Data Extraction from SystemC Designs Using Debug Symbols and the SystemC API

Jannis Stoppe* Robert Wille*† Rolf Drechsler*†
*Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
†Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
{stoppe,rwille,drechsle}@informatik.uni-bremen.de

***Abstract*—Due to the ever increasing complexity of hardware and hardware/software co-designs, developers strive for higher levels of abstractions in the early stages of the design flow. To address these demands, design at the *Electronic System Level* (ESL) has been introduced. SystemC currently is the "de-facto standard" for ESL design. The extraction of data from system designs written in SystemC is thereby crucial e.g. for the proper understanding of a given system. However, no satisfactory support of reflection/introspection of SystemC has been provided yet. Previously proposed methods for this purpose either focus on static aspects only, restrict the language means of SystemC, or rely on modifications of the compiler and/or parser.**

In this work, we present an approach that overcomes these limitations. A methodology is introduced which enables full extraction of the desired information from a given SystemC design without changing the SystemC library or the compiler. For this purpose, debug symbols generated by the compiler and SystemC API calls are exploited. The proposed system retrieves both, static and dynamic information. A comparison to previously proposed solutions shows the benefits of the proposed method, while its application is illustrated by means of a visualization engine.

I. Introduction

The complexity of the design of hardware and software systems, especially in the hardware/software co-design, is ever increasing. To counter this development, engineers strive for higher levels of abstractions which, eventually, led to the design of such systems at the *Electronic System Level* (ESL) [16]. For this approach, SystemC [21] currently is the "de-facto standard" [15], [22].

However, due to SystemC being a C++ library, it lacks the native support for reflection and introspection other system description languages (like Esys.net [4], [12], [13] or HJJ [10]) provide. Consequently, extracting meta information from a given system description is a crucial task as (1) the source code is hard to parse due to the complex structure and the variety of dialects of C++ and (2) the compiled binaries are basically stripped of all the information that is not needed for execution, but might be required for reflection and introspection.

Current approaches try to address these problems by providing solutions which

- focus on static aspects only [2], [3], [6], [7], [11] or
- execute the design using custom(ized) parsers, compilers, and/or SystemC libraries that only support a restricted sub-set of SystemC [8], [9], [19].

This results in limitations concerning the SystemC constructs being used.

In this work, we introduce a methodology for information extraction of SystemC designs which addresses these problems. The general idea is to exert available data and interfaces as much as possible in order to avoid any restrictions and dependencies. More precisely, debugging symbols generated by an off-the-shelf compiler in combination with the existing SystemC API are applied for this purpose. Both, static and dynamic information is thereby retrieved.

Using the proposed approach, restrictions and limitations of the previously introduced solutions are avoided. The proposed approach considers the established criteria for SystemC design extraction [19]:

- As few as possible a priori limitations are assumed,
- precise information on all parts of a given system are provided,
- code reuse is maximized in order to avoid creating a new C++ dialect and to ensure that the solution is also applicable to future SystemC compilers, and
- high level TLM constructs are supported.

A comparison to previously introduced approaches confirms the benefits of our approach. Particularly with respect to a priori limitations and code reuse, our approach clearly satisfies the desired criteria. The precision ultimately depends on the amount of details provided in the debug symbols. However, the amount of data that can be gathered is more than sufficient for tasks like system understanding. The respective pros and cons of the proposed approach are discussed in more detail later in Section VI-A. Furthermore, the applicability of the proposed approach is demonstrated by means of a visualization engine.

The remainder of this work is structured as follows: In order to keep the paper self-contained, the next section briefly reviews the basics on SystemC and introduces a running example to be used in this work. Section III reviews and discusses previously introduced solutions and, therefore, forms the motivation of this work. Afterwards, Section IV introduces the general idea of the proposed approach while Section V provides details on its implementation. Results of the evaluation, i.e. the comparison to previous work as well as the application of the extracted results in a visualization engine, are reported in Section VI. Finally, the paper is concluded in Section VII.

