Automated Extraction of Structural Information from SystemC-based IP for Validation

David Berner[⊕] Hiren D. Patel[*] Deepak A. Mathaikutty[*] Sandeep K. Shukla[*]
[⊕]Institut de Recherche en Informatique et Systèmes Aléatoires (IRISA/INRIA)
Campus universitaire de Beaulieu, 35042 Rennes, France
[*]Virginia Polytechnic and State University, FERMAT Lab.,
302 Whittemore Hall, Blacksburg 24061, VA, USA
david.berner@irisa.fr hiren@vt.edu mathaikutty@vt.edu shukla@vt.edu

ABSTRACT

The increasing complexity and size of system level design models introduces a difficult challenge for validating them. Hence, in most industries, design validation takes a large percentage of the overall design time. The immediate solution is to automate certain procedures of generating testbenches from the design given certain information about the model. However, the volatile nature of models used for design exploration results in the designer having to alter the testbenches or the automation for the testbenches to reflect the design changes. In efforts to alleviate this problem of constantly changing designs and generating appropriate testbenches for the changed design, we propose a methodology of using structural reflection to extract structural information from design sources allowing the use of tools such as testbench generators and model viewers to seamlessly employ this extracted information. In this paper we present a methodology to automatically extract structural information from already existing SystemC projects and we show how this information can be exploited for system management and validation tasks. We illustrate example uses such as visualization, design management tasks, and automated test generation.

1. INTRODUCTION

The rising complexity of embedded system design and a widening of the productivity gap have raised the importance of System Level Design (SLD)s languages and frameworks. In recent years, we have seen SLDs such as SpecC and SystemC [7, 12] in efforts to raise the level of abstraction in hardware description languages. These SLDs assist designers in modeling, simulation, validation and verification of complex designs. However, the high complexity and heterogeneity of designs make it difficult for embedded system designers to meet the time-to-market. Designers require improved methodologies for verification and validation and tools for debugging and visualization for easier model building to mitigate this productivity crisis. The growing size of common system models is forcing design houses to reuse intellectual property from other designs or from third party companies. The fact that all the different parts of the design have not been conceived in one toolset and by the one actually using it is becoming an additional challenge.

Aside from writing regular testbenches as test-driver modules in SystemC, the SCV library [12] is a good medium of writing different types of testbenches allowing features such as randomized testbneches. Unfortunately, the designer must have an understanding of the design, interconnections, datatypes, etc. to generate a testbench using SCV. Furthermore, as the design undergoes changes, the testbench in SCV must be altered. Automating this process of altering the testbench for a design requires access to the the structural design information, but most SLD languages and frameworks do not provide a clean mechanism for querying such information. This is why we think it is important that tools are able to automatically extract and exploit structural design information from existing SLD models in order to further facilitate a realm of design tasks for easier model management, model visualization, automated test generation, improved debugging, etc.

Our approach to extracting structural design information uses a suite of open-source technologies consisting of Doxygen [4], Apache's Xerces-C++ XML [15], in combination with a C++ library to enable validation tasks exploiting this information. We also do not require any interface description language for entering meta-data. Our approach is based on pre-processing SystemC models through our tools. To show the benefits of having easy access to structural design information, we implement several clients that use it. These clients serve only as examples of exploiting this kind

Submitted to IEEE International Workshop on Microprocessor Test and Verification (MTV) 2005

of information and using it for validation purposes. One example is a visualization backend that generates graphical views of the structural information, another is an automated test generator.

In this paper, we provide details on our approach to the extraction of structural information from existing SystemC designs and describe clients that exploit this information for design validation purposes. We show the benefits and importance of having access to structural design data for the validation of system level designs.

1.1 Organization

In Section 2 we discuss some related work, along with the technologies we employ. We discuss the main contributions of this work in Section 3. Section 4 then describes how the structural information is extracted, Section 5 describes how this information can be used for different validation aspects and we finally give some concluding remarks and future work in Section 6.

2. RELATED WORK

In this section we describe some related frameworks and languages as well as the open-source tools that we employ for our implementation of the extraction and exploitation of structural information.

