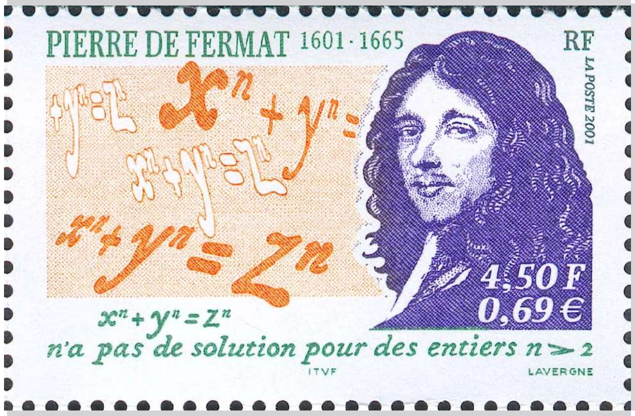

PIERRE DE FERMAT 1601 · 1665
RF
LA POSTE 2001
4,50 F
0,69 €
$x^n + y^n = z^n$
n'a pas de solution pour des entiers n > 2
ITVF
LAVERGNE

FERMAT

Formal Engineering Research using Methods, Abstractions and Transformations

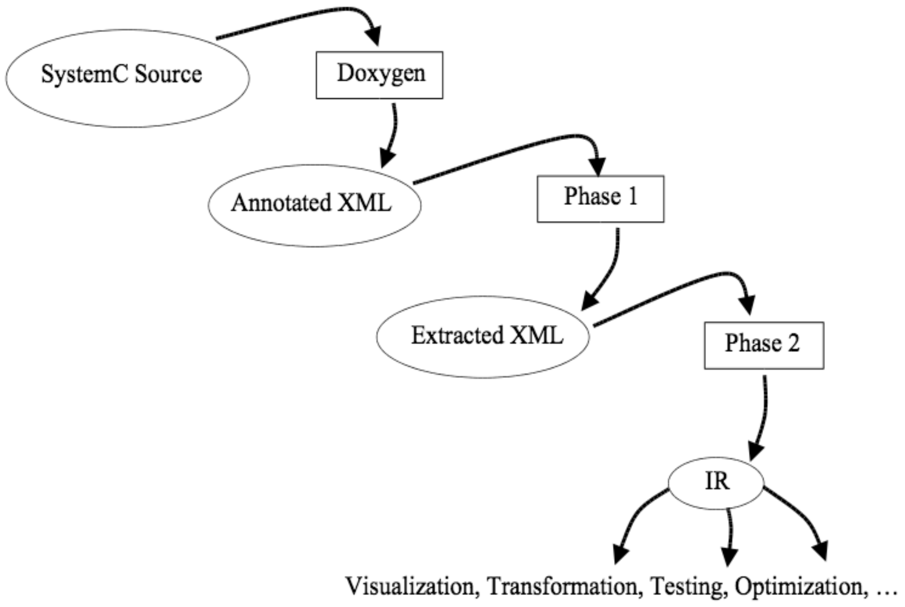
Technical Report No: 2005-06

SystemCXML: An Extensible SystemC Front End Using XML

Abstract -- The proliferation of system level design methodologies and frameworks is a direct result of the efforts in dealing with the productivity gap. Consequentially, System Level Design Languages such as SystemVerilog and SystemC are particularly fit for high level design methods such as design space exploration and assisted design refinement. However, to draw full benefit of these methods requires the introduction of tools that promote a better design experience by providing visual representation of models, better debugging facilities, integrated development environments, etc. In particular, we experiment with SystemC and realize that the foremost task in providing these tools necessitates the parsing of SystemC source to directly access the structural design information. It is desirable to perform the parsing in an unintrusive manner such that neither the model, the SystemC source, nor the compiler require alterations. In this paper, we present a front end for SystemC called SystemCXML that uses an XML-based approach to extract structural information from SystemC models, which can be easily exploited by 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, which is implemented in merely 60 lines of code.

