refer to the same "*if*" statement. So depending on the path to analyze, the group of assertions will be different, depending on condition that the path needs.

In the case of "*for*" loop we should remember that in most cases the instruction is composed of variable initialization, and in some cases also a declaration of a variable. Then we have a stop condition and increment instruction. In some cases the "*for*" instruction is equivalent to a "*while (true)"* when the "*for*" has no parameters "*for(; ;)*". The "*while*" instruction receives similar treatment as the "*for*" instruction and just the parameter corresponding to the stop conditional is consider.

In the case of cycle instructions, we will not convert the instruction to assertions at this stage. Instead, we will only store the information in a structured way at the "*fileSMT.txt*" file.

In the case of the "*for*" instruction, we will store the declaration and initialization of the variable, the condition and stop the increment instruction.

The body's instructions are converted to SMT assertions and stay associated with an ID so that the *Generate Path* subcomponent can identify them as the loop body instructions. Loops are only unrolled on the **Generate Path** subcomponent according to the information that will receive by the **IVG** component of the conditions of each path.

This approach is due to the fact that the Z3 solver will only receive inputs of linear paths that were discovered by the **IVG** component.

In the case of the "*invoke*" instruction, earlier we referred that it is used to SystemC library calls, among other cases. As an example we have the declaration of input/output module signals. In these cases we will consider that they refer to instructions of primitive types of variables such as *int*, *bool*, among others. So we will create a SMT assertion for each signal the same way as a declaration of a variable.

Finally in the case of the "*return*" function, we only need to treat the return value. This value can correspond to various objects.

At the end, the file "*fileSMT.txt*" is created and it contains all instructions in the SMT format associated with the edge ID where they were stored. As previously seen, the conversion of some instructions for SMTs will only be made at the time of construction of the files containing the SMT assertions for each path, described in the next section. The conversion of the remaining instructions follows the same methodology as that referred in this section.

5.3.2. Generate Path

In the **Generate Path** the goal is to create a file with the extension "smt2", which lists the instructions of the path to test the SMT format. First we must use the files "*listPaths.txt*" and "*fileSMT.txt*".

We start by performing a parse to the file "*listPaths.txt*" containing for each path the list of IDs of the edges that form the path. The information of each path is stored in a list of Integers. Based on the list that we created previously, another parse is performed to the file "*fileSMT.txt*" containing a list of SMT assertions. For each of these assertions, we have the edge ID, where they are stored. As a result of this second parse, we will create for each path a list of assertions contained in the path, using as a guide the list of ID edges from the first parse.

This procedure will be performed for all the existing paths in the file "*listPaths.txt*". The file name generated for each of the path is called test<idList>.smt2 where <idList> represents the position of the path in the list. This designation serves only to distinguish the files of each path.

At this phase we can find an instruction that is not converted to SMT, because their values depend on the path analysis. These instructions that correspond to conditional and cycle instructions will be converted at this phase to SMTs taking into account the path conditions.

At the end a parse is performed for each generated file to determine if any constant defined in #define directives are found in SMT assertions. This parse is done with the help of "*structure.txt*" file generated by the **Extract Structure**. If any constant is found, it is replaced by its value. This step is necessary so that Z3 does not consider the constants defined in #define directives as variables. If the Z3 consider a constant as a variable, the problem to be solved by the Z3 becomes more complex due to the increasing number of variables to consider. If it would happen, the result of the Z3 could be compromised since the value of the variables may not be correct or Z3 could not get the values of variables for a given path.

5.3.3. Z3

Z3 [3] is a Satisfiability Module Theories (SMT) solver. Z3 was developed by Microsoft and is used in many analysis, verification and test-case generation.

In our project we intend to use the Z3 to discover the input parameters of a given path. For each path we are expecting two possible outputs, SAT or UNSAT. If the result is SAT, *i.e.*, a viable path, the Z3 will return the value of the variables that are in this path. If the result is UNSAT, it means that the path tested is not viable and Z3 cannot obtain values for the path variables. These variables are considered as input parameters for each path. Whatever for each path it is generated an output by Z3, it is stored in a file.

For the input parameters for each of our path, we built a small program that receives the file generated in **Generate Path** and injects the file content to the Z3. In the end we received the result of Z3 on the tested path.

6. Validation

In this section we present some results obtained with the **SysCFG**, **Extract Structure** and **SysSMT** components to validate their methodologies. We tested our tool in an Intel® Core™ i7 running at 3.4GHz with 8GB of memory. We are using LLVM-3.2 and its compatible version of PinaVM.