2.1 Existing Tools for Structural Reflection

Several tools may be used for implementing structural reflection in SystemC. Some of them are SystemPerl [14], EDG [6], or C++ as in the BALBOA framework [3]. However, each of these approaches have their own drawbacks. For instance, SystemPerl requires the user to add certain hints into the source file and although it yields all SystemC structural information, it does not handle all C++ constructs. EDG is a commercial front-end parser for C/C++ that parses C/C++ into a data structure, which can then be used to interpret SystemC constructs. However, interpretation of SystemC constructs is a complex and time consuming task, plus EDG is not to be freely used in public domain. BALBOA implements its own reflection mechanism in C++ which again only handles a small subset of the SystemC language. As for runtime reflection, to our knowledge, there is no framework that exposes runtime characteristics of SystemC models.

2.2 BALBOA Framework

The BALBOA [3] framework describes a framework for component composition, but in order to accomplish that, they required R-I capability of their components. They also discuss some introspection mechanisms and whether it is better to implement R-I at a meta-layer or within the language itself. We limit our discussion to only the approach used to provide R-I in BALBOA.

BALBOA uses their BIDL (BALBOA interface description language) to describe components, very similar to CORBA IDLs [11]. Originally IDLs provide the system with type information, but BALBOA extends this further by providing structural information about the components such as ports, port sizes, number of processes, etc. This information is stored at a meta-layer (a data structure representing the reflected characteristics). BALBOA forces system designers to enter meta-data through BIDL which is inconvenient. Our method only needs pre-processing of SystemC models.

A drawback of this framework is that the BIDL had to be implemented. Furthermore, the designer writes the BIDL for specifying the reflected structure information which can be retrieved automatically from SystemC source. Furthermore, runtime reflection was not done in BALBOA.

2.3 Java, C# .NET Framework, C++ RTTI

Here, we discuss some existing languages and frameworks that use the R-I capabilities. They are Java, C# and the .NET framework and C++ RTTI. Java's reflection package `java.lang.reflect` and .NET's reflection library `System.Reflection` are excellent examples of existing R-I concept implementations. Both of these supply the programmer with similar features such as the type of an object, member functions and data members of the class. They also follow a similar technique in providing R-I, so we take the C# language with .NET framework as an example and discuss in brief their approach. C#'s compiler stores class characteristics such as attributes during compilation as meta-data. A data structure reads the meta-data information and allows queries through the `System.Reflection` library. In this R-I infrastructure, the compiler performs the reflection and the data structure provides mechanisms for introspection.

C++'s runtime type identification (RTTI) is a mechanism for retrieving object types during execution of the program. Some of the RTTI facilities could be used to implement R-I, but RTTI in general is limited in that it is difficult to extract all necessary structural SystemC information by simply using RTTI. Furthermore, RTTI requires adding RTTI-specific code within either the model, or the SystemC source and RTTI is known to significantly degrade performance.

2.4 Doxygen, XML, Apache's Xerces-C++

Two main technologies we employ in our solution for obtaining and exploiting structural information from SystemC models are Doxygen and XML. Doxygen [4] is a documentation system primarily for C/C++, but has extensions for other languages. Since SystemC is simply a library of C++ classes, it is ideal to use Doxygen's parsing of C/C++ structures and constructs to generate XML representations of the model. In essence Doxygen does most of the difficult work in tagging constructs and also documenting the source code in a well-formed XML. By using XML parsers from Apache's Xerces-C++ we can parse the Doxygen XML output files and obtain any information about the original C / C++ /

SystemC source. In [1] we describe the front end tool using these techniques that we call SystemCXML. The article [5] describes a service oriented architecture using that.

3. MAIN CONTRIBUTIONS

Our main contributions in this paper are to show how extracting structural information from IPs can open a path for a plethora of validation applications and we demonstrate this for SystemC examples. The main contributions are:

- Indicate and demonstrate how structural information is a key for design validation and management applications such as visualization, automated test generation, browsing, and re-packaging.
- Show how such structural information can be relatively easily obtained from already existing SystemC models with the use of open source tools such as, Doxygen, XML, and Xerces-C++.

4. EXTRACTING STRUCTURAL INFORMATION

Here, we present details on the infrastructure for the automated extraction of structural information. We only provide small code snippets to present our approach and the concept of using Doxygen, XML, Xerces-C++, and a C++ data structure to perform the extraction and provide the information to use it for validation purposes. For more dtails on the inner workings of the tool please refer to [1].