II. SystemC

SystemC is a C++ library for modeling and simulating system designs. By providing descriptions means for both, hardware concepts (like modules, signals, ports, etc.) and software concepts (like class instantiations, function calls, memory allocation, etc.), it allows to model and to execute hardware and software systems on various levels of abstraction. While modules (representing parts of a hardware system) and their connections are instantiated, the logic behind those can be both, made up of simulated hardware elements down to gate level or just a software simulation of the behavior that is supposed to be realized in hardware later on. SystemC is considered an "industry standard" [22] and is widely used to prototype hardware/software systems and co-systems as well as their behavior.

```
#include <systemc.h>
SC_MODULE(fullAdder) {
        sc_in<bool> a, b, carryIn;
        sc_out<bool> result, carryOut;
        void calculate() {
                carryOut.write((a && carryIn) || (b &&
                     carryIn) || (a && b));
                result.write((a && !b && !carryIn) || (!a
                     && b && !carryIn) ||
                         (!a && !b && carryIn) || (a && b &&
                              carryIn));
        }
        SC_CTOR(fullAdder): a("a"), b("b"), result("result"
             ), carryOut("carryOut") {
                SC_METHOD(calculate);
                sensitive << a << b;
        }
};
int sc_main (int argc , char *argv[]) {
        int bits = 2;
        if (argc > 1)
                bits = max(0, min(16, atoi(argv[1])));
        fullAdder* previous;
        for (int i = 0; i < bits; i++) {
                fullAdder* fa = new fullAdder("fullAdder");
                if (i > 0) {
                        sc_signal<bool>* sig = new
                             sc_signal<bool>("carrySignal");
                        previous->carryIn(*sig);
                        fa->carryOut(*sig);
                }
                previous = fa;
        }
        return 0;
}
```

Fig. 1. SystemC program

Example 1. *Fig. 1 shows a SystemC program which is used as running example in the remainder of this paper. The program realizes a simple carry-ripple adder. The bit-width of the adder is not statically defined, but will be provided by the user when executing the program. This is realized by iteratively instantiating new one-bit full adders.*

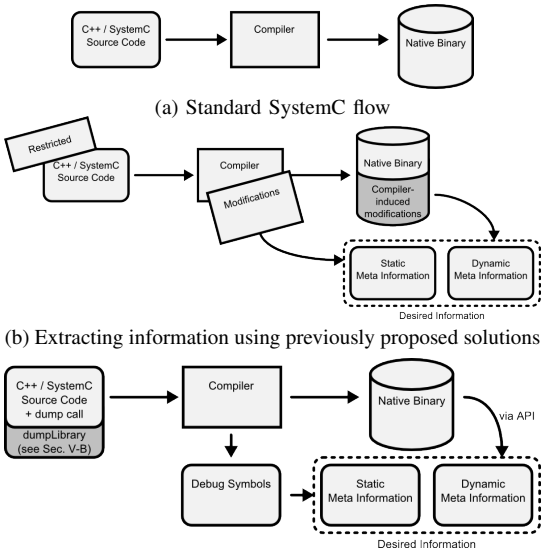
III. Motivation

SystemC enables to implement and simulate complex hardware/software systems. Fig. 2a briefly shows the corresponding flow which is basically identical to the typical software design flow: The desired system is implemented in SystemC and compiled by an off-the-shelve compiler. Finally, the resulting binary can be executed to simulate the system.

However, to be able to further process a SystemC design beyond its mere simulation, information concerning the structure of the design needs to be extracted. Due to the lack of native support for reflection and introspection, researchers developed several approaches concerning the extraction of information from a given SystemC design. Basically, they can be categorized as follows [15]:

- *Static Approaches* extract information by interpreting the source code without executing it. This allows for an extraction of all static information of the design including e.g. classes, operations, etc. For this purpose, either *custom* C++/SystemC parsers (e.g. [3], [6], [7], [11]) are applied or the intermediate representation of existing C++ compilers (e.g. the *Abstract Syntax Tree* (AST)) is exploited from which the desired information is obtained [19].
- In addition to a static data extraction, *Hybrid Approaches* (e.g. [5], [8], [9], [19]) additionally consider the dynamic behavior of a given SystemC program. For this purpose, the considered SystemC description is executed and the desired dynamic information is obtained during run-time.

All existing approaches are thereby subject to serious restrictions and limitations. Obviously, static approaches inherently suffer from a purely static analysis only. Dynamic constructs of the systems like loops, recursions, user defined variables, etc. often can hardly be extracted [15]. Particularly, if the program depends on user input (e.g. the parameter specifying the bit-width in the carry-ripple adder from Fig. 1), this information cannot be grasped from the static code at all.