David Berner, Hiren Patel, Deepak Mathaikutty,
Sandeep Shukla, Jean-Pierre Talpin,

{david.berner,talpin}@irisa.fr
{hiren,damathai,shukla}@vt.edu,



Virginia Tech

SystemCXML: An Extensible SystemC Front End Using XML

David Berner, Hiren Patel, Deepak Mathaikutty,
Jean-Pierre Talpin, Sandeep Kumar Shukla

INRIA, France
Virgina Tech, USA

{*dberner,talpin*}*@irisa.fr*
{*hiren,damathai,shukla*}*@vt.edu*

Contents

List of Figures

List of Tables

Abstract

The proliferation of system level design methodologies and frameworks is a direct result of the efforts in dealing with the productivity gap. Consequentially, System Level Design Languages such as SystemVerilog and SystemC are particularly fit for high level design methods such as design space exploration and assisted design refinement. However, to draw full benefit of these methods requires the introduction of tools that promote a better design experience by providing visual representation of models, better debugging facilities, integrated development environments, etc. In particular, we experiment with SystemC and realize that the foremost task in providing these tools necessitates the parsing of SystemC source to directly access the structural design information. It is desirable to perform the parsing in an unintrusive manner such that neither the model, the SystemC source, nor the compiler require alterations. In this paper, we present a front end for SystemC called SystemCXML that uses an XML-based approach to extract structural information from SystemC models, which can be easily exploited by 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, which is implemented in merely 60 lines of code.

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 [14], 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 [18] and VHDL [19]. 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 [4] and Incisive functional verification [2] 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 [5], a public-domain documentation system, and XML parsers for extracting structural information that we call SystemCXML.

Doxygen is a documentation system mainly for C/C++ projects with extensions that can also handle other languages such as Java. We use the capability of Doxygen to process C/C++ and generate documentation in the XML format. A large part of the tagging is undertaken by Doxygen that recognizes class declarations, class member variables, class member functions, global functions, and also tags the original source code in XML format. We leverage the XML output by employing XML parsers to generate an abstract system level design (ASLD) XML representation of the SystemC design. This is then used as the input for an internal representation (IR) to allow access to the extracted information.

In this paper, we describe a tool that simplifies SystemC parsing using the following tools: XML, Apache XML parsers [17], and Doxygen. We also provide an 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 to use 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.

There is a variety of opportunities for using SystemCXML. So far we use it for providing an introspective architecture for SystemC [13]. We also integrate this into a validation framework in [13] and our current efforts are focused on automatically generating synchronous component type interfaces for Signal [1]. We foresee the use of SystemCXML in visualization tools, graphical user interfaces and design space exploration tools with other back-end passes including analysis steps e.g. of the parallelism present in a design, or the re-scheduling of a component to meet different performance constraints. Another important application domain is the generation of visual information about a design, and in this document we demonstrate a simple module hierarchy graph generator.

The remainder of this paper is organized as follows. In Section 2 we present some related work and Section 3 explains the design-flow in terms of the different tools we use and how they integrate. We then present a back-end pass for the module hierarchy visualization in Section 4 and finally we draw some conclusion in Section 5.

2 Related Work

SystemC SystemC [14, 12] is an Open Source hardware description language (HDL) developed as a C++ class library. The purpose of SystemC is to allow for modeling and simulation of models at the RTL and higher levels of abstraction. Unlike many other HDLs such as Verilog [18] and VHDL [19], SystemC also provides a free simulator for designers to download and experiment designing models.

SystemPerl's SystemC::Parser An alternative to EDG is SystemPerl's `SystemC::Parser` module [16] 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.

EDG A popular commercial tool that enables SystemC parsing is EDG [7]. 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.

Pinapa: A SystemC Front End A recent release of Pinapa [11], an 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 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 is a very good solution for tackling the SystemC parsing issue. However, Pinapa is a very intrusive approach. It requires 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.

Doxygen A tool that we employ in our approach is Doxygen [5], which is an Open Source documentation project for mainly C/C++ along with extensions for Java and other languages. Though the main purpose of Doxygen is for documenting large C/C++ projects in hyperlinked and organized HTML webpages, a recent addition of an XML representation was introduced. The XML representation captures the classes declared, their private, and public data members as well as member functions. It also transfers the entire source into an XML format. The advantages of an XML representation are obvious in that XML is easily interpreted using XML parsers.