6.1. SysCFG

To validate the **SysCFG** component described in Section 5.1, we used the following test cases described in SystemC:

- Examples from the PinaVM distribution;
- Program “sum”, which is a sum of three values;
- A real system, the echo cancellation;

PinaVM has some limitations and not all programs run successfully in the PinaVM’s front-end. As we need the file extracted from PinaVM’s front-end for our analysis we decided to choose some of the examples from the PinaVM distribution that passed in its front-end. We also used the “sum” program that follows the same implementation methodology that the echo cancellation and its source code can be found in Annex A and Annex B. We decided to include the results so that the reader has access to the source code and results of a test case. Finally we will present the results of a real system, the echo cancellation, described in section 3.6. We will prove that this tool has promissory results not only with small SystemC examples, but also with real systems. As we cannot provide all the source codes of the test case programs, due to their dimensions, we summed up some of its characteristics in figures. So each test case in Table 2 shows the number of conditions and cycles, while Table 3 presents the number of source code lines of code, excluding comments and white lines of each test case.

Programs		**Flow control statements**		
		while	*for*	*if/else*
elab_easy		1	-	1
elab_easy_unit		1	-	1
elab_easy_int		1	-	1
elab_ports_array		1	1	-
elab_multiple_instances		-	2	-
empty-sc		-	-	-
events		-	1	1
jerome-chain		-	2	4
sc_clock		1	-	1
sum		1	-	1
echo cancelation		9	2	28

Table 2 – Flow control statements.

6.1.1. Front-end

As described in Section 5.1.1, the **Front-end** component is designed to convert SystemC into LLVM IR, using PinaVM.

The PinaVM generates many files as output, but the file that we will use for our tool contains *opt.bc* or *linked.opt.bc* extension. When the test case successfully passes through the PinaVM front-end it generates a file with the extension *opt.bc*. If it continues to its PinaVM back-end the file with *opt.bc* extension is replaced by the file with the extension *linked.opt.bc*. However SysCFG can handle both file types, because they contain the LLVM IR of the analyzed program. Given the name of the file "*linked.opt.bc*", we assume that the file is changed so that the references to the library functions are linked to the object files or libraries of the target machine. These changes made by the linker do not affect the structure of the function definitions that exist in the LLVM IR. Also since our parse does not handle functions related to C/C++/SystemC libraries, the changes produced by PinaVM back-ends do not affect the parse. This way the **SysCFG** can handle the LLVM IR produced by the PinaVM front-end and LLVM IR produced by the back-ends.

6.1.2. Middle-end

In this component the goal goes through select only the functions and instructions that correspond to the source code (see Section 5.1.2). In Table 3 we can see that when we convert the source code to LLVM IR, the number of the functions and instructions increases substantially. This is due to the fact that LLVM IR source code functions are mixed with the C++ and SystemC library functions. In the end, the **Middle-end** component will only select functions and instructions of the source code. As we see in Table 3, the decrease is substantial in relation to the LLVM IR original. Regarding the number of lines in the source code the number of functions and instructions is sometimes higher because the LLVM IR unfolds the calls to the module components which still produce a high amount of objects at the end of the **Middle-end** component in regard to the original source code.

Programs	**Input**			**Output**	
	Source code lines	*LLVM IR Functions*	*LLVM IR Instructions*	*Selected LLVM IR Functions*	*Selected LLVM IR Instructions*
elab_easy	170	495	4449	10	267
elab_easy_unit	189	560	5091	10	320
elab_easy_int	203	525	4845	11	310
elab_ports_array	143	473	4264	17	520
elab_multiple_instances	175	472	4497	17	752
empty-sc	7	213	1599	1	4
Events	40	253	1936	10	258
jerome-chain	88	268	2159	18	487
sc_clock	57	480	4178	17	425
Sum	68	685	6666	9	510
echo cancelation	1240	1149	15776	73	5587

Table 3 - Middle-end component results.

6.1.3. Back-end

This component aims to generate the graph of each source code function (see Section 5.1.3). In Table 4 we have the inputs and outputs of the **Back-end** component. The **Back-end** component receives as input the functions and instructions previously selected by **Middle-end** component. As outputs the component generates the graphs for each function, composed by vertexes and edges. The number of instructions stored by graphs is less than the number of source code lines in Table 3, so we conclude that the **SysCFG** can generally select only the source code instructions.

Programs	**Input**		**Output**			
	Selected LLVM IR Functions	*Selected LLVM IR Instructions*	*Flows*	*Vertexes*	*Edges*	*CFG Instructions*
elab_easy	10	267	10	67	68	64
elab_easy_unit	10	320	10	78	79	75
elab_easy_int	11	310	11	79	80	76
elab_ports_array	17	520	17	108	109	98
elab_multiple_instances	17	752	17	126	127	118
empty-sc	1	4	1	5	4	4
Events	10	258	10	67	68	61
jerome-chain	18	487	-	-	-	-
sc_clock	17	425	17	111	112	106
sum	9	510	9	129	130	126
echo cancelation	73	5587	73	1077	1114	1053

Table 4 - Back-end component results.

