F E R M A T

Formal Engineering Research using Methods, Abstractions and Transformations

Technical Report No: 2004-22

Abstract -- Reflection is an increasingly important feature in software systems as evidenced by introduction of datatype reflection abilities in Java, RTTI runtime type information reflection package for C++, and reflection service in .NET environment. Since systems being designed today are supposed to self-configure, self-heal and make a lot more intelligent decisions about itself, including versioning, fault-repair etc., reflection, and introspection are important. System Level Design (SLDs) languages such as SystemC are used for modeling software/hardware systems, validating such models, measuring performance, visualizing various aspects for debugging etc. Therefore, introspection and/or reflection capabilities for SystemC and other SLDs can be of great use, for automated reconfiguration for fast design alternatives exploration, automated test generation and coverage analysis, for visualization of signal activities, hot-spots analysis and so on. In this paper, we describe in detail how certain public domain XML based tools may be used to provide structural reflection for SystemC, and how to enhance this reflection into a service, which can also interact with the SystemC models under simulation to reflect useful runtime information as well. Ability for SystemC to query such reflection renders introspection feature to SystemC. We show through some application examples the utility of SystemC reflection and introspection.

Introspective-SystemC: Reflection and Introspection in System Level Design

Hiren D. Patel, Deepak A. Mathaikutty, David Berner and **Sandeep K. Shukla**

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

VirginiaTech
VIRGINIA POLYTECHNIC INSTITUTE AND STATE UNIVERSITY

Contents

List of Figures

Introspective-SystemC: Reflection and Introspection in System Level Design

Hiren Patel, Deepak Mathaikutty, David Berner and Sandeep Shukla

{**hiren,mathaikutty,shukla**}**@vt.edu**

Center for Embedded Systems in Critical Applications (CESCA),

Virginia Polytechnic Institute and State University, Blacksburg.

{**david.berner@irisa.fr**}

Inria - Irisa Campus de Beaulieu 35042 Rennes, France.

January 18, 2005

Abstract

Reflection is an increasingly important feature in software systems as evidenced by introduction of datatype reflection abilities in Java, RTTI runtime type information reflection package for C++, and reflection service in .NET environment. Since systems being designed today are supposed to self-configure, self-heal and make a lot more intelligent decisions about itself, including versioning, fault-repair etc., reflection, and introspection are important. System Level Design (SLDs) languages such as SystemC are used for modeling software/hardware systems, validating such models, measuring performance, visualizing various aspects for debugging etc. Therefore, introspection and/or reflection capabilities for SystemC and other SLDs can be of great use, for automated reconfiguration for fast design alternatives exploration, automated test generation and coverage analysis, for visualization of signal activities, hot-spots analysis and so on. In this paper, we describe in detail how certain public domain XML based tools may be used to provide structural reflection for SystemC, and how to enhance this reflection into a service, which can also interact with the SystemC models under simulation to reflect useful runtime information as well. Ability for SystemC to query such reflection renders introspection feature to SystemC. We show through some application examples the utility of SystemC reflection and introspection.

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 SystemC, SpecC [8, 10] 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. We propose the addition of reflection and introspection capabilities in SLDs to further facilitate a realm of tools and methodologies for easier model building, model visualization, model execution analysis, automated test generation, improved debugging, etc.

Though the idea of reflection and introspection (R-I) is not common in SLDs as it is in some programming languages such as Java, C++ RTTI [11] and C# or other variants using the .NET framework, we show that it is a very useful one for SLDs. Previously, the BALBOA framework [2] took advantage of the R-I concept to facilitate IP reuse and component integration for SystemC models. We also select SystemC as our SLD to illustrate the concepts, because SystemC emerged as an open-source hardware description language providing free modeling and simulation libraries and there is significant industrial momentum in making SystemC successful. The open-source nature allows users to experiment, alter and enhance existing SystemC. Unfortunately, due to the unmanaged C++ implementation of SystemC and rudimentary reflection capability of RTTI, it is difficult to use R-I functionalities without implementing additional libraries and infrastructures to support R-I for SystemC.

