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 SystemC 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

I. INTRODUCTION

Embedded systems are used in a growing number of diverse applications. Examples include consumer electronics, automotive systems and telecommunications, among others. This prevalence is due to the fact that embedded systems result from a mix of hardware/software systems. The software part, which runs on a processor, gives the system the flexibility, since it can be easily changed depending on the application. The hardware portion, which executes more specialized functions, is used in time critical subsystems.

Validation of embedded systems is hard because of their heterogeneity. Software and hardware should be simulated simultaneously, and furthermore, hardware and software simulations must be kept synchronized, so that they behave as close as possible to the physical implementation.

But validation of embedded systems is a complex problem where most of the times it is impossible to guarantee, regardless of the approach used, a 100% bug free system. Nevertheless, we can always try to eliminate the largest possible number of bugs and thus produce more reliable systems. With the increased complexity of embedded systems, testing has been a problem that needs to be more explored. There are several techniques and metrics for software testing and hardware testing. Because embedded systems are a combination of these two components, we can combine techniques and metrics from both areas and apply to an embedded system. One tool that is very useful when testing a system is a graph description of it.

SystemC [1] is a C++ library, which let us make use of all the faculties of C++, adding the necessary components for hardware emulation. It allows the specification of hardware/software, using a well-known language, C++. SystemC has a wide acceptance by the scientific community. It is used as a standard for specification, modelling, simulation and verification. Whatever the aim in which a SystemC tool is developed, it is always necessary to deal with the intricacies of C++.

The objective of this project is 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 SystemC description of the system and a coverage level that needs to be achieved. In the scope of this project, we implemented and validated the components SysCFG, Extract Structure and presented a viable solution for the implementation of the SysSMT component. Unfortunately SysSMT component was not developed due to delays that occurred during the development of other components. This article has the following structure. In Section II we give an overview of SystemC, PinaVM and LLVM. We also mention briefly some solutions for validation of embedded software. In Section III we present the architecture of our tool. In Section IV, we apply our method to an echo cancellation system developed in SystemC and present some results. We end with conclusions and future work in Section V.

II. RELATED WORK

In this Section we introduce some of the concepts and methods that we use in our tool. We also present some solutions for validation of embedded software.

A. SystemC

SystemC [1] 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). These are fundamental concepts that are scattered among several languages. It has been developed by the Open SystemC Initiative (OSCI) since 1999 and is now an IEEE standard. SystemC was designed for simulation but can also be used for formal analysis. SystemC allows the system to be modelled in different levels of abstraction, from the function description to
the cycle-accurate modelling. This brings great advantages 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 these reasons, SystemC does not introduce any new syntax to C++, but allows 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 [2], LLVM [3], EDG [4] or Visual C++ [5], just by adding the SystemC libraries for generating an executable of the program.

The executable is not always the best way to analyse a program. There are applications that need to make changes, analyses or optimizations before the generation of the executable. In these cases it is necessary to use a front-end created especially to cope with specificities of the language. A SystemC front-end may have several objectives: hardware synthesis, optimized compilation, symbolic formal verification, source code instrumentation or verification. There are several SystemC front-ends on the market [6], but all of them have different approaches, accompanied by their 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 these two parts. Another decision to make is the format for the intermediate representation (IR). This 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. 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 that supports 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 that 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 [6].

SystemC had a wide acceptance by the scientific community, because it brings together in a single language many essential hardware concepts and software, using C++. It also has a slight learning curve since it is C++.

B. LLVM

Low Level Virtual Machine (LLVM) [3] 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 [7]. 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 also 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, easily to incorporate with other tools. This innovative approach is faster than GCC and consumes less memory [8] [9].

LLVM has a classical architecture of three-tier as illustrated in Figure 1. 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).

LLVM’s 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. 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. It 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.

C. PinaVM

PinaVM [10] is an open source SystemC front-end for LLVM, with the purpose of making formal and symbolic verification of SystemC programs. It can also be used for other applications quite easily. PinaVM is based on the Pinapa [11] 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, an approach used 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 and 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 single static assignment (SSA) form [2], the LLVM bytecode. Thanks to PinaVM approach we can extract accurate information from the system architecture and the behavior of its components, without major limitations.
In practical terms, the use of PinaVM in SysCFG will enable our parse to extract the original instructions from the LLVM bytecode, without great difficulty.

D. Validation of Embedded Software

We intend to build a tool for co-validation of software/hardware description. Several tools have been proposed [12] [13], but all have their origins in hardware or software techniques [14]. 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.

In [15], 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 [16], 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 [17];
- MILP module that extracts the input vectors of a given path using the lp_solve [18] 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.