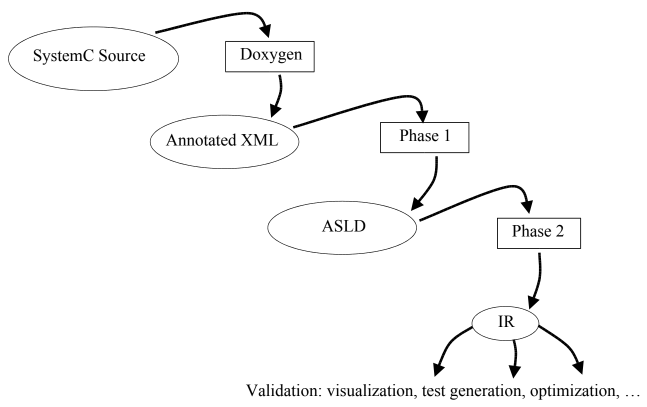


Figure 1: Design Flow for the extraction

Doxygen pre-processing: Using Doxygen has the immediate benefit of C/C++ parsing and its corresponding XML representations. However, Doxygen requires declaration of all classes for them to be recognized. Since all SystemC constructs are either, global functions, classes or macros, it is necessary to direct Doxygen to their declarations. For example, when Doxygen executes on just the SystemC model then declarations such as `sc_in` are not tagged, since it has no knowledge of the class `sc_in`. The immediate alternative is to process the entire SystemC source along with the model, but this is very inconvenient when only interested in reflecting characteristics of the SystemC model. However, Doxygen does not perform complete C/C++ compilation and grammar check and thus, it can potentially document incorrect C/C++ programs. We leverage this, by indicating which particular classes need to be tagged, by simply adding the class definition in a file that is included during processing. There are only a limited number of classes that are of interest and they can easily be declared such that Doxygen recognizes them. As an example we describe how we force Doxygen to tag the `sc_in`, `sc_out`, `sc_int` and `sc_uint` declarations. We include this description file everytime we perform our pre-processing such that Doxygen recognizes the declared ports and datatypes as classes. A segment of the file is shown in Figure 2, which shows declaration for input and output ports along with SystemC integer and SystemC unsigned integer datatypes.

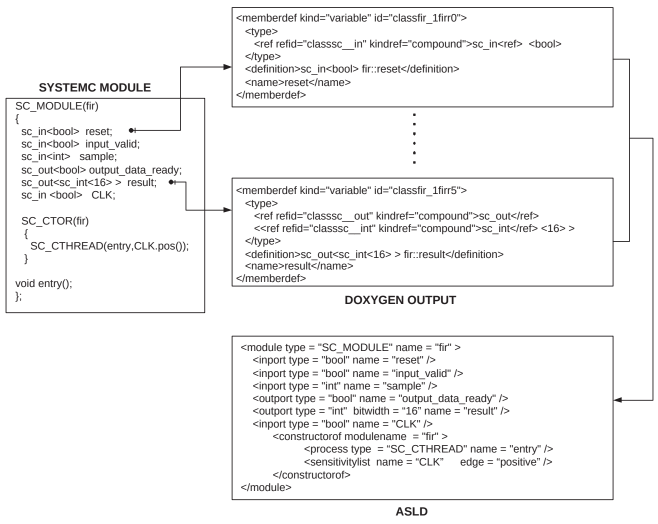
```
/*! SystemC port classes !*/
template<class T> class sc_in { };
template<class T> class sc_out { };

/*! SystemC datatype classes !*/
template<class T> class sc_int {  };
template<class T> class sc_uint { };
```

Figure 2: Examples of class declarations

The resulting XML for a small code example is shown in Figure 3. Doxygen itself also has some limitations though and it cannot completely tag all the constructs of SystemC without explicitly altering the source code, which we avoid doing. For example, the `SC_MODULE(arg)` macro defines a class specified by the argument `arg`. Since we do not include all SystemC files in the processing, Doxygen does not recognize this macro when we want it to recognize it as a class declaration for class `arg`. However, Doxygen allows for macro expansions during pre-processing. Hence, we insert a pre-processor macro as: `SC_MODULE(arg)=class arg: public sc_module` that allows Doxygen to recognize `arg` as a class derived from class `sc_module`. We define the pre-processor macro expansions in the Doxygen configuration file where the user indicates which files describe the SystemC model, where the XML output should be saved, what macros need to be run, etc. We provide a configuration file with the pre-processor macros defined such that the user only has to point to the directory with the SystemC model. More information regarding the Doxygen configuration is available at [4].