Despite the presented metrics it is necessary to analyze the **SysCFG** output files to see if there are any errors generated. At the end, three of the eleven test cases have failed: *elab_ports_array*, *elab_multiple_instances* e *jerome-chain*. When examining the file "*fileCFG.txt*", we verified that some functions could not generate the vertex with the ID 1 (end function) which was replaced by the ID 4005. This value warns about an error occurrence at the end vertex function. The error was due to the use of arrays, something that was not supported at this time by the **Back-end** component.

In the case of jerome-chain test, the **Back-end** component also failed to generate the function CFGs due to an unknown instruction. The unknown instruction is the switch instruction, a LLVM terminator instruction. This statement is not supported by **SysCFG** at the moment since had never appeared in the examples tested earlier. The **Middle-end** save the instruction as sequential, causing the **Middle-end** parse to ran successfully until the end. The number of instructions selected by the **Middle-end** is not correct because part of a function path was ignored, not guaranteeing the entire source code coverage. The **Back-end** parse could not finish its parse because an error occurred due to a dependency on an edge.

In test cases containing the while instruction, there is a problem in sorting the vertexes of the file "*fileCFG.txt*". In these cases the analysis was a success, however the order of the vertexes containing the while instruction does not comply with the expected ordering of the **IVG** component.

In Figure 15 we have a graph where the execution times of each test case are visible. For each test case the time of the **Front-end** implementation represents the PinaVM runtime. The time of **Middle-**

end and **Back-end** are measured together. We can also verify that the more complex the source code is, the longer our tool runs.

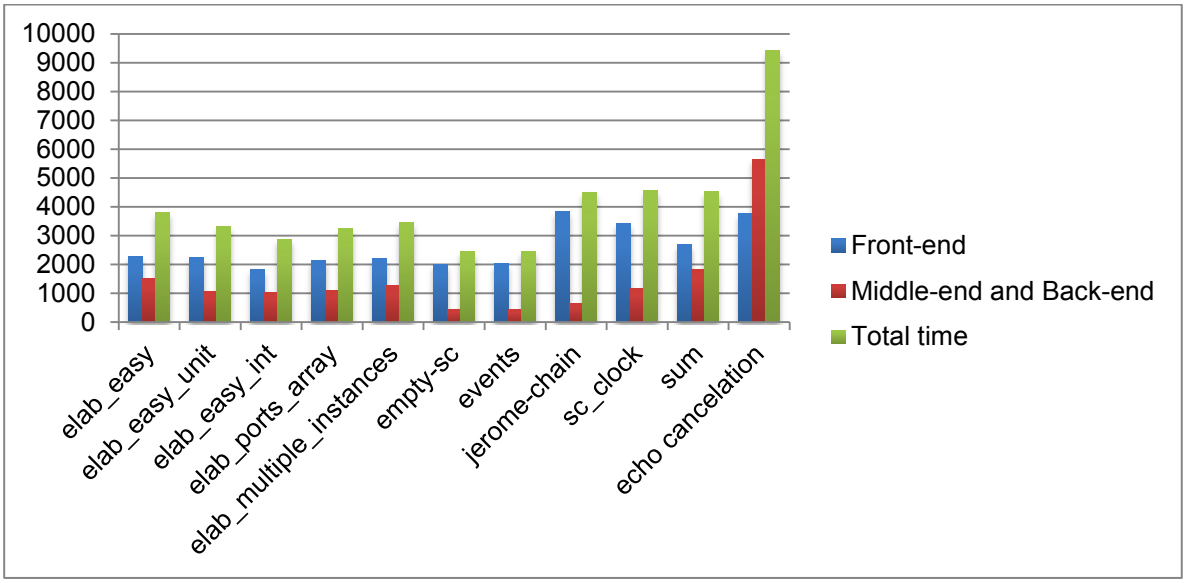

Figure 15 - Execution times (milliseconds).

6.1.4. Current Limitations

SysCFG has some limitations when dealing with certain characteristics of C, C++ and SystemC languages. Below, we enumerate some of the limitations that our method presents at this time:

- It does not support arrays;
- Compound predicates: do not use *if (i==3 && k <0) {//code}*, but rather use simple predicates like *if (i==3) {if (k <0) {//code}}*;
- Calling more than one function: do not call more than one function in the same instruction so that our tool can be able to collect all the function calls *(int a = fact(i) + fact(i+1).)* Only the call "*fact(i)*" is counted. So that every function is handled do: *int f1 = fact(i); int f2 = fact(i+1); int a = f1 + f2;*

6.2. Extract Structure

The **Extract Structure** component consists of a parse to the test program source code (see Section 5.2). At this stage we must manually change the code to indicate the following data:

- Location of the source code files;
- Source code module names;
- Module function names;

This component has been changed and tested with the sum program and the echo cancellation. The generated file of the sum program contains 49 lines and is in Annex C. In the case of the echo cancellation the generated file contains 1745 lines.