3 SystemCXML Toolchain

In this section, we describe the different phases of the parsing process and the tools that are used. We also present our ASLD format and the structure of the IR. Figure 3 shows the usage flow of SystemCXML. We begin by processing the SystemC source through Doxygen that annotates the model using XML tags. We take the XML output from Doxygen and employ it in our first phase that extracts all the structural information from the SystemC model and generates the ASLD. The ASLD itself is a complete representation of the structural information, including representation of the structural hierarchy. However, for this to be usable, we follow it up with the second phase that constructs a data structure allowing access to this information through an API. We document our approach in detail in the following subsections.

3.1 Parsing SystemC

In order to handle SystemC projects typically consisting of multiple files, we flatten the entire SystemC project into a single file, containing all .cpp and header files. This file is then processed with the Doxygen tool, to generate an XML file. While Doxygen is a tool for automatically generating documentation for projects that analyzes the structure of the source code and beautifies the internal

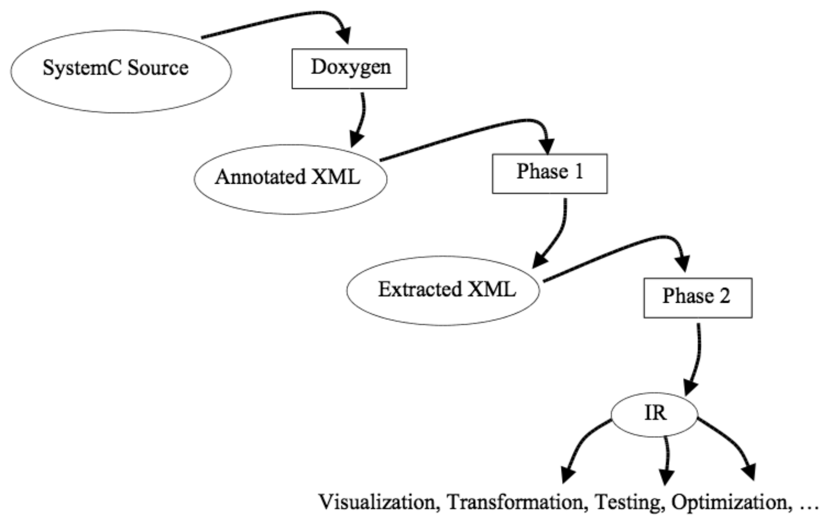


Figure 1. Toolchain of the SystemCXML project

documentation such as comments, there is also an option to include the source code in the output. The source code is then formatted using colors and markup tags for beautification purposes. The standard output for Doxygen is HTML but it also can generate XML code [6, 10]. The tags of the original source code in this XML output can be recognized by any standard XML parsing library. We take advantage of this functionality and therefore break down the problem of parsing plain C++ into parsing XML. Listing 1 shows some raw XML output of Doxygen. We can see that while most of the information is now delimited by tags, there is still some intelligence needed to extract it correctly.

3.2 The Abstract System Level Description

Using Doxygen and the Xerces-C++ XML parser [17], we reflect the following structural properties of the SystemC model: port names, signal names, signal/port types, signal/port sizes, modules, submodules, and entry functions. We extract the sensitivity list of each module and the netlist describing the connections between submodules including the structural hierarchy of the model. We represent this extracted information in an ASLD XML file. Listing 2 shows the

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;<sp/>
<codeline lineno="320" refid="classfir_top_1fir_topr5">
<sp/>sc_out&lt;int&gt;<sp/>
<sp/>RESULT;<codeline lineno="322" refid="classfir_top_1fir_topr6">
<sp/>sc_signal&lt;unsigned&gt;<sp/>
<sp/>state_out;<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>
```

part of an ASLD that corresponds to the Doxygen output shown in Listing 1. During processing, the ASLD is validated against a Document Type Definition (DTD) that defines the legal building blocks of the structural information in a SystemC model. Constraints that the DTD enforces are for example that two ports of a module should have distinct names or that a signal has to carry a type. If the ASLD validates successfully, it represents a valid interpretable SystemC model. The main entities of the ASLD are shown in Listing 3. The ASLD is an abstract representation that is easily extendible and is complete, since it does handle all SystemC constructs that can be used to represent structural properties.

3.3 Building the Data Structure

The second phase is entirely independent from the first pass. The only input it requires is an ASLD conforming to the DTD description. The goal of this phase is to read in the generated ASLD, process the information and store it in an internal structure that is both, easily accessible and does closely represent the structure of SystemC code. As the structure of the IR is the basis for all data manipulations and

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"/>
  <signal type = "unsigned" name = "state_out"/>
  <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"/>
```

