SystemCXML: An Extensible SystemC Front End Using XML

David Berner, Jean-Pierre Talpin
Institut de Recherche en Informatique et Systèmes Aléatoires (IRISA/INRIA), France

Hiren Patel, Deepak Abraham Mathaikutty, Sandeep Shukla
Virginia Polytechnic and State University, FERMAT Lab., USA

Abstract

To draw full benefit of the possibilities that system level design languages such as SystemC offer, we require tools that enhance the design experience through visual representation of models, improved debugging facilities, integrated development environments, etc. the primary task in providing these tools is the parsing of SystemC source to directly extract the structural design information. In this paper, we present a front end for SystemC called SystemCXML that uses an XML-based approach to achieve the extraction of structural information from SystemC models, which can be easily exploited by different back end passes for analysis, visualization and other structural analysis purposes. Our unique approach uses the documentation system Doxygen and an Open Source XML parser. We demonstrate its extensibility by incorporating an automated graph generator that visualizes the SystemC module hierarchy.

1 Introduction

Industrial use of languages such as C/C++ for system level design are common for achieving fast simulation and realization of ideas and concepts. Many times industries create their own C/C++ based simulation environments for just this purpose. However, with the introduction of SystemC [9], the Open Source Consortium Initiative (OSCI) proposed a standardization for a C++ based hardware modeling language (HDL) with an extensive datatype library, free discrete event simulator, and useable with most C++ compilers. In addition, the inherent abstraction capabilities of C++ such as templates, classes, polymorphism, etc., further promote its use and improve design experience. For example, transaction level modeling or communication refinements are not easily accomplished with traditional HDLs such as Verilog and VHDL. The advantage of using SystemC is that all capabilities of C/C++ are available for the designer to use, in addition to the SystemC constructs. This allows intellectual property components (IPs) written in C/C++ to be easily integrated into designs. However, the proliferation of third party tools for visual modeling environments, debuggers, integrated development environments, call tracing etc. requires understanding the SystemC syntax and the structure of the models. For this, it is necessary to parse and understand the SystemC's structural information. By structural we mean the modules, their ports, signals that connect them and the structural hierarchy used in connecting the entire model.

As SystemC continues to gain momentum as an HDL, this demand for graphical user interface (GUI) based design approaches, integrated development environments (IDEs), and SystemC-specific debuggers, is rapidly growing. Some industry tools such as ConvergenSC system verifier [3] and Incisive functional verification [1] provide some of these functionalities. However, most tool vendors provide SLD tools with varying usability. Most of them, do not yet exploit all possibilities of system level design analysis and transformation. Furthermore, SystemC is simply a library of C++ classes and thus it is heavily dependent on the C++ compiler, making the extraction of structural or behavioral information a difficult task without altering the compiler and perhaps the SystemC source. Doing this would, however, limit the choice of compilers that support SystemC. We term any alteration to either the compiler, the SystemC libraries, or the SystemC language as `intrusive`. Hence, we aspire the extraction of structural and behavioral information from a SystemC model using an unintrusive methodology that is also made available to the public-domain community. As a step towards providing an unintrusive approach for interpreting a SystemC model, we discuss the use of extensible markup language (XML) based approach using Doxygen [4], a public-domain documentation system, and XML parsers for extracting structural information that we call SystemCXML.

In this paper, we describe a tool that simplifies SystemC parsing using the following tools: XML, Apache XML parsers [13], and Doxygen. We also provide an intermediate representation (IR) to

facilitate the access to the extracted information during the parsing phase. However, our focus is primarily on extracting structural information. In the current release version, we ignore behavioral information, disallowing use of it for some applications such as synthesis. However, our approach provides a lightweight and Open Source solution for *source-to-source translations*, *structural information extraction*, *model visualization* and *documentation*. The intermediate XML data format makes it easy to extend this solution by plugging in a different front end or back-end passes, or even simpler, by just adding an additional passes to the back-end.

2 Related Work

EDG [6] is popular commercial tool that enables SystemC parsing. EDG is a C/C++ front end that parses C/C++ and represents the source using an IR data structure. Multiple traversals through the data structure can be performed to extract the required information. Therefore, the structure of SystemC models is available for extraction from the IR along with the behavioral information of the model as well. Unfortunately, EDG is a commercial tool, and requires licenses for use and does not allow source code distribution.