To verify that the information collected by the **Extract Structure** was the expected content, we manually checked the contents of the generated files. The purpose was to confirm that all modules, signals and their connections were properly identified and registered. With this analysis to the generated files, we verified that the **Extract Structure** meets its goals.

6.3. SysSMT

The **SysSMT** component approach described in Section 5.3 was never tested in conjunction with the other tool components. The reason was related to the fact that the processing of LLVM instructions to SMT assertions has not been implemented and the **IVG** component is still under development.

To validate the solution described in Section 5.3.3, we created small programs in C++ and SystemC. **SysSMT** is composed of three subcomponents: **LLVM to SMT, Generate Path** and **Z3**, as mentioned in Section 5.3.

The subcomponent **LLVM to SMT** and **Generate Path** are not automated and their methodology is used manually. To validate our solution, we start by execute the **SysCFG** to extract the file "*fileSMT.txt*". At this stage this file contains the source code instructions in LLVM IR format.

To perform the actions of the first subcomponent it is necessary to change the "*fileSMT.txt*" file manually to convert the instructions of LLVM for SMTs. Regarding the second component it was necessary to find the edges of a path. To do this we analyzed the source code and manually track which the edges formed the path. This is useful to the next stage to find out which the instructions in *"fileSMT.txt*" belong to the path for analysis. This step had to be manual since the **IVG** component is still being developed.

Finally we created files with the extension "*smt2*" containing only the SMT assertions of the path. Then we run our program that takes as input files with extension "*smt2*" and injects the content of the input files to the Z3 solver so that the input vectors are discovered. Despite only having tested the solution with small C++ and SystemC programs, this allowed us to test the approach presented and conclude that this approach is valid.

7. Conclusion

In this thesis we propose a solution for a co-validation of an embedded system in order to provide the engineer with an extra tool for validation in the early stages of the design process of an embedded system. It receives as input a description of the system described in SystemC and a coverage level that needs to be achieved. This solution extends methodologies already used for validating hardware and software descriptions.

We discussed some methodologies that seem most relevant, that had the better results and we adapt them to the embedded system.

The tool proposed here consists of five components, where the implementation of three of them is part of the scope of this thesis. Section 4 briefly describes the tool architecture, describing its components and the interaction between them. Section 5 describes in more detail the implementation of the three components belonging to this thesis, the **SysCFG**, **Extract Structure** and **SysSMT**.

SysCFG aims to extract the CFGs of the source code functions from the SystemC description. To accomplish this it was necessary to convert the SystemC source code to an easily manageable intermediate representation. Thus it was chosen LLVM framework which has a wide acceptance in recent years by the community. At this time there are several front-ends for LLVM built, while others are being developed and improved, as is the case of PinaVM. This is the LLVM front-end for SystemC used in this thesis. This thesis (see Section 6.1) shows that this component presents promissory results not only for small programs but also to real systems like the echo cancellation system.

The **Extract Structure** consists on a source code parser to extract the static structure of the embedded system modeled in SystemC. This component saves the signs, modules and their connections between the two of them. At the same time, it also extracts the #define directives. This component arose from the need to create a helper structure at **IVG** component (outside the scope of this thesis) to create the CFG of the embedded system under analysis. In SystemC, module functions that emulate the module behavior are never called directly from the source code, only in simulation time SystemC decides who and when to call. Thus it is necessary to know beforehand the static structure to help in the organization of CFGs of the source code functions in order to build the embedded system CFG.

Finally, we propose a solution to the **SysSMT** component that aims to convert the LLVM instructions to SMTs assertions and discover the input parameters for all the paths discover by the **IVG** component. To discover the input parameters we used the Z3 solver produced and maintained by Microsoft. Unfortunately **SysSMT** component was not developed due to delays that occurred during the development of the other components.

In short, the development of this solution was hard, because there is few developed work for SystemC. There are many academic papers that at the beginning showed promising results, but in the

end they did not fit with the objectives of this project. Thus we had to find a solution from scratch where most components had to be developed.

7.1. Future Work

As future work, we propose to implement the solution presented in Section 4.1 to integrate the different components in an automated tool and removing some manual configurations needed at this time.