Listing 3. Main Entities of the DTD

```
<!ELEMENT model (module)* >
<!ATTLIST model name CDATA #REQUIRED>
<!ELEMENT module (inport | outport | inoutport | signal | submodule)* >
<!ATTLIST module name CDATA #REQUIRED type CDATA #REQUIRED >
<!ELEMENT submodule EMPTY >
<!ATTLIST submodule type CDATA #REQUIRED name CDATA #REQUIRED
                    instancename CDATA #REQUIRED >
<!ELEMENT signal EMPTY >
<!ATTLIST signal type CDATA #REQUIRED bitwidth CDATA #IMPLIED
                 name CDATA #REQUIRED >
<!ELEMENT inport EMPTY >
<!ATTLIST inport type CDATA #REQUIRED bitwidth CDATA #IMPLIED
                 name CDATA #REQUIRED >
<!ELEMENT constructorof (process * | sensitivitylist) >
<!ATTLIST constructorof modulename CDATA #REQUIRED >
<!ELEMENT process EMPTY >
<!ATTLIST process type CDATA #REQUIRED name CDATA #REQUIRED >
<!ELEMENT sensitivitylist (trigger)* >
<!ELEMENT trigger EMPTY >
<!ATTLIST trigger name CDATA #REQUIRED edge CDATA #REQUIRED>
<!ELEMENT connection EMPTY>
<!ATTLIST connection instance CDATA #REQUIRED member CDATA #REQUIRED
          local_signal CDATA #REQUIRED>
```

back-end passes, it is important for it to be generic and to provide necessary accessory functions to traverse the module hierarchy and extract the required structural information. The UML class diagram in Figure 2 gives an overview of all container classes of the data structure and how they are related. There are classes for all constructs that we extract: *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. For example one step is to determine a list of toplevel modules. Toplevel modules are modules that are not used as a submodule

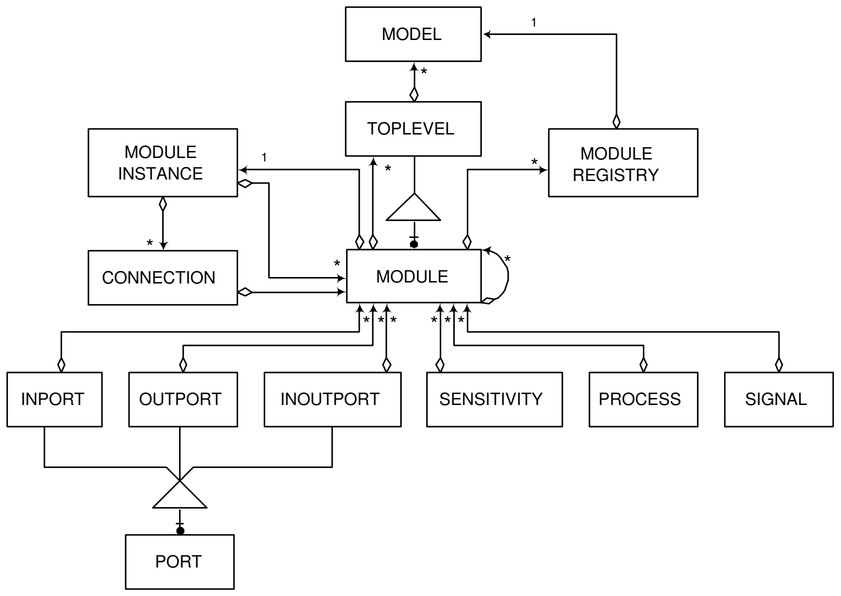


Figure 2. UML class diagram of the internal data structure

in any other module. Typically this is *sc_main* but also testbenches or additional material can be found at the highest level of hierarchy. All processing passes start from these toplevel modules and traverse the entire module hierarchy. Another processing step is making the connections more accessible. In the XML file, we only have connections connect a local signal with a single port of a module instance. During processing, the connections for the same local signal are grouped together such that it is clear which ports of the submodules are connected to each other. The *Connection* class in the IR is designed to hold this information.