Unlike other technologies used in frameworks that require reflection in SystemC such as BALBOA [2] and ESys.NET [5]; our approach to reflection uses a suit of open-source technologies consisting of Doxygen [3], Apache's Xerces-C++ XML [13], in combination with a C++ library for introspection to complete a SystemC **Reflection Provider** (RP). 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. A Reflection Provider provides R-I capabilities for SystemC. To show the benefits of having R-I in SLDs we implement two clients that use the RP. These two clients serve only as examples of using the RP, and they are the Testbench Generator and the dynamic-Value Change Dump (d-

VCD) clients. In addition to a unique solution for R-I, we provide structural and runtime reflection as two subclasses of reflection. Most reflection tools for SystemC such as [2] and [9] only expose structural information of the model, such as the number of ports, their types, bitwdiths, and netlist. Our approach not only allows structural reflection, but it also allows the clients to obtain runtime information of the model such as the d-VCD, number of invocations of a particular entry function, processes on a runlist and so on.

In this paper, we provide details on our approach to R-I in SystemC and describe two clients that use the RP to assist in better tool development. We show the benefits and importance of having R-I capabilities in SLDs via this paper.

1.1 Organization

In Section 3 we discuss related work with R-I along with the technologies we employ in endowing SystemC with R-I. We discuss the main contributions of this work in Section 4. Section 5 then describes the clients that use the RP followed by an example and finally concluding remarks and future work in Section 6.

2 Background & Related Work

In this section we define reflection and introspection followed by descriptions of some frameworks and languages that provide R-I along with the open-source tools that we employ in deriving our solution for R-I.

2.1 Reflection and Introspection

Introspection is the ability of an executable system to query internal descriptions of itself through some *reflective* mechanism. The reflection mechanism exposes the structural and runtime characteristics of the system and stores it in a data structure. We call data stored in this data structure *meta-data* in this paper. This data structure is used to query the requested internal characteristics. The two subcategories of the reflection meta-data are structural and runtime. Structural reflection refers to descriptions of the structure of a system. For SystemC, structural reflection implies module name, port types and names, signal types and names, bitwidths, netlist and hierarchy information. On the other hand runtime reflection exposes information such as the number of invocations of a particular process, the number of events generated for a particular module and so on. An infrastructure

that provides for R-I (either structural or runtime reflection) is what we term an RP.

2.2 Existing tools for structural reflection

Several tools may be used for implementing structural reflection in SystemC. Some of them are SystemPerl [9], EDG [4], or C++ as in the BALBOA framework [2] and ESys.NET [5]. 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 and ESys.NET implement their own reflection mechanism in C++ which again only handle 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.

2.3 ESys.NET Framework

ESys.NET [5] uses an interesting idea where they implement a composite design pattern for datatypes into the original source code. Their work is inspired by the .NET framework's reflection mechanism. ESys.NET enhances SystemC's datatype library by implementing the design pattern with additional C++ classes. This altered datatype library introduces member functions that provide introspection capabilities for the particular datatypes. However, this requires altering the datatype library and altering the original source code to extract structural information.

2.4 BALBOA Framework

The BALBOA [2] 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 [7]. Originally IDLs provide the system with type information, but BALBOA extends this further by providing structural information about the component 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.

2.5 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.

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

Two main technologies we employ in our solution for R-I for SystemC are Doxygen and XML. Doxygen [3] 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.

3 Main Contributions

Our main contributions in this paper are to show the importance and possibilities from implementing a R-I architecture in SLDs, which we illustrate through SystemC. The main contributions are:

- Show the use of public domain tools, Doxygen, XML and Xerces-C++ for structural reflection of SystemC models, thus avoiding using front-end parsing tools such as EDG.
- Exploit the open-source nature of SystemC to perform runtime reflection.
- Utilize the RP in a Testbench Generator.
- Utilize the RP in a d-VCD that displays value changes "as they happen"; hence the dynamic value change dump.

4 Reflection Provider

Here, we present details on the infrastructure for reflection and introspection. We only provide small code snippets to present our approach and the concept of using Doxygen, XML, Xerces-C++, and C++ data structure to complete the RP.