In the case of **SysCFG**, this component currently has some limitations when dealing with some particularities of the language and some objects, like arrays. Currently this component was developed with the LLVM-3.2 version to be compatible with PinaVM. A migration to the latest version is advisable, since the PinaVM is a project that is also being migrated to the latest version of the LLVM and some of its current limitations are being addressed. This would also bring advantages to **SysCFG** since not all SystemC source codes successfully pass with PinaVM. This can cause problems in **SysCFG** component, since it depends on the PinaVM output.

As mentioned earlier, **SysCFG** is a component constructed with the aim to become a LLVM back-end. It would be desirable to add the functionality of generating source code functions CFG's with more than one standard output format, like XML. This would allow an easy integration with other tools.

The **IVG** component needs to study a solution to support the embedded systems that have feedback, *i.e.*, modules that are connected in loop, taking into account the way the system is implemented. At this stage **IVG** can only generate paths for embedded systems without feedback. Finally, we propose the implementation of the solution of the **SysSMT** component described in Section 5.3.

8. References

1. **Bhasker, J.** *A SystemC Primer.* 2nd. Allentown : Star Galaxy Publishing, 2004.

2. **Lattner, Chris Arthur.** *LLVM: An Infrastructure for multi-stage optimization.* Urbana, Illinois : University of Illinois at Urbana-Champaign, 2002.

3. **Moura, Leonardo de e Bjørner, Nikolaj.** *Z3: An Efficient SMT Solver.* Redmond, WA, 98074, USA : Microsoft Research, 2008.

4. **Free Software Foundation, Inc.** GCC, the GNU Compiler Collection. [Online] GCC, the GNU Compiler Collection, 12 de 05 de 2013. http://gcc.gnu.org/.

5. **University of Illinois at Urbana-Champaign.** The LLVM Compiler Infrastructure. [Online] http://llvm.org/.

6. **Edison Design Group.** The C++ Front End. [Online] http://www.edg.com/index.php?location=c_frontend.

7. **Microsoft.** Visual C++. [Online] Microsoft, 2013. [Citação: 12 de January de 2013.] http://msdn.microsoft.com/en-us/library/vstudio/60k1461a.aspx.

8. **Marquet, Kevin, Moy, Matthieu e Karkare, Bageshri.** *A theoretical and experimental review of systemc front-ends.* s.l. : Verimag, 2010.

9. **Moy, Matthieu, Maraninchi, Florence e MailletContoz, Laurent.** *PINAPA: An Extraction Tool for SystemC Descriptions of SystemsonaChip.* New York, NY, USA : Proceedings of the 5th ACM international conference on Embedded software, 2005.

10. **Marquet, Kevin e Moy, Matthieu.** *PinaVM: a SystemC Front-End Based on an Executable Intermediate Representation.* New York, NY, USA : Proceedings of the tenth ACM international conference on Embedded software, 2010.

11. **Lattner, Chris e Adve, Vikram.** *LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation.* Urbana-Champaign : University of Illinois, 2004.

12. **Lattner, Chris.** *LLVM and Clang: Next Generation Compiler Technology.* s.l. : BSDCan 2008, 2008.

13. **Erhardt, Christoph.** *Design and Implementation of a TriCore Backend for the LLVM Compiler Framework.* 2009.

14. **LLVM Team.** llvmgcc - LLVM C front-end. *llvmgcc - LLVM C front-end.* [Online] http://llvm.org/releases/1.6/docs/CommandGuide/html/llvmgcc.html.

15. clang: a C language family frontend for LLVM. [Online] The Clang Team. http://clang.llvm.org/.

16. **Thomas, Donald e Moorby, Philip.** *The Verilog Hardware Description Language.* s.l. : Springer, 2008.

17. **Brown, S. e Vranesic., Z.** *Fundamentals of Digital Logic with VHDL Design.* s.l. : McGraw-Hill Science Engineering, 2004.

18. **Gajski, D. D.** *SpecC: Specification Language and Methodology.* s.l. : Kluwer Academic, 2000.

19. **Black, David C. e Donovan, Jack.** *SystemC: From the ground up.* Boston : Kluwer Academic Publishers, 2004.

20. **Costa, José Carlos Campos.** *Coverage-Directed Observability-Based Validation Method for Embedded Software.* Lisboa : Universidade Técnica de Lisboa, Instituto Superior Técnico, 2010.

21. **Binder, Robert V.** The Control Flow Graph. *Testing Object-Oriented Systems, Models Patterns, and Tools.* 6th. Upper Saddle River : Addison-Wesley, 2005, pp. 362-370.

22. **Myers, G. J. , et al.** *The Art of Software Testing.* s.l. : Wiley, 2004.

23. **Beizer, Boris.** *Software Testing Techniques.* 2nd. New York : Van Nostrand Rheinhold, 1990.