In a SystemC program, each module is only defined once but it can be used in many places. In order to reflect this behavior, we add the class *moduleinstance*, which has a unique instance name and holds a pointer to its module structure. This way, when creating multiple instances of a module, the information about ports and submodules is not replicated each time but we rather create another instance that points to the module that is holding this information.

3.4 Improvements

In previous sections, we explained how we used the XML output of the SystemC source code to extract the required information. However, we can improve on that by further helping Doxygen recognize some predefined macros. For example, a commonly used macro in SystemC is the SC_MODULE(arg) macro that represents a class `arg`. For Doxygen to recognize this construct, we provide a macro expansion in the following manner:`SC_MODULE( arg )=class arg: public sc_module`. However, to identify all the SystemC datatypes without including the entire source code, we supply an include file that contains the type definitions. For example, we define `sc_in` in the following way: `template <typename T> class sc_in{};` and Doxygen is able to tag instances of `sc_in` as of type `sc_in` class. Listing 4 shows a segment of the FIR module declaration after configuring Doxygen to process the macro and the supplemental header file. Notice, how the XML tag identifies the data members of the class and also identifies the type such as `sc_in` in Listing 4, Line 8. This makes identification of data members, member functions and their types and arguments much simpler than interpreting from the annotated source code.

4 Usage Example: A Visualization Back End Pass

One possible usage of SystemCXML is graphical visualization. Especially for large projects, it is very intuitive to explore a design visually rather than trying to understand the structure of the project by browsing through the code. There are many different display possibilities that can help to better understand a design, such as the module connection on one or multiple levels, the module hierarchy, a map with blocks whose sizes corresponding to the code size of the modules. For most of these the fact that we do not have information about the behavior is no obstacle at all. As visualization can greatly improve productivity, it should be an integral part of any SLDL tool suite. Design visualization tools are especially helpful for design space exploration and semi-automated design refinements. In

Listing 4. Improved Doxygen output

```
<type>void</type>
<definition>void fir::entry</definition>
<argsstring>()</argsstring><name>entry</name><type>
<ref refid=”classsc__in”kindref=”compound”>sc_in</ref>&lt;bool&gt;</type>
<definition>sc_in&lt;bool&gt; fir::reset</definition>
<name>reset</name><type>
<ref refid=”classsc__in”kindref=”compound”>sc_in</ref>&lt;bool&gt;</type>
<definition>sc_in&lt;bool&gt; fir::input_valid</definition>
<name>input_valid</name><type>
<ref refid=”classsc__in”kindref=”compound”>sc_in</ref>&lt;int&gt;</type>
<definition>sc_in&lt;int&gt; fir::sample</definition>
<name>sample</name><type>
<ref refid=”classsc__out”kindref=”compound”>sc_out</ref>&lt;bool&gt;</type>
<definition>sc_out&lt;bool&gt;fir::output_data_ready</definition>
<name>output_data_ready</name><type>
<ref refid=”classsc__out”kindref=”compound”>sc_out</ref>&lt;int&gt;</type>
<definition>sc_out&lt;int&gt; fir::result</definition>
<name>result</name><type>
<ref refid=”classsc__in__clk”kindref=”compound”>sc_in_clk</ref></type>
<definition>sc_in_clk fir::CLK</definition>
<name>CLK</name><type>
<ref refid=”classsc__int”kindref=”compound”>sc_int</ref>&lt; 9 &gt;</type>
<definition>sc_int&lt;9&gt; fir::coefs[16]</definition>
<argsstring>[16]</argsstring><name>coefs</name>
```

addition to that the automatic generation of graphs and diagrams visualizing the design can be used 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, we implement a back-end pass that generates a graph of the SystemC module hierarchy. Since, there are many free libraries available for graph rendering, we decided to use the DOT format [8] from the graphviz [9] package to render our graphs. It is a comprehensive and easy to use package, which is used in many Open Source projects.

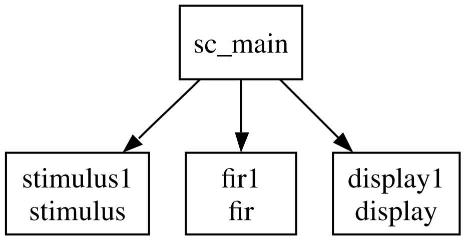
4.1 The DOT Format

Figure 3 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.