Even through macro preprocessing and class declarations, some SystemC constructs are not recognized without the original SystemC source code. However, the well-formed XML output allows us to use XML



```
SC_MODULE(fir)
{
  sc_in<bool>  reset;
  sc_in<bool>  input_valid;
  sc_in<int>   sample;
  sc_out<bool> output_data_ready;
  sc_out<sc_int<16> >  result;
  sc_in <bool>   CLK;

  SC_CTOR(fir)
   {
    SC_CTHREAD(entry,CLK.pos());
   }

void entry();
};
```

```
<memberdef kind="variable" id="classfir_1firr0">
  <type>
    <ref refid="classsc__in" kindref="compound">sc_in<ref>  <bool>
  </type>
  <definition>sc_in<bool> fir::reset</definition>
  <name>reset</name>
</memberdef>
```

```
<memberdef kind="variable" id="classfir_1firr5">
  <type>
    <ref refid="classsc__out" kindref="compound">sc_out</ref>
    <<ref refid="classsc__int" kindref="compound">sc_int</ref> <16> >
  </type>
  <definition>sc_out<sc_int<16> > fir::result</definition>
  <name>result</name>
</memberdef>
```

```
<module type = "SC_MODULE" name = "fir" >
   <inport type = "bool" name = "reset" />
   <inport type = "bool" name = "input_valid" />
   <inport type = "int" name = "sample" />
   <outport type = "bool" name = "output_data_ready" />
   <outport type = "int"  bitwidth = “16” name = "result" />
   <inport type = "bool" name = "CLK" />
        <constructorof modulename  = "fir" >
                <process type  = “SC_CTHREAD" name = "entry" />
                <sensitivitylist  name = “CLK”     edge = “positive” />
        </constructorof>
</module>
```

Figure 3: Doxygen XML Representation

parsers to extract the untagged information. We employ Xerces-C++ XML parsers to parse the Doxygen XML output, but we do not present the source code here as it is simply a programming exercise, and point the readers at [10] for the source code.

XML Parsers: Using Doxygen and XML parsers we reflect the following structural characteristics of the SystemC model: port names, types and widths, signal names, types and widths, module names and processes in modules and their entry functions. We reflect the sensitivity list of each module and we also reflect the netlist describing the connections including structural hierarchy of the model. We represent this reflected information in an Abstract System Level Description (ASLD) XML file. The ASLD validates against a Document Type Definition (DTD) which defines the legal building blocks of the ASLD that represents the structural information of a SystemC model. For example, some constraints that the DTD enforces are that two ports of module should have distinct names or all modules within a model should be unique, which verifies that the ASLD correctly represents an executable SystemC model. The main entities of the ASLD are shown in Listing 1.

ASLD: In Listing 1, the topmost *model* element corresponds to a SystemC model with multiple modules. Each *module* element acts as a container for the following: input ports, output ports, inout ports, signals and submodules. Each *submodule* in a *module* element is the instantiation of a module within another module. This way the ASLD embeds the structural hierarchy in the SystemC model and allows the introspective architecture to infer the toplevel module. The *submodule* is defined similar to a *module* with an additional attribute that is the instance name of the submodule. The *signal* element with its name, type and bitwidth attributes represents a signal in a module. Preserving hierarchy information is very important for correct structural representation. The element *inport* represents an input port for a module with respect to its type, bit width and name. Entities *outport* and *inoutport* represent the output and input-output port of a module. Line 16 describes the *constructorof* element which contain multiple process elements and keeps a *sensitivitylist* element. The *process* element defines the entry function of a module by identifying whether it is an sc_method, sc_thread or sc_cthread. The *sensitivitylist* element registers each signal or port and the edge that a module is sensitive to as a *trigger* element. Connections between submodules can be found either in a module or in the *sc_main. Each* connection element holds the name of the local signal, the name of the connected instance and the connected port within that instance. This is similar to how the information is present in the SystemC source code and is sufficient to infer the netlist for the internal data structure.