SystemPerl's SystemC::Parser An alternative to EDG [6] is SystemPerl's `SystemC::Parser` module [12] that implements a SystemC parser and netlist generator using Perl scripts. Using the power of regular expressions, the task of recognizing SystemC constructs is made easy, and the Open Source nature of SystemPerl makes its use much more attractive. However, one major distinction between EDG and SystemPerl is that SystemPerl only extracts structural information and not behavioral. For most uses, structural information is sufficient, unless considering synthesis from SystemC which requires all the behavioral information as well. To our understanding, SystemPerl has some limitations. One of them is that it requires source-level hints in the model for the extraction of necessary information and the IR of SystemPerl cannot be easily adapted to other environments and purposes.

Pinapa [8] is a recently released Open Source SystemC front end that uses GCC's front end to parse all C++ constructs and infer the structural information of the SystemC model by executing the elaboration phase, which is a very attractive solution. SystemC's elaboration constructs all the necessary objects and performs the bindings after which a regular SystemC model begins simulation via the `sc_start` function. Instead, Pinapa examines the data structures of SystemC's scheduler and creates its own IR. Once the IR is constructed, the SystemC model executes. This solves the SystemC parsing issue, it requires, however, modifications of the GCC source code which makes it (i) dependent on changes in the GCC codebase, and (ii) forbids the use of any other compiler. Furthermore, the SystemC libraries also have to be changed for this to work.

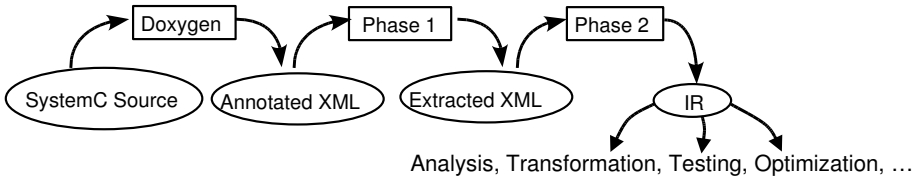


Figure 1: Toolchain of the SystemCXML project

3 SystemCXML Toolchain

The SystemCXML toolflow (Figure 3) consists of three steps. In the first step we process the SystemC code with Doxygen in order to generate an XML output that contains all the information of the original SystemC code, but embedded withing XML tags. While Doxygen is a tool for automatically generating documentation for projects through analysis of the source code, highlighting the structure and comments, it also has an option for including the source code in the generated output. Furthermore, since the output is a well-formed XML document, any standard XML parsing library can easily traverse it. We take advantage of this functionality and to approach the difficult problem of parsing C++ code. Listing 1 shows some raw XML output that Doxygen generates. Note that most of the information is delimited by tags, but there is need for some intelligence to extract it correctly.

The second step consists of reading in the Doxygen generated XML data with the help of the Xerces-C++ XML parser [13], extract the structural information of interest, and write this information back into an XML format in a way such that it is readily accessible for other purposes. We represent this extracted information in an Abstract System Level Description (ASLD) XML file. Listing 2 shows part of the ASLD, describing the structural information that was extracted from the Doxygen output shown in Listing 1.

Listing 1: Output from Doxygen

```
SC_MODULE(<ref refid="classfir__top">fir_top</ref>)<sp/>{
<codeline lineno="315" refid="classfir__top_1fir__topr0"><sp/>sc_in&lt;bool&gt;<sp/><sp/>CLK;
<codeline lineno="316" refid="classfir_top_1fir_topr1"><sp/>sc_in&lt;bool&gt;<sp/><sp/>RESET;
<codeline lineno="317" refid="classfir_top_1fir_topr2"><sp/>sc_in&lt;bool&gt;<sp/><sp/>IN_VALID;
<codeline lineno="318" refid="classfir_top_1fir_topr3"><sp/>sc_in&lt;int&gt;<sp/><sp/>SAMPLE;
<codeline lineno="319" refid="classfir_top_1fir_topr4"><sp/>sc_out&lt;bool&gt;<sp/><sp/>OUTPUT_DATA_READY;
<codeline lineno="320" refid="classfir_top_1fir_topr5"><sp/>sc_out&lt;int&gt;<sp/><sp/>RESULT;
<codeline lineno="324" refid="classfir_top_1fir_topr7"><sp/><ref refid="classfir__fsm">fir_fsm</ref><sp/>
<sp/>*fir_fsm1;
<codeline lineno="325" refid="classfir_top_1fir_topr8" ><sp/><ref refid="classfir__data">fir_data</ref>
<sp/>*fir_data1;
<codeline lineno="327" refid="classfir_top_1fir_topd0"><sp/>SC_CTOR(<ref refid="classfir__top">fir_top
</ref>)<sp/>fir_fsm1<sp/>=<sp/><ref refid="classfir__fsm">fir_fsm</ref><class="stringliteral">&quot;
FirFSM&quot;);<sp/>fir_fsm1-&gt;<ref refid="classfir__fsm_1fir__fsmr0">clock</ref>(CLK);</codeline>
```