(a) Standard SystemC flow

(b) Extracting information using previously proposed solutions

(c) Solution proposed in this work

Fig. 2. Extracting information from SystemC designs

Hybrid approaches usually cover these drawbacks by additionally taking into account run-time information. Most existing approaches rely on customized, i.e. non-standardized, tools. As an example, *Pinapa* [15], [19] uses a "slightly modified version of SystemC" and requires "a patch to the GNU C++ compiler" [20]. The approach presented in [8] uses a PCCTS based parser, which effectively restricts the descriptions means of SystemC. Hence, current hybrid approaches do not fully support the entire instruction set of C++. Moreover, these approaches are written specifically for the existing sub-set of SystemC – in order to support future releases of SystemC, they need to be re-developed.

Overall, existing approaches either suffer from

- being focused on static aspects only,
- need a customized adaption of SystemC and, therefore, are not applicable e.g. to future releases of SystemC, or
- rely on customized compilers/parsers which limit the applicable language constructs.

This is also briefly summarized in Fig. 2b.

Even recently published hybrid systems *PinaVM* [14] and *SHaBE* [5] are applying complex measures to extract system designs that cannot be easily ported to other project structures. *SHaBE* "uses the GNU debugger (GDB) to retrieve the module hierarchy from a SystemC model" and a GCC plugin to retrieve static information. While this results in all GCC-supported constructs being supported as well, not all compilers offer a plugin interface to modify the compilation process or a scriptable debugger, making the approach inapplicable for other setups. *PinaVM* relies on the intermediate representation that is generated by the LLVM compiler for execution and static extraction, making the approach only seem applicable for setups that rely on IRs (e.g. C++/CLR or LLVM).

While [1] recognized these flaws and presented a solution that left the SystemC core untouched, the reflection and introspection capabilities are available only through a Python wrapper.

In this work, we are aiming at overcoming the given drawbacks. More precisely, we consider the question:

> *How to extract the desired information from a given SystemC design without assuming restricted language means and without modifying the existing infrastructure like parsers or compilers?*

IV. General Idea

In order to address the research question stated above, we are exerting the data and interfaces that are available in SystemC/C++ design as much as possible. In particular,

- C++ compilers generate debug symbols from which relevant (static) information of the considered design can be obtained and
- the SystemC API allows for an extraction e.g. of values from data-structures during the execution of a program

are being used.

In this work, we propose an approach which exploits these existing interfaces for a generic and flexible information extraction of SystemC designs. More precisely, we use the debug symbols that are generated during compilation anyway to extract the static meta information of SystemC programs. By this, we can avoid modifying a compiler (e.g. to dump the accumulated information) as all desired information can also be extracted from these symbols.

Dynamic information can obviously be retrieved only during the execution of a program as some values might not be known at compile-time. But instead of trying to retrieve this information by modifying the given design (e.g. to dump values currently assigned to signals) or by deduction from the static information, we again make use of the existing infrastructure. In fact, the SystemC API is exploited to extract the desired information during run-time, making the solution for dynamic extraction independent from the platform or the compiler being used.

As a result, information extraction of SystemC designs can be conducted as illustrated in Fig. 2c. Instead of modifying the compiler and working with a binary extended by additional information needed for information extraction only, the existing data structures and interfaces are exploited (namely the debug symbols and the SystemC API). By relying on this existing infrastructure, the proposed solution is flexible, quite independent from compiler versions, and fully supports the whole range of SystemC.

In the next section, the proposed approach is described in detail.

V. Implementation

Two different modules have been implemented to realize the ideas proposed above: The first module reads the compiler-generated debug symbols and extracts static information of the design from it. The second module is a C++ library which can be called during run-time to export SystemC objects that are currently residing in memory. This allows for an extraction of the dynamic information. Afterwards, the extracted dynamic information is matched with its static counterpart.