Using our well-defined ASLD, any SystemC model can be translated into an XML based representation and furthermore models designed in other HDLs such as VHDL or Verilog can be translated to represent synonymous SystemC models by mapping them to the ASLD. This offers the advantage that given a translation scheme from say a Verilog design to the ASLD, we can introspect information about the Verilog model as well.

Listing 1: Main Entities of the DTD

```
 1<!ELEMENT model (module)* >
 2<!ATTLIST model name CDATA #REQUIRED>
 3<!ELEMENT module (inport | outport |
      inoutport | signal | submodule)* >
 4<!ATTLIST module name CDATA #REQUIRED type
      CDATA #REQUIRED >
 5<!ELEMENT submodule EMPTY >
 6<!ATTLIST submodule type CDATA #REQUIRED
      name CDATA #REQUIRED instancename CDATA
       #REQUIRED >
 7<!ELEMENT signal EMPTY >
 8<!ATTLIST signal type CDATA #REQUIRED
      bitwidth CDATA #IMPLIED name CDATA #
      REQUIRED >
 9<!ELEMENT inport EMPTY >
10<!ATTLIST inport type CDATA #REQUIRED
      bitwidth CDATA #IMPLIED name CDATA #
      REQUIRED >
11<!ELEMENT constructorof (process * |
      sensitivitylist) >
12<!ATTLIST constructorof modulename CDATA #
      REQUIRED >
13<!ELEMENT process EMPTY >
14<!ATTLIST process type CDATA #REQUIRED name
       CDATA #REQUIRED >
15<!ELEMENT sensitivitylist (trigger)* >
16<!ELEMENT trigger EMPTY >
17<!ATTLIST trigger name CDATA #REQUIRED edge
       CDATA #REQUIRED>
18<!ELEMENT connection EMPTY>
19<!ATTLIST connection instance CDATA #
      REQUIRED member CDATA #REQUIRED
      local_signal CDATA #REQUIRED>
```

Data structure: The ASLD file serves as an information base for our reflection capabilities. We create an internal data structure that reads in this information, enhances it and makes it easily accessible. The class diagram in Figure 4 gives an overview of the data structure. The *topmodule* represents the toplevel module from where we can navigate through the whole application. It holds a list of module instances and a list of connections. Each connection has one read port and one or more write ports. The whole data structure is modeled quite close to the actual structure of SystemC source code. All information about ports and signals and connections are in the module structure and only replicated once. Each time a module is instantiated, a *moduleinstance* is created that holds a pointer to its corresponding module.

The information present in the ASLD and the data structure does not contain any behavioral details about the SystemC model at this time, it merely gives a control perspective of the system. It makes any control flow analysis and optimizations on the underlying SystemC very accessible.

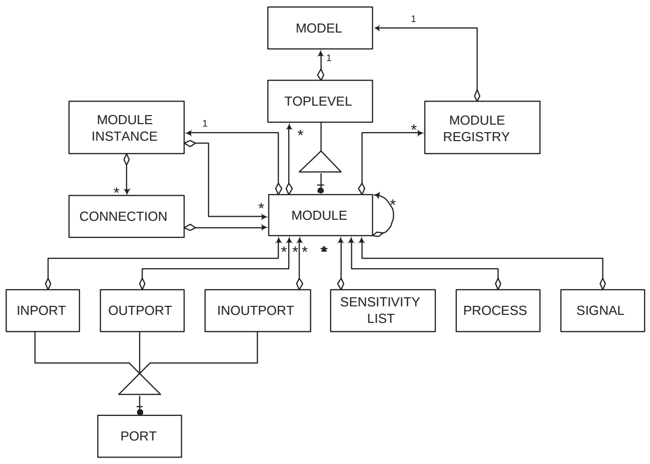


Figure 4: Class diagram showing data structure

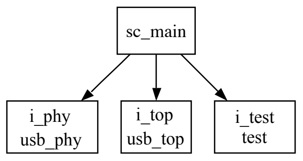
5. APPLICATIONS FOR VALIDATION

5.1 Visualization

One possible usage of SystemCXML is to facilitate graphical visualization. For large models especially, it is very intuitive to explore a design visually rather than trying to infer the structural aspects of the model by browsing through the code. This problem becomes even more difficult, if the model description is disturbed over multiple files. Therefore having any form of visualization for the project does ease exploration and debugging capabilities. There are visualization modes at different levels of abstraction that can help to better comprehend a design, such as the netlists displaying module connections on one or multiple levels, the module hierarchy, a layout with blocks whose sizes are mapped to the code size of the corresponding modules. The ability to provide the above visualization modes can be easily achieved through an extraction of the structural information of the model, therefore it is not necessary

```
digraph usb
{
  node[shape=box];
  ratio=fill;
  sc_main;
  sc_main->"i_phy\usb_phy";
  sc_main->"i_top\usb_top";
  sc_main->"i_test\test";
}
```