Listing 2: Intermediate XML format

```
<module type = "SC_MODULE" name = "fir_top">
  <inport type = "bool" name = "CLK"/>
  <inport type = "bool" name = "RESET"/>
  <inport type = "bool" name = "IN_VALID"/>
  <inport type = "int" name = "SAMPLE"/>
  <outport type = "bool" name = "OUTPUT_DATA_READY"/>
  <outport type = "int" name = "RESULT"/>
  <submodule module="fir_fsm" name="fir_fsm1"/>
  <submodule module="fir_data" name="fir_data1"/>
  <constructorof modulename = "fir_top">
     <connection instance="fir_fsm1" member="clock" local_signal="CLK"/>
```

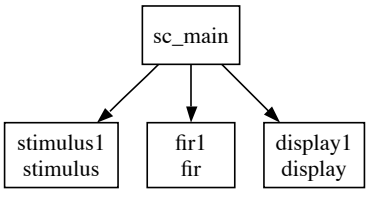
In the third step, we read in the ASLD and check if it conforms to the DTD description. We process the information and store it in an internal structure that is both, easily accessible and one that closely resembles the structure of the SystemC code. As the structure of the IR is the basis for all data manipulations and back-end passes, it is important for it to be generic and the appropriate accessory functions need to be implemented to query the IR. These functions make is possible to traverse the module hierarchy and extract the required structural information. The IR contains classes for all constructs that we extract such as *Inport*, *Outport*, *Signal*, *Sensitivity*, *Process*, *Signal*, and *Module*. Some information that is not readily available in the XML can be obtained by analyzing the available data. The *topmodules* list that points to modules that are not instantiated as a submodule, or the connection class that holds connectivity information are examples for this.

4 Usage Example: A Visualization Back End Pass

One possible usage of SystemCXML is graphical visualization. There are many different display possibilities that can help in better understanding a design for which most of the time the model's behavior is not needed at all. Visualization tools are especially helpful for design space exploration and semi-automated design refinements. In addition to that, the automatic generation of graphs and lists can be used in documenting system components, a step often neglected, leading to better collaboration and easing component reuse. In order to demonstrate the ease of creating such a visualization tool, we implement a back-end pass that generates a graph of the SystemC module hierarchy.

```
digraph fir_top
{
  node[shape=box];
  ratio=fill;
  sc_main;
  sc_main->"stimulus1\nstimulus";
  sc_main->"fir1\nfir";
  sc_main->"display1\ndisplay";
}
```

(a) DOT code



(b) Resulting graph

Figure 2: DOT code and resulting graph for the FIR filter

Among the libraries available for graph rendering, we use the DOT format from the graphviz [7] package to render our graphs. It is a comprehensive and easy to use package, which is used in many Open Source projects. Figure 2 shows the DOT code for the FIR filter example and the resulting graph. We use a *digraph* layout and choose boxed nodes whose width automatically adjusts to the length of the node label. The first occurrence of a node name creates the node. Directed connections are indicated with the "`->`" symbol. A DOT file can be visualized with *Dotty* the standard viewer which is part of Graphviz, but there many other tools for conversion and visualization of DOT files as [11].

To generate the graph, we start at the list of toplevel modules and then call the recursive function *submod_dot* that writes out the relations to all submodules and successively calls itself for all the submodules. As a node label we give the module name and the name of the instance. In order to keep a strict tree structure with no rejoining branches, all instances have to have different names which is not the case in the SystemC code. In order to avoid this we keep track of multiple instantiations of a module and distinguish between the respective submodules. Figure 3 shows part of the module hierarchy of a USB controller the code of which was obtained from OpenCores [2]. In the lower right hand corner you can see four instances of $usb\_fifo128x8$, containing an instance of $usb\_ram128x8$. These have been numbered in order to be able to distinguish them. The figure also shows that there is not only one connected graph but multiple graphs. Larger projects often contain multiple *sc_main* functions, used to individually simulate parts of the design in a separate testbench - like it is the case in this example. The visualization of the hierarchy helps to see all these parts of the design and understand their utility.