A. Extraction of Static Information via Debug Symbols

Existing approaches for the extraction of static information rely on parsing the code using a custom program or interpreting the intermediate language of a given compiler. In the proposed solution, debug symbols of the compiled program are exploited instead. Such debug symbols are created by almost every modern compiler and contain meta information of the code which are usually applied to aid the designer in developing and debugging his/her implementation. For this purpose, the compilers collect and build extensive data about the program which, after the compilation, is written to the disk so that the debugger can use them. However, in a similar fashion, the desired static information can be extracted from these symbols.

```
.stabs "fullAdder:Tt(0,4872)=s5332!1,020,(0,1927);a
    :(0,1979),736,448;b:(0,1979),1184,448;carryIn:(0,1979)
    ,1632,448;result:(0,3933),2080,480;carryOut:(0,3933)
    ,2560,480;␣...
```

Fig. 3. Debug symbols in the STABS format

Example 2. *Fig. 3 shows some debug symbols in the STABS format as they have been generated by the GNU compiler using the code from Fig. 1. As can been seen, relevant static information of the design (e.g. the* fullAdder *class and its fields) can be recognized. This kind of information is available to a very deep level, including access modifiers of fields, a function's lines in the source code, function parameter types, base class information, size of types, etc. Furthermore, many debug symbols are arranged in a hierarchical manner, i.e. a debug symbol may be composed of several sub-symbols.*

In the following, we show how the information available through these debug symbols can be exploited to extract the desired static information of a given system. Therefore, we use the Microsoft VC++ compiler and the *Program Database*-files (PDB-files) generated by it [18][1]. Additionally applying the *Debug Interface Access* (DIA) SDK [17], the debug symbols that are collected in those files can be accessed.

Fig. 4 shows the general procedure of the extraction. Given a PDB-file created by the VC++ compiler from a SystemC design, all topmost debug symbols are loaded first (line 2). Afterwards, each symbol is separately considered and analyzed (lines 3-5). The desired information is thereby dumped into an XML-data structure representing the static information of the system. Since the debug symbols are hierarchically structured, the analysis is recursively conducted through the function *analyzeSymbol* (line 7). Here, all the desired information of the currently considered debug symbol are dumped to the XML-data structure first (line 8). Afterwards, it is checked whether further hierarchical information is available (lines 9-17). If this is the case, the corresponding sub-symbols are analyzed by recursively calling *analyzeSymbol* for them. To avoid redundancies, lines 13 and 14 stop the recursion if types are found that are also part of the *symbolTable* and would otherwise be extracted several times.

Example 3. *Consider again the SystemC code from Fig. 1 to be analyzed. Using the PDB-file generated by the VC++ compiler, the analysis of the first debug symbol (line 8) results in an XML-tag like*

```
<userDefinedType name="fullAdder"
addressOffset="0" addressSection="0"
constType="0" length="428">
```

stating that the considered system contains the class fullAdder *(an instance of which occupies 428 bytes in memory). More information about this class can be gained by the analysis of the corresponding sub-symbols through the recursive calls*

[1]However, although we implemented the proposed solution with these tools, the same concept can be realized using other modern compilers as well.

```
1  function analyzeDebugData(filename) begin
2          symbolTable = loadDataFromDebugFile(filename)
3          for each symbol in symbolTable
4                  analyzeSymbol(symbol)
5          end for
6  end function
7
8  function analyzeSymbol(currentSymbol) begin
9          dumpAllData(currentSymbol)
10         if currentSymbol has typeInformation then
11                 analyzeSymbol(typeInformation)
12         end if
13         if currentSymbol has subSymbols AND
14                 currentSymbol is NOT baseClass AND
15                 currentSymbol is NOT typedef then
16                 for each subsymbol in subsymbols
17                         analyzeSymbol(subsymbol)
18                 end for
19         end if
20 end function
```

Fig. 4. Pseudo code of debug information extraction

(line 9-17). One of the fullAdder *class's sub-symbols contains e.g. information about its field "a" (i.e. the full adder's first input bit):*

```
<data name="a" [...] >
<type> <userDefinedType
name="sc_core::sc_in&lt;bool&gt;"[...]>
```

Note the field's name ("a") and the field's type (`sc_core::sc_in<bool>` with "<" being replaced by "<" and ">" by ">" respectively to keep the XML structure valid) in the description. This information is contained in its own debug symbol that is part of the former symbol. The hierarchy is encompassed by the recursion. More information could be gained by searching for the description of the given type itself. Other sub-symbols provide information e.g. on inheritance, functions, their parameters, etc.