There exist many programs to interactively view DOT files or convert them

```
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 3. DOT code and resulting graph for the FIR filter

into various picture formats. Dotty is the standard viewer and part of the Graphviz, but there are better viewers such as [15].

4.2 Graph Generation

To generate the graph, we start at the list of toplevel modules, these are modules that are not a submodule of any other module. Then we 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. However in the SystemC code this is not the case. If for example, we have modules A and B and C, and $B1$ is an instance of B and a submodule of A. Now if $A1$ and $A2$ are submodules of C, we get 2 instances of B that have the name $B1$, namely in $A1$ and $A2$. In order to avoid this we keep track of multiple instantiations of a module and distinguish between the respective submodules.

Figure 4 shows part of the module hierarchy of a USB controller the code of which was obtained from OpenCores [3]. 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. This is due to the fact that we read in the whole SystemC project as one file containing all source and header files. 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.

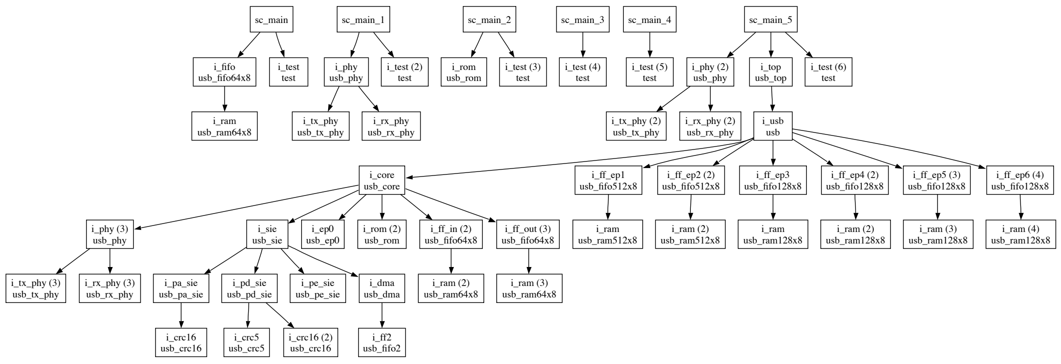


Figure 4. Visualization of the module hierarchy of a USB controller

The implementation of the module hierarchy graph generation back-end pass took only about 60 lines of C++ code. This is small when considering the added value. We assert that given the captured structural information other visualization or transformation back-end passes can be added with comparable effort, lowering the threshold effort to try new things or implement desired functionality. An upgraded version of the visualization may allow a visualization of the module connections, maybe with an adjustable number of displayed hierarchy levels. We were looking into this option as well, but as Graphviz does not natively support this kind of nested hierarchy the use of a different graph rendering library might be necessary to render this kind of graph.

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. There are many applications that can benefit from exploiting structural SystemC information, among them we have looked into automated testbench generation, introspection, and we are currently working on the generation of synchronous component interfaces. In this paper, we illustrate the creation of a back-end pass with the example of a module hierarchy graph generator. Automated design visualization can also used to generate graphs and diagrams system component documentation, lowering obstacles to component sharing and reuse.

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 [13].

References

[1] Albert Benveniste, Paul Le Guernic, and Christian Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Science of Computer Programming*, 16(2):103–149, 1991.

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

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

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

[5] Doxygen Team. Doxygen. http://www.stack.nl/∼dimitri/doxygen/.

[6] Bob DuCharme. *XML: The Annotated Specification.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1998.

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

[8] Emden R. Gansner, E. Koutsofios, S. North, and K.-P. Vo. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering*, 19(3):214–230, March 1993.

[9] 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.

[10] Charles F. Goldfarb and Paul Prescod. *The XML handbook*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1998.

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

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

[13] Omitted for blind review.

[14] The Open SystemC Initiative (OSCI). SystemC Reference Manual. Website: http://www.systemc.org.

[15] Emmanuel Pietriga. Zgrviewer - a 2.5D graph visualizer for the DOT language. http://zvtm.sourceforge.net/zgrviewer.html, 2005.

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

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

[18] D. Thomas and P. Moorby. *The Verilog Hardware Description Language*. Kluwer Academic, 1991.

[19] VHDL. VHDL. Website: http://www.vhdl.org/.