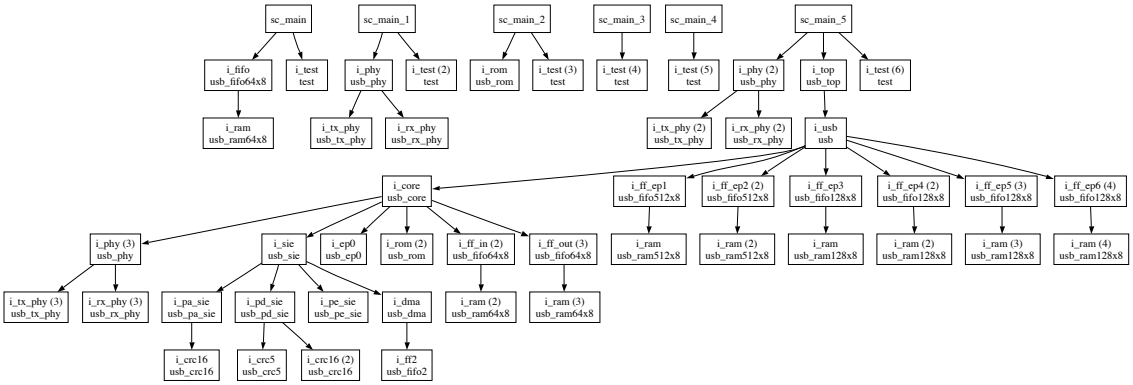


Figure 3: Visualization of the module hierarchy of a USB controller

5 Conclusion

Structural SystemC extraction is needed by many problem domains. We present SystemCXML that avoids the creation of a full fledged parser that handles the expressive power of C++ by the usage of Doxygen, breaking the problem down to parsing XML with widely available libraries. To our knowledge there is no existing solution that utilizes a documentation system to facilitate the extraction of structural information targeted towards applications such as introspective architectures, test generators and visualizations. This is a very lightweight solution, that employs C++ and does not restrict the use of the compiler or any modifications in the SystemC libraries remaining unintrusive. Our approach is built to be easily extensible; adding a backend pass for the module hierarchy graph generation only took about 60 lines of C++ code. There are many applications that can benefit from exploiting structural SystemC information, among them we have looked into visualization, automated testbench generation, and introspection [5] and are currently working on the generation of synchronous component interfaces. As we judge the tool to be easily useable and very beneficial for other research groups and companies, we made it available under an Open Source license available at [10].

References

[1] Cadence. Incisive Functional Verification. http://www.cadence.com.

[2] Open Cores. Free open source IP cores and chip design. http://www.opencores.org.

[3] CoWare. ConvergenSC. http://www.coware.com.

[4] Doxygen Team. Doxygen. http://www.stack.nl/~dimitri/doxygen/.

[5] Hiren D.Patel, Deepak A. Mathaikutty, David Berner, and Sandeep Shukla. CARH: An introspective and service oriented architecture for validating system level designs. *Accepted for publication in IEEE Transactions on Computer-Aided Design.*

[6] Edison Design Group C++ Front-End. Edison design group c++ front-end. Website: http://edg.com/cpp.html.

[7] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Softw. Pract. Exper.*, 30(11):1203–1233, 2000.

[8] GreenSocs. Pinapa: A SystemC front-end. Website: http://greensocs.sourceforge.net/.

[9] T. Groetker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC.* Kluwer Academic Publishers, 2002.

[10] D. Mathaikutty, D. Berner, H. Patel, and S. Shukla. SystemCXML SystemC Parser. http://systemcxml.sourceforge.net.

[11] Emmanuel Pietriga. Zgrviewer - a 2.5D graph visualizer for the DOT language. http://zvtm.sourceforge.net/zgrviewer.html.

[12] W. Snyder. SystemPerl. http://www.veripool.com/systemperl.html.

[13] The Apache Software Foundation. Xerces C++ validating XML Parser. Website: http://xml.apache.org/xerces-c/.