24. **Costa, José C., Devadas, Srinivas e Monteiro, José C.** *Observability Analysis of Embedded Software for Coverage-Directed Validation.* Piscataway, NJ, USA : Proceedings of the 2000 IEEE/ACM international conference on Computer-aided design, 2000. 0-7803-6448-1.

25. **Voas, J. M.** *PIE: A Dynamic Failure-Based Technique.* August : IEEE Transactions on Software Engineering, 18(8):717–727, 1992.

26. **Goradia, T.** *Dynamic Impact Analysis: A Cost Effective Technique to Enforce Error Propagation.* s.l. : Proceedings of Int'l Symposium on Software Testing and Applications, 1993.

27. **Rowson, J. A. .** Hardware/Software Co-Simulation. *Proceedings of the 31st Automation Conference.* 1994, pp. 439–440.

28. **Lee, S. e Rabaey, J. M.** A Hardware-Software Co-simulation Environment. *Proceedings of the International Workshop on Hardware-Software Codesign.* October de 1993.

29. **Harris, I. G.** Hardware/Software Covalidation. *IEE Proceedings of Computers and Digital Techniques.* 2005, pp. 152(3):380–392.

30. C2C. [Online] ftp://theory.lcs.mit.edu/pub/c2c/.

31. *Effective Lower Bounding Techniques for Pseudo-Boolean Optimization.* **Manquinho, V. M. e Marques-Silva, J.** s.l. : Proceedings of the Design, Automation & Test in Europe Conference, 2005. pages 660–665.

32. *lp_solve reference guide menu.* [Online] [Citação: 2 de December de 2014.] http://lpsolve.sourceforge.net/.

33. The CBMC Homepage. *Bounded Model Checking for Software.* [Online] Carnegie Mellon. [Citação: 02 de December de 2014.] http://www.cprover.org/cbmc/.

34. SCOOT. *A Tool for the Static Analysis of SystemC.* [Online] [Citação: 12 de November de 2014.] http://www.cprover.org/scoot/.

35. **Merz, Florian, Falke, Stephan e Sinz, Carsten.** *LLBMC: Bounded Model Checking of C and C++ Programs Using a Compiler IR.* s.l. : Springer Berlin Heidelberg, 2012.

36. *SMT-Based Bounded Model Checking of C++ Programs.* **Ramalho, Mikhail, Freitas, Mauro e Sousa, Felipe.** Scottsdale, AZ : IEEE, 2013.

37. **Besnard, Loic, et al.** *Automatic translation of C/C++ parallel code into synchronous formalism using an SSA intermediate form.* [Online] June de 2009. [Citação: 3 de January de 2015.] https://hal.archives-ouvertes.fr/inria-00400272/document.

38. *Programming real-time applications with SIGNAL.* **Le Guernic, Paul, et al.** 79, s.l. : Proceedings of the IEEE, September de 1991, Proceedings of the IEEE, Vol. 9, pp. 1321-1336.

39. *Polychrony for System Design.* **Le Guernic, Paul, Talpin, Jean-Pierre e Le Lann, Jean-Christophe.** 12, s.l. : World Scientific Publishing Company, April de 2003, Journal for Circuits, Systems and Computers, Vol. 3, pp. 261-304.

40. **Rakamari, Zvonimir e Emmi, Michael.** *SMACK: Decoupling Source Language Details from Verifier Implementations.* [Online] 2014. [Citação: 2 de January de 2015.] http://soarlab.org/publications/cav2014-re.pdf.

41. *DAG - a Program That Draws Directed Graphs.* **Gansner, E. R., North, S. C. e Vo., K. P.** 11, s.l. : John Wiley & Sons, Ltd, 1988, SoftwarePractice & Experience, Vol. 18, p. 18(11):1062.

42. **West, D. B. .** *Introduction to Graph Theory.* New Delhi : Prentice-Hall of India, 2005.

43. **Costa, J. e Monteiro, J.** *Computation of the minimal set of paths for observability-based statement coverage.* s.l. : 15th International Conference Mixed Design of Integrated Circuits and Systems, 2008.

44. *Automatic test data generation using constraint solving techniques.* **Gotlieb, A., Botella, B. e Rueher, M.** 1998. International Symposium on Software Testing and Analysis.

45. *A Dynamic Approach of Test Data Generation.* **Korel, B.** November de 1990, Conf. on Software Maintenance, pp. 311–317.

46. **Sen, K., Marinov, D. e Agha, G.** Cute: a concolic unit testing engine for c. *Proceedings of the 10th European software engineering conference.* New York : ACM Press, 2005, pp. 263–272.

47. **Cadar, C., et al.** Exe: automatically generating inputs of death. *CCS '06: Proceeding of the 13th ACM conference on Computer and communications security.* New York, NY, USA : ACM Press, 2006, pp. 322-335.