(a) DOT code



(b) Resulting graph

Figure 5: Example DOT code and resulting graph

to understand the behavioral aspects. 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 addition to the above advantages, the automatic generation of a graphical visualization of the design can also be used for documenting different IPs, a step often neglected, leading to better collaboration in terms of IP exchange between different vendors and enhancing reusability of components.

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.

5.1.1 The DOT Format

Figure 5 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 into various picture formats. Dotty is the standard viewer and part of the Graphviz, but there are better viewers such as [13].

5.1.2 Graph Generation

To generate the graph, we start with 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 necessarily the case. For example, we have three modules A, B and C, with $B1$ being 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 6 shows a part of the module hierarchy of the SystemC implementation of a USB controller from OpenCores [2]. In the lower right hand corner you can see four instances of $usb_fifo128$x8, 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, which is the case in this example. The visualization of the hierarchy helps to see the different parts of the design and understand their usage.

The code for the back-end pass to generate the graphical visualization illustrating the module hierarchy, took around 60 lines of C++ code. This is small when considering the value add. We assert that given the captured structural information, multiple back-end passes for other visualization modes or transformations can be added with comparable effort, reducing the effort needed in trying different things or implementing a desired functionality. An enhancement to the current version of the our visualization mode would allow the module connections where the user can choose the number of displayed hierarchy levels. We were looking into this option as well, but Graphviz does not natively support this kind of nested hierarchy, so it may be necessary to use of a different graph rendering library.

5.2 Design Management

Another important possible use of structural design information is design management. Large designs are getting difficult to maintain - even if they are kept in a file and folder structure that is readily accessible. Browsing a design graphically or through tree and list views can help in managing and maintaining large designs. This makes it easy to view the different components of a design and furthermore it helps in identifying the component of concern, which reduces the effort needed in isolating a particular component of interest to the user.

Suppose the user is interested in knowing the amount of RAM attached to the overall design. This is not obvious to infer from the code since a single RAM module can be instantiated with different sizes at multiple places within the same module as well as across modules. Furthermore, there may be possible RAM instantiations in the design that were done for testing purposes, which is not a part of the actual design. For these cases, the designer can use the module hierarchy visualization, to view all the instances down to the leaf level, from which it is easier to filter out all the RAM instantiations and sum them up to get an idea of how much memory is been used and how much space is needed on the chip. Figure 6 illustrates the module structure of a USB controller, it contains two 64k blocks, four 128k blocks, and two 512k blocks, which sums up to 1664 kilobytes RAM. This calculation does not include the 64 kilobytes instantiated in the test of usb_fifo. This valuable information can be easily achieved using our module hierarchy view, which is otherwise very tedious.

Another design management task can be to identify certain functionalities in old designs and package them into a new IP for future use. In a netlist view the graphical selection of a set of modules can then be put into a new module with an automatically generated interface, containing all these modules and their dependent submodules. Again this kind of operation only necessitates knowing the structural information, but otherwise tedious to perform.

5.3 Automated Test Generation

We develop a Testbench Generator client that uses the structural information to support automated test generation. The test generation client is built using the SystemC verification (SCV) [12], which is a library of C++ classes, that provide tightly integrated verification capabilities within SystemC. The Testbench Generator interacts with the ASLD by invoking the respective API calls to access the structural information pertaining to test generation. The generator takes as input a SystemC model and generated automatically a SystemC test for the selected part of the model. This Testbench Generator uses structural information such as the type, bitwidth of ports and signals to generate test vectors appropiate for this specification of the SystemC model. The generator also has the abilities to generate tests for pre-specified ports or signals of a SystemC model. The client generates different tests based on the mode in which it is set. The different mode can be set during initialization of the client. The *unconstRand*, *simpleRand* and *distRand* are the currently defined modes.