Using this procedure allows for an exploitation of debug symbols, which are generated anyway, for the purpose of static information extraction. Compared to previously proposed solutions with their respective constraints and requirements, this results in a near-to-none setup as all the information is retrieved from existing compilers and tools.

B. Extraction of Dynamic Information via the SystemC API

The SystemC library comes with an API that provides not only means of virtually creating and simulating systems, but also allows for accessing and inspecting the created instances of a system during run-time. That is, SystemC itself is, in principle, able to deliver an overview of the dynamic information of the instantiated system. However, in addition to the dynamic features, the static information also needs to be extracted. Existing approaches exploiting this API for the extraction of dynamic information still rely on modifying the SystemC library [9] and, hence, only provide a limited and restricted solution.

We propose a solution that requires as few changes as possible to the existing setups by performing the following steps each time dynamic information should be retrieved:

1) Accessing the instantiated objects: The SystemC API provides a function to get access to the simulation context (via `sc_get_curr_simcontext()`) through which an `object_manager` can be retrieved (via `context->get_object_manager()`). The `object_manager` in turn provides access to all instantiated objects that are being used in the current run of the SystemC program.

2) Naming the retrieved objects: To name the retrieved objects, [9] named the instances based on the fields' names. That is, an object created by the SystemC line `fullAdder faField("faName")` would be named *faField*. This leads to serious problems as

- field names may be used more than once (at different locations in the program) and
- a single instance may be assigned to several fields and, hence, a single instance might be referred to by several different field names.

In order to overcome these problems, the proposed solution uses the respective SystemC name field for naming an instantiated object. That is, an object created by the SystemC line `fullAdder faField("faName")` would be named *faName*. As SystemC automatically renames duplicates and assigns names to unnamed objects, this solves the above mentioned problems. Moreover, as the name is chosen by the designer and is not required to comply with the rules of C++ variable names, the naming of the respective objects becomes more intuitive for the designer. Finally, this solution makes any parsing of the SystemC source code obsolete. Limiting modifications in the SystemC library (as done e.g. in [9]) can be dropped.

3) Mapping instances: In order retrieve a complete model of the given design, in a last step the extracted dynamic information is mapped to the static information gathered by the debug symbols[2] and/or other objects that are being used in the design. To comply with the non-invasiveness principle, the C++ and SystemC APIs are exploited:

- The types of the modules are retrieved by *Run Time Type Inspection* (RTTI) using the `typeid` operator (via `typeid(*module).name()`). As SystemC objects already have virtual methods, all SystemC objects provide a so called `vtable` during run-time, making the use of RTTI reasonable. This type name is the same as in the debug symbols, mapping the instances to their classes.
- Channels are differentiated from modules using an attempted cast to `sc_interface*` (via `if (dynamic_cast<sc_interface*>(module) != 0) isChannel = true`). This allows for channels to be marked as such during extraction albeit they usually behave like modules.
- Ports are matched either by the name of their channel (when being connected to one) (via `(dynamic_cast<sc_channel*>(chan))->name()`) or the memory address of the respective signal (via `reinterpret_cast<int>(chan)`). All ports that share the same signal address are considered to be connected depending on being either input, output or both. In contrast, ports that share the same channel instance's name are considered to be connected to the given channel.

The type names that are extracted by the SystemC API in this fashion exactly match those that are extracted from the debug symbols. Hence, the remaining mapping between the extracted static information and the extracted dynamic information is a simple comparison operation.

Incorporating these steps, the designer only has to specify at which point during the execution of the project, dynamic

[2]Note that e.g. [9] did not retrieve any static information and, hence, no such mapping at all was considered there.

```
<SystemCDesign>
 <module name="fullAdder_0" type="struct fullAdder">
  <In name="fullAdder.a"
    type="class sc_core::sc_in<bool>"></In>
  <In name="fullAdder.b"
    type="class sc_core::sc_in<bool>"></In>
   ...
```

Fig. 5. Result of the dynamic design extraction of SystemC

information should be extracted. This is realized by providing a function `dumpModulesToFile(string filename)`. Whenever this function is called during the execution of a SystemC program, the respective steps from above are performed and, similar to the extraction of the static analysis, the determined information is stored in an XML-data-structure.