III. ARCHITECTURE

After introducing the related work to this project, we will now present the developed solution. In the following sections, we will present the tool, describing the various components and tasks.

A. 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’s not possible to achieve this coverage, the tool will return the higher coverage that it found.

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 project solution combines hardware and software testing techniques and it is based on a validation method for embedded software described in C [15]. We extended the idea to embedded systems described in SystemC. 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, where new components had to be implemented and incorporated with existing ones. Figure 2 shows the architecture’s solution of this project.

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

1) SysCFG: SysCFG (SystemC to CFG) aims to extract from a description in SystemC, the Control Flow Graph (CFG) 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 program’s LLVM IR. 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;
- Convert the instructions of LLVM IR to SMTs.

By cataloging only the instructions of the source code, it will provide a basis to construct the CFG function and calculate...
the longest path. 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 III-A4).

Finally in the Back-end we will build the CFGs 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).

This component is part of the scope of this project and it was implemented and validated.

2) 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 III-A3.

If the constants defined in #define directives appear in the SMTs assertions, we will replace them with the associated value. 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 project and it was implemented and validated.

3) IVG: Input Vector Generator (IVG) is an imported component of the solution created by Costa at [15] and briefly described in Section II-D. 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 3 shows the architecture of IVG. The IVG consists of three components: GenerateCFG, GenerateDAG and GeneratePaths.

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 the 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 (see Section II-D) used 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) [20] [21]. 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 [22], 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 [22] 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.

This component is not part of the scope of this project and at the moment is still under developed.

4) 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;
- “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 [23], dynamic execution [24] or using a combination of both [25] [26]. 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 [27]. 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 [15] 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.

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 "fileSMT2.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 "fileSMT2.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.

The solution of the SysSMT component was created within the scope of this thesis.

5) **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 [28] 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 consider as reference is the one based on observability.

This component is not part of the scope of this project but it has developed as part of the research project PTDC/EEA-ELC/122756/2010 – Cervantes Co Validation Tool for Embedded Systems.

### IV. Results

We applied our method to an echo cancellation system developed in SystemC. This is an embedded system, based on adaptive filtering. We develop our tool in C++ and used LLVM-3.2 and its compatible version of PinaVM. We tested our tool in an Intel® Core™ i7 running at 3.4GHz with 8GB of memory.

**A. Echo Cancellation System**

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.

**B. SysCFG Results**

Since this system is quite extensive, it is not possible to show here the CFG of the entire system. Thus we present in this section some statistics of the system and the resulting graph.

**SysCFG** takes an average of 9 seconds to parse the echo cancellation system. The SystemC description of the echo cancelation system consists of 1240 lines of code. The resulting CFG is composed of 1077 vertexes and 1114 edges. Of these vertexes, 9 are while statements, 28 are if conditions and 2 are for loops.

To show the effect of the different phases of our tool on the embedded system of the echo cancelation system, we show in Table I the number of functions and instructions collected at the end of each phase of the **SysCFG**. As can be seen from Table I, we start with a very large number of functions and instructions. This number consists not only of the functions and instructions of the echo cancellation system but also of
the ones internal to C/C++ and SystemC. From Table I we can see that at each phase we refine the number of instructions and functions that really matter for the system that is being analyzed.

<table>
<thead>
<tr>
<th></th>
<th>Front-end</th>
<th>Middle-end</th>
<th>Back-end</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functions</td>
<td>1132</td>
<td>73</td>
<td>73</td>
</tr>
<tr>
<td>Instructions</td>
<td>15776</td>
<td>5587</td>
<td>1053</td>
</tr>
</tbody>
</table>

1) 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. Do not use
  - 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;

C. Extract Structure

The Extract Structure component consists of a parse to the test program source code. 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 tested with the echo cancellation system and the generated file contains 1745 lines.

We can say that all the necessary information has been successfully collected, because we inspected the content of the generated file manually.

D. SysSMT

The SysSMT component approach 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 briefly described in Section III-A4, we created files with extension "smt2" and manually inserted the assertions into the files. Then our program injected the contents of the files on the Z3 solver to obtain the input vectors. This allowed us to test that assertion injection from a file is possible in our solution. This result came to validate the approach presented.

V. CONCLUSION

In this project 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.

The tool proposed here consists of five components, where the implementation of SysCFG, Extract Structure and SysSMT are part of the scope of this project. Section III describes the tool architecture, describing its components and the interaction between them.

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 that 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.

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 project) 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 discovered 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 a 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.

A. Future Work

As future work, we propose to implement the solution presented in section 3 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 briefly described in Section III-A4.

ACKNOWLEDGMENT

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.

REFERENCES