48. *An observability-based code coverage metric for functional simulation.* **Devadas, Srinivas, Ghosh, Abhijit e Keutzer, Kurt.** Washington, DC : IEEE Computer Society, 1996.

49. **LLVM Project.** LLVM Language Reference Manual. [Online] LLVM Compiler Infrastructure, 2014. http://llvm.org/docs/LangRef.html.

50. GML Format. [Online] Gephi.org, 2012. https://gephi.org/users/supported-graph-formats/gml-format/.

51. **Refsnes Data.** XML Tutorial. [Online] 2014. http://www.w3schools.com/xml/default.ASP.

52. **LLVM Project.** 2. Kaleidoscope: Implementing a Parser and AST. *LLVM Compiler Infrastructure.* [Online] 2015. http://llvm.org/docs/tutorial/LangImpl2.html.

53. —. ‘store‘ Instruction. *LLVM Language Reference Manual.* [Online] 2015. http://llvm.org/docs/LangRef.html#store-instruction.

54. *"Program Analysis as Constraint Solving".* **S. Gulwani, S. Srivastava and Ramaratahnam Venkatesan.** 2008. Proceedings of the Conferece on Programming Language Design and Implementation.

55. **Majumdar, R. Jhala and R.** "Software Model Checking". *ACM Computing.* 2009. Vols. 41(4):1-54.

56. **J.Bhasker.** *A SystemC Primer.* USA : Star Galaxy Publishing, 2004.

57. **Lattner, Chris.** *The LLVM Compiler System.* s.l. : Bossa International Conference on Open Source, Mobile Internet and Multimedia, 2007.

58. **Lv, Tao, et al.** *An Observability Branch Coverage Metric Based on Dynamic Factored Use-Define Chains.* Beijing, China : Institute of Computing Technology, Chinese Academy of Sciences, 2006.

59. **Sirpatil, Brijesh.** *Software Synthesis of SystemC Models.* Blacksburg, Virginia : Virginia Polytechnic Institute and State University, 2002.

60. **Terei, David Anthony.** *Low Level Virtual Machine for Glasgow Haskell Compiler.* Sydney-Australia : The University of New South Wales, 2009.

61. **Yu, Ke.** *Real-Time Operating System Modelling and Simulation Using SystemC.* s.l. : The University of York, 2010.

62. **Pedroni, V. A.** *Circuit Design with VHDL.* s.l. : MIT Press, 2004.

63. **Fallah, F., Ashar, P. e Devadas, S.** Simulation Vector Generation from HDL Descriptions for Observability-enhanced Statement Coverage. *Proceedings of the 36th Design Automation Conference.* 1999, pp. 666–671.

64. **Fallah, F. , Devadas, S. e Keutzer, K. .** Functional Vector Generation for HDL Models Using Linear Programming and 3-Satisfiability. *Proceedings of the 35th Design Automation Conference.* 1998, pp. 528–533.

65. **Fallah, F. , Devadas, S. e Keutzer, K.** OCCOM: Efficient Computation of Observability-Based Code Coverage Metrics for Functional Simulation. *Proceedings of the 35th Design Automation Conference.* June 1998, pp. 152–157.

66. **Godefroid, P. , Klarlund, N. e Sen, K.** Dart: directed automated random testing. *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation.* New York, NY, USA : ACM Press, 2005, pp. 213–223.

67. **Costa, J. e Monteiro, J.** A MILP-based Approach to Path Sensitization of Embedded Software. *Proceedings of the Design Automation & Test in Europe.* Nice : s.n., 2009.

68. *Observability-based Coverage-directed Path Search using PBO for Automatic Test Vector Generation.* **Costa, J. e Monteiro, J.** October de 2009, Proceedings of the 17th IFIP/IEEE International Conference On Very Large Scale Integration (VLSI-SOC).

69. **Edmonds, J. e Johnson, E. L. .** Matching, Euler Tours and the Chinese Postman. *Mathematical Programming.* 1973, pp. 5(1):88–124.

70. **Eiselt, H. A., Gendreau, M. e Laporte, G. .** Arc Routing Problems, Part I: The Chinese Postman Problem. *Operations Research.* 1995, pp. 43(2):231–242.

71. **Thimbleby, H. W. .** The Directed Chinese Postman Problem. *Software Practice and Experience .* 2003, pp. 33(11):1081–1096.

72. **Refsnes Data.** XML Tutorial. *W3Schools.com.* [Online] [Citação: 2 de March de 2015.] http://www.w3schools.com/xml/default.asp.

73. *GRAPH FILE FORMATS.* [Online] [Citação: 2 de March de 2015.] http://www2.sta.uwi.edu/~mbernard/research_files/fileformats.pdf.