Example 4. *Consider again the SystemC code from Fig. 1 to be analyzed. Assuming the designer is interested in all dynamic information available after the full adder has completely been instantiated. Then the designer only has to add the command* `dumpModulesToFile(string filename)` *after line 28 in the code from Fig. 1. Then, during the execution of the program, the respective API calls are conducted eventually leading to the XML-tags as partially shown in Fig. 5. From this, the designer can obtain the desired information, e.g. that there is an object called "fullAdder_0" of the type "fullAdder" with two inputs "a" and "b", each of which is of type* `sc_core::sc_in<bool>`.

VI. Evaluation

In this section, we evaluate the proposed approach. To this end, the methods introduced in Section V have been implemented as the tool *LENSE* (the Lightweight Expedient for Non-Invasive System Extraction).

There are two major components in this tool: the first one analyzes the debug symbols of the compiled SystemC programs as introduced in Section V-A, while the second one is the library to extract the dynamic information as introduced in Section V-B. Both have been implemented in C++, the former additionally using the DIA SDK to access the debug information, the latter using the SystemC library. While, in the current implementation, the static analysis is only applicable to Microsoft VC++ projects under Windows, dynamic extraction is platform independent and has been tested in Linux and Windows environments. Both modules build an XML structure that is written to disk for further usage.

In our evaluation we (1) discuss the characteristics and the applicability of *LENSE* and (2) illustrate the application of *LENSE* by means of a visualization scenario.

A. Discussion

According to [19], SystemC design extraction approaches should satisfy the following criteria:

1) Assuming as few a priori limitations as possible,
2) providing precise information on all parts of the program,
3) maximizing code reuse to avoid creating new C++ dialects and to ensure that the solution is also applicable to future SystemC compilers, as well as
4) being able to manage high level *Transaction Level Modeling* (TLM) constructs.

Concerning the **a priori limitations** (#1), our approach clearly outperforms previously proposed solutions. In fact, *LENSE* enables a full data extraction without interfering with the overall compilation process. While all approaches proposed so far rely on a modification of the applied compiler and/or the SystemC library, existing debug symbols and interfaces are exploited here (as illustrated in Fig. 2). Our hybrid approach does not rely on custom code annotations or language constructs being used. Also, we do not require the user to use a modified or custom compiler or parser (effectively limiting the language constructs being used). While the implementation currently only supports Microsoft VC++ debug symbols, the methodology is not limited to it and an implementation for alternative symbols (e.g. STABS or DWARF) are straight-forward.

With regard to the **precision** (#2), previously proposed solutions retrieve their information from intermediate representations like the AST. Our tool ultimately depends on the debug symbols generated by the compiler. This, however, ensures a significant amount of information concerning the structure of a program is available (e.g. information about class hierarchy, member functions, fields, visibility, etc.), including precise information concerning the individual lines in the source files that a particular element is made up of.

Concerning **code reuse** (#3) our approach, while having been implemented for a Microsoft VC++ environment, is applicable to basically any setup, requiring no changes in the user's code base at all (apart from a single function call to denote at which point during the execution dynamic information should be retrieved). We are exploiting the existing infrastructure of debug symbols (for static information) and the SystemC API (for dynamic information), leaving no dependency on versions, kinds of compilers, or SystemC libraries being used.