The test generator can create constrained and unconstrained randomized testbenchs. In the *unconstRand* mode, the client generates unconstrained randomized tests using objects of the *scv_smart_ptr<T>* class of SCV, which are containers for objects of type T. In the *simpleRand* mode constrained randomized testbenchs are created. These tests issue *Keep_out* and *Keep_only* commands to define the legal range of values given in the data file. Similarly in the *distRand* mode, SCV bag objects are used in test environments providing which takes a data file as input with the values and their probability.

Furthermore, the Test Generator uses the SystemC constructs to generate trace file in the format of Value Change Dump (VCD). This provides the user with a trace file with value changes on all the reflected variables, ports and signals of the model, which can used for debugging purposes. To generate a trace file, the

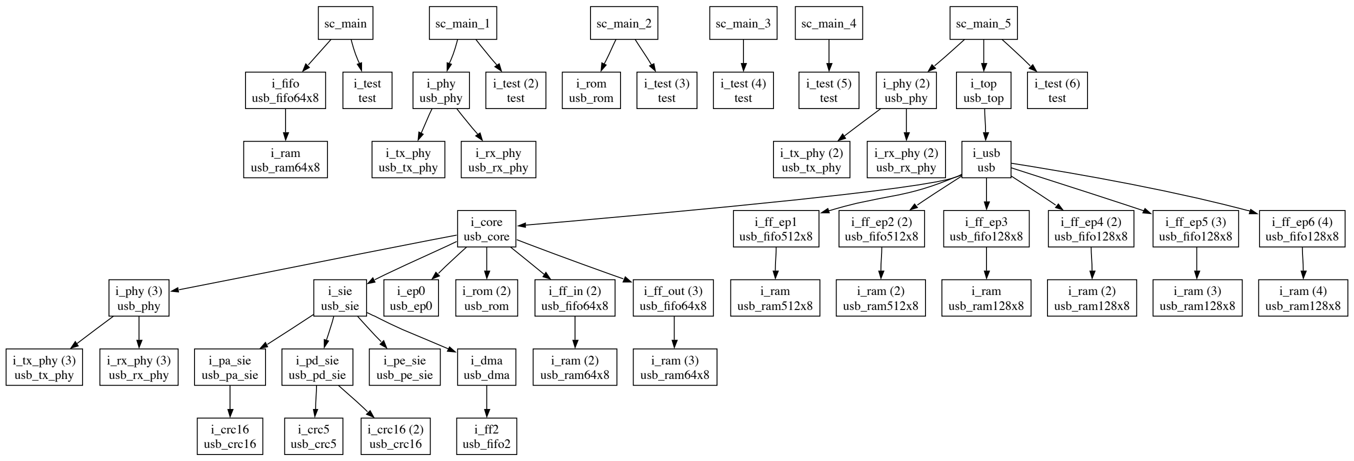


Figure 6: Visualization of the extracted module hierarchy of a USB controller

Test Generator creates a trace file, registers the reflected variables, ports and signals to be traced and closes the trace file.

5.3.1 Test Generation Example

We briefly describe the automatic test generation using a small part of the USB example in Figure 6. We take a look in particular at the module *usb_crc5*, a simple CRC checksum checker. Figure 7 shows the interface of the CRC checker module. It has to input variables, *din* for the data value, and *crc_in*, for the CRC value. For this very simple example we show how to automatically generate test vectors for these inputs.

```
SC_MODULE(usb_crc5) {
  public:
sc_in<sc_uint<5> >crc_in;
sc_in<sc_uint<11> >din;
sc_out<sc_uint<5> >crc_out;
void update(void);
SC_CTOR(usb_crc5) {
SC_METHOD(update);
sensitive << crc_in << din;
}
};
```

Figure 7: Interface of the module usb_crc5 from the USB example

In the *simpleRand* mode, constrained testbenchs are created by initializing SCV smart pointers for the pre-specified port as shown above. Furthermore the random values generated are constrained by defining *keep_out* and *keep_only* constructs with the legal ranges given from the input data file as shown in Figure 9. If no data file is provided then by default random legal ranges are defined.