74. **Traulsen, Claus, et al.** A SystemC/TLM Semantics in Promela and Its Possible Applications. [Online] http://www-verimag.imag.fr/~moy/publications/MicMacToSpin.pdf.

75. **Holzmann, Gerard.** *The SPIN Model Checker: Primer and Reference Manual.* s.l. : Addison-Wesley Professional, 2004.

Annex A. sc_main function

```
1:  #include "systemc.h"
2:  #include "somador.h"
3:
4:  int sc_main(int argc, char **argv) {
5:
6:      sc_signal<bool> reset;
7:      sc_signal<sc_int<16> > operador1;
8:      sc_signal<sc_int<16> > operador2;
9:      sc_signal<sc_int<16> > operador3;
10:     sc_signal<sc_int<16> > resultado;
11:     sc_signal<sc_int<16> > resultado2;
12:
13:     sc_clock clk("clk",10,SC_NS);
14:
15:     somador16b som1("som1");
16:        som1.clock(clk);
17:        som1.reset(reset);
18:        som1.opr1(operador1);
19:        som1.opr2(operador2);
20:        som1.resultado(resultado);
21:
22:     somador16b som2("som2");
23:        som2.clock(clk);
24:        som2.reset(reset);
25:        som2.opr1(resultado);
26:        som2.opr2(operador3);
27:        som2.resultado(resultado2);
28:
29:     operador1 = 10;
30:     operador2 = 40;
31:     operador3 = 2;
32:
33:     cout << "Starting SIM" << endl;
34:     reset = 0;
35:
36:     sc_start(200,SC_NS);
37:     reset = 1;
```

```
38:     cout << "Reset!" << endl;
39:     sc_start(200,SC_NS);
40:     reset=0;
41:     sc_start(2000,SC_NS);
42: 
43:     cout << "resultado = " << resultado << endl;
44:     cout << "resultado2 = " << resultado2 << endl;
45:     cout << "SIM done" << endl;
46: 
47:     return 1;
48: }
```

Annex B. Module somador16b

```
1:  #include "systemc.h"
2:
3:  struct somador16b : public sc_module
4:  {
5:      // input ports
6:      sc_in_clk clock;
7:      sc_in<bool> reset;
8:      sc_in<sc_int<16> > opr1;
9:      sc_in<sc_int<16> > opr2;
10:
11:     // output ports
12:     sc_out<sc_int<16> > resultado;
13:
14:     //local variables
15:     sc_int<16> temp;
16:
17:     //CONSTRUCTOR
18:     SC_HAS_PROCESS(somador16b);
19:     somador16b(sc_module_name name) : sc_module(name)
20:     {
21:         SC_THREAD(som);
22:         sensitive << reset.pos();
23:         sensitive << clock.pos();
24:     }
25:     //module function
26:     void som(){
27:         int i = 1;
28:         while(i == 1){
29:             if(reset == 1){
30:                 resultado.write(0);
31:             }else{
32:                 temp = opr1.read()+opr2.read();
33:                 cout << temp << endl;
34:                 resultado.write(temp);
35:             }
36:             wait();
```

```
37:             }
38:         }
39:     };
```

Annex C. Example of generated file from Extract Struture

```
### list of Modules ###
0     somador16b
### list of Instances ###
0     0     som1
1     0     som2
## list of CFGs ##
0     sc_main
1     _ZN10somador16bC1EN7sc_core14sc_module_nameE
2     _ZN10somador16bD1Ev
3     _ZN10somador16bD2Ev
4     _ZThn40_N10somador16bD1Ev
5     _ZN10somador16bD0Ev
6     _ZThn40_N10somador16bD0Ev
7     _ZN10somador16bC2EN7sc_core14sc_module_nameE
8     _ZN10somador16b3somEv
## list of functions per Module ##
0     1     2     3     4     5     6     7
## list of Module functions##
0     8
## list of edges ##
0     -1    0
1     -2    0
2     -7    0
3     -7    0
4     0     1
5     -1    1
6     -2    1
7     -7    1
8     1     -6
### list of Signals ###
0     reset  bool
1     operador1    int16
2     operador2    int16
3     operador3    int16
```

```
4      resultado    int16
5      resultado2   int16
6      clk    int
### list of Ports ###
0      som1   clock  sc_in_clk    6
1      som1   reset  sc_in  0
2      som1   opr1   sc_in  1
3      som1   opr2   sc_in  2
4      som1   resultado    sc_out 4
5      som2   clock  sc_in_clk    6
6      som2   reset  sc_in  0
7      som2   opr1   sc_in  4
8      som2   opr2   sc_in  3
9      som2   resultado    sc_out 5
## list of Defines ##
```