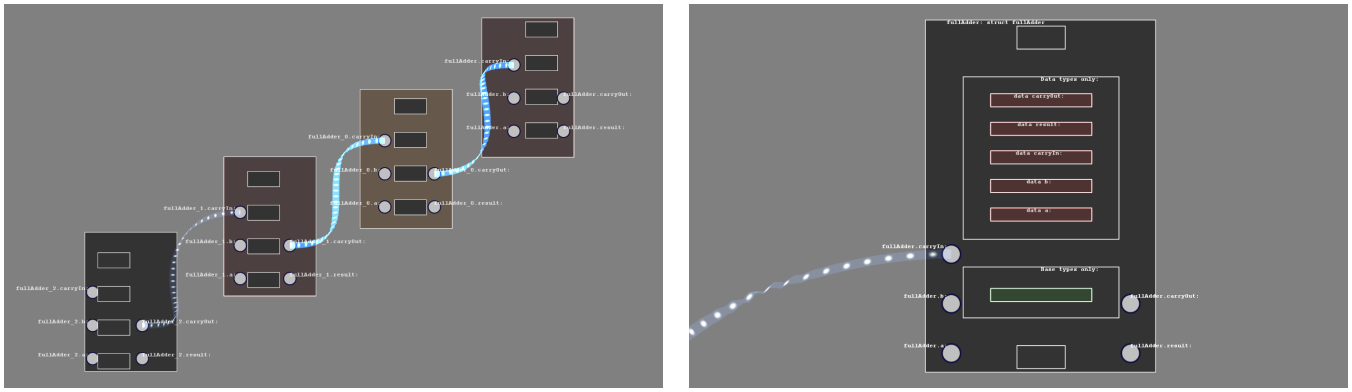
Our approach provides **TLM support** (#4). As our tool inspects the full class hierarchy, all objects that at some point inherit from or implement any SystemC class are recognized and extracted properly. As, however, the interesting part of TLM is usually the way information is being transferred in the system, we recognize a limitation of our approach as no run-time behavior of the simulation is being tracked yet. But this is no limitation of the methodology itself. A more detailed consideration of this aspect is left for future work.

Overall, the proposed approach improves the existing state-of-the-art particularly with respect to criterion #1 and criterion #3. However, also the precision (criterion #2) is high, but ultimately depends on the amount of details being provided by the debug symbols. The amount of data that can be gathered is still tremendous and more than suffices for tasks like system understanding. Finally, also TLM support (criterion #4) is provided, although this will be considered in more detail in future work.

B. Application

The extracted data was used to create a proof-of-concept visualization tool for SystemC designs. This tool reads the extracted XML files and merges them into a system visualization that combines the static information (i.e. the source code elements that were retrieved from the debug symbols) and the dynamic information (i.e. instances of modules, signals, etc. that were retrieved from the SystemC API) in one single view. This tool has been written in C#/XNA. The goal of this tool is to illustrate how the independently generated data from the different sources (static, dynamic) are merged and exploited.

Two possible views of the visualization are depicted in Fig. 6. More precisely, Fig. 6a shows the dynamic information retrieved by executing the carry-ripple adder program from Fig. 1 with a bit-width of 4. Each box represents thereby



(a) Global view (b) Closer view

Fig. 6. Visualization of the SystemC example

an instantitation of a 1-bit fullAdder with its corresponding fields and connections. A closer view to one of the boxes is provided in Fig. 6b. Here, the smaller boxes within the top box represent the respective fields of the `fullAdder` which have been obtained from the static meta information.

The approach is also applicable to complex SystemC designs. More precisely, the tool has successfully been tested on all examples that are delivered with the SystemC library. For all these designs, a corresponding visualization has been created within a few moments.

VII. Conclusion

In this work, an alternative approach to extract both, dynamic and static data from a SystemC program has been presented. Unlike previous approaches, the proposed solution neither requires modifications of the SystemC library nor does it have prerequisites concerning the compilation process (as long as debug files are generated). While the implementation itself is currently limited to the Microsoft VC++ debug format, this limitation is not systemic. A similar solution can be realized with other modern compilers as well. A comparison to previously proposed solutions illustrated the benefits of the proposed method. By means of a visualization engine, a possible application has been shown.

VIII. Acknowledgments

This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project *VisES* under contract no. 16M3197B and the German Research Foundation (DFG) within the Reinhart Koselleck project under contract no. DR 287/23-1.

References

[1] Giovanni Beltrame, Luca Fossati, and Donatella Sciuto. Resp: A nonintrusive transaction-level reflective mpsoc simulation platform for design space exploration. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 28(12):1857 –1869, dec. 2009.

[2] David Berner, Jean-Pierre Talpin, Hiren Patel, Deepak Abraham Mathaikutty, and Sandeep Shukla. SystemCXML: An extensible SystemC front end using XML. In *Proceedings of the Forum on Specification and Design Languages*, pages 405–409, 2005.

[3] Carlo Brandolese, Paolini Di Felice, Luigi Pomante, and Daniele Scarpazza. Parsing SystemC: an open-source, easy-to-extend parser. In *IADIS International Conference on Applied Computing*, pages 706–709, 2006.