In the *distRand* mode, constrained testbenchs are generated by defining *SCV_bags* that are given legal ranges and the probabilistic distribution of these ranges

```
/*! Defining SCV smart pointers !*/
 scv_smart_ptr <int> r_crc_in;
 scv_smart_ptr <int> r_din;

/*! Generating the randomized values !*/
 r_crc_in->next();
 r_din->next();
```

Figure 8: Snippet of the testbench in the *unconstRand* mode for a pre-specified port

```
/*! Defining simple constraints !*/
 scv_smart_ptr <int> r_crc_in;
 scv_smart_ptr <int> r_din;

/*! Defining the legal ranges !*/
r_crc_in->keep_only(10,1000);
r_crc_in->keep_out(100, 300);
r_crc_in->keep_out(600, 900);
r_din->keep_only(1,10000000);
r_din->keep_out(1000, 30000000);
r_din->keep_out(3001000, 9000000);
r_din->keep_out(9001000, 10000000);
```

Figure 9: Snippet of the testbench in the *simpleRand* mode for a pre-specified port

from an input data file as shown in Figure 10. As in the *simpleRand* mode, if an input data file is not given then a default distribution and its probability is provided.

During initialization, if the ports are not specified then the test generating client generates tests with respect to all the ports of the given model in focus.

The code snippet that creates a trace file for the usb_crc5 module with the reflected ports crc_in, din and crc_out is shown in Figure 11:

We intend to improve our automated testbench generation capabilities by first implementing additional clients such as coverage monitors and simulation perfor-

```
/*! Defining weights for the distribution mode !*/
 scv_smart_ptr <int> r_crc_in;
 scv_bag<pair<int,int> > d_crc_in;

/*! Defining the legal ranges !*/
 d_crc_in.add(pair<int, int> (1, 100), 40 );
 d_crc_in.add(pair<int, int> (5000100, 500700), 30 );
 d_crc_in.add(pair<int, int> (8000600, 800900), 60 );

/*! Setting the distribution mode !*/
 r_crc_in->set_mode(d_crc_in);
```

Figure 10: Snippet of the testbench in the *distRand* mode for a pre-specified port

```
/*! Step 1: Creating a trace file !*/
    sc_trace_file* tf = sc_create_vcd_trace_File("trace");

/*! module!*/
    usb_crc5 crc_inst("crc_inst");

/*! Step 2: Register signals and variables to be traced !*/
    sc_trace(tf, crc_inst.crc_in, "crc_input");
    sc_trace(tf, crc_inst.din, "din");
    sc_trace(tf, crc_inst.crc_out, "crc_output");

/*! Step 3: Close the trace file !*/
    sc_close_vcd_trace_file(tf);
```

Figure 11: Snippet of the testbench showing the trace file creation

mance monitors to better analyze the SystemC model. These additional clients assist the Testbench Generator in making more intelligent and concentrated testbenchs.

6. CONCLUSION

Large scale system designs and increasing component reuse from diverse sources makes design validation a nightmare. Consistent structural information from the entire design models is difficult to obtain, and we claim that it would be sufficient for many validation tasks.

We present a methodology for automated extraction of structural information from already existing SystemC projects and illustrate how the data can be easily exploited with the examples of a visualization backend pass, design management tasks, and an automated test generator.

The entire system has been implemented using open source tools such as Doxygen, Xerces-C, and GraphViz and source code of the tool itself is published as an open source project at Sourceforge.net [10] for others to study and use.

7. REFERENCES

[1] David Berner, Hiren D.Patel, Deepak A. Mathaikutty, and Sandeep Shukla. SYstemCXML: An Extensible SystemC Front End Using XML. In *To be published in Proceedings of the Forum on specification and design languages (FDL).*, Lausanne, Switzerland, September 2005.

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

[3] F. Doucet, S. Shukla, and R. Gupta. Introspection in System-Level Language Frameworks: Meta-level vs. Integrated. In *Design and Test Automation in Europe*, 2003.

[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 of Integrated Circuits and Systems (TCAD).*

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

[7] D. D. Gajski, J. Zhu, R. Domer, A. Gerstlauer, and S. Zhao. *SpecC: Specification Language and Methodology.* Kluwer Academic Publishers, January 2000.

[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] D. A. Mathaikutty, D. Berner, H. D. Patel, and S. K. Shukla. FERMAT's SystemC Parser. http://systemcxml.sourceforge.net, 2004.

[11] OMG. OMG CORBA. http://www.corba.org/.

[12] OSCI. SystemC and SystemC Verification. Website: http://www.systemc.org.

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

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

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