[4] Olivier Brassard, Frederic Rousseau, Jean David, Mathieu Kastle, and El Aboulhamid. Automatic Generation of Embedded Systems with .NET Framework Based Tools. *2006 IEEE North-East Workshop on Circuits and Systems*, pages 165–168, June 2006.

[5] H. Broeders and R. van Leuken. Extracting behavior and dynamically generated hierarchy from systemc models. In *Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE*, pages 357 –362, june 2011.

[6] Javier Castillo, Pablo Huerta, and Jose Ignacio Martinez. An open-source tool for SystemC to Verilog automatic translation. *Latin American Applied Research*, 37(1):53–58, 2007.

[7] Görschwin Fey, Daniel Große, Tim Cassens, Christian Genz, Tim Warode, and Rolf Drechsler. ParSyC: an efficient SystemC parser. In *Workshop on Synthesis And System Integration of Mixed Information technologies*, pages 148–154, 2004.

[8] Christian Genz and Rolf Drechsler. Overcoming limitations of the SystemC data introspection. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 590–593, 2009.

[9] Daniel Große, Rolf Drechsler, Lothar Linhard, and Gerhard Angst. Efficient automatic visualization of systemc designs. In *Forum on Specification & Design Languages*, pages 646–658, 2003.

[10] John Hopf, G. Stewart Itzstein, and David Kearney. Hardware Join Java: a high level language for reconfigurable hardware development. In *International Conference on Field-Programmable Technology*, pages 344–347, 2002.

[11] FZI Karlsruhe. KaSCPar - Karlsruhe SystemC Parser Suite, 2012. http://www.fzi.de/index.php/de/component/content/article/238-ispe-sim/4350-kascpar-karlsruhe-systemc-parser-suite.

[12] James Lapalme, El Mostapha Aboulhamid, Gabriela Nicolescu, Luc Charest, Francois R. Boyer, Jean Pierre David, and Guy Bois. ESys. Net: a new solution for embedded systems modeling and simulation. *ACM SIGPLAN Notices*, 39(7):107–114, 2004.

[13] James Lapalme, El Mostapha Aboulhamid, Gabriela Nicolescu, Luc Charest, Francois R. Boyer, Jean Pierre David, and Guy Bois. .NET framework - a solution for the next generation tools for system-level modeling and simulation. In *Design, Automation and Test in Europe Conference*, pages 732–733, 2004.

[14] Kevin Marquet and Matthieu Moy. PinaVM: a SystemC Front-End Based on an Executable Intermediate Representation. In *Proceedings of the tenth ACM international conference on Embedded software*, pages 79–88. ACM, 2010.

[15] Kevin Marquet, Matthieu Moy, and Bageshri Karkare. A theoretical and experimental review of SystemC front-ends. In *Forum on Specification and Design Languages*, pages 124–129, 2010.

[16] Grant Martin, Brian Bailey, and Andrew Piziali. *ESL Design and Verification: A Prescription for Electronic System Level Methodology*. Morgan Kaufmann, 2007.

[17] Microsoft. Debug Interface Access SDK, 2010. http://msdn.microsoft.com/de-de/library/x93ctkx8%28v=vs.100%29.aspx.

[18] Microsoft. Program Database Files (C++), 2010. http://msdn.microsoft.com/en-us/library/yd4f8bd1%28v=vs.100%29.aspx.

[19] Matthieu Moy, Florence Maraninchi, and Laurent Maillet-Contoz. Pinapa: An extraction tool for systemc descriptions of systems-on-a-chip. In *Conference on Embedded software*, pages 317–324, 2005.

[20] Matthieu Moy, Florence Maraninchi, and Laurent Maillet-Contoz. LusSy: An open tool for the analysis of systems-on-a-chip at the transaction level. *Design Automation for Embedded Systems*, 10(2-3):73–104, 2006.

[21] O.S.C. Initiative. IEEE Standard SystemC Language Reference Manual. *IEEE Computer Society*, 2006.

[22] Carsten Schulz-Key, Markus Winterholer, Thomas Schweizer, Tommy Kuhn, and Wolfgang Rosentiel. Object-oriented modeling and synthesis of SystemC specifications. In *Asia and South Pacific Design Automation Conference*, pages 238–243, 2004.