4.1 Introspection in SystemC

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

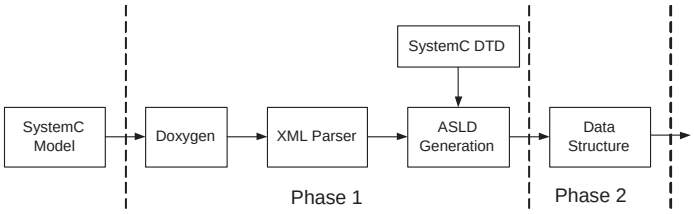

Figure 1: Design Flow for Reflection Provider

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 one code line 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 [3].

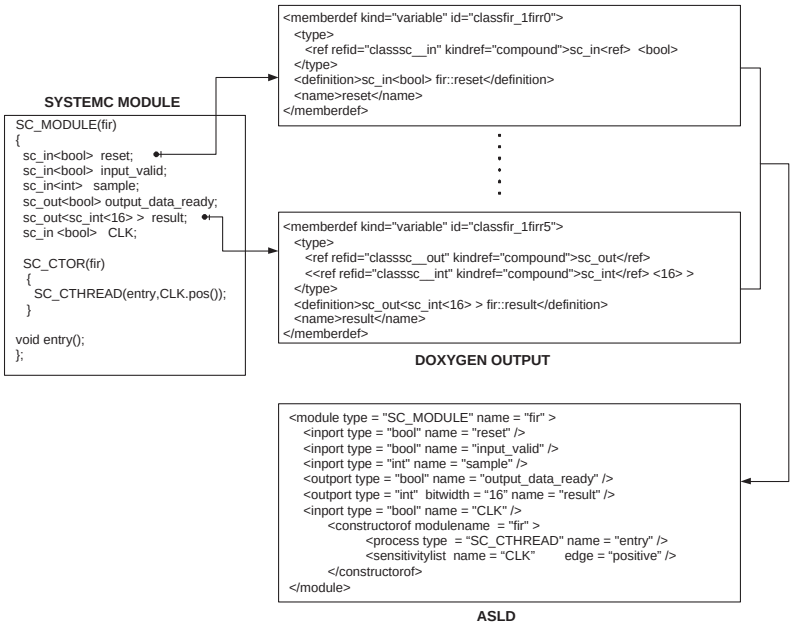

Figure 3: Doxygen XML Representation for `sc_in`

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 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 [6] 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

```
<!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>
```

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 instantiate, however 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.

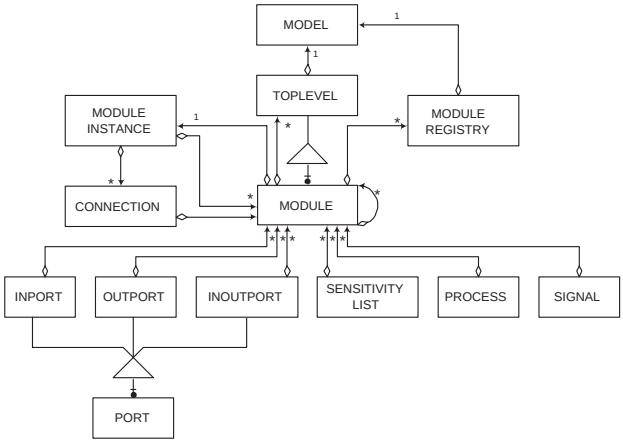

Figure 4: Class diagram showing data structure

5 Clients Using Reflection

5.1 Testbench Generator

We develop a Testbench Generator client that interacts with the RP to support automated test generation. The test generation client is built using the SystemC Verification (SCV) [8], which is a library of C++ classes, that provide tightly integrated verification capabilities within SystemC. The Testbench Generator interacts with the RP by instantiating an object of the RP and invoking the respective API calls to access the structural information pertaining to test generation. The generator takes as input a SystemC model and invokes the respective API call on the reflection object that creates the corresponding ASLD for the SystemC model. Then it invokes the API call for initialization of the data structure and enabling the introspective capabilities of the reflection object. This Testbench Generator uses these introspective capabilities to extract information such as the type, bitwidth of ports and signals to generate tests for the SystemC model. The generator also has the abilities to generate testbenchs 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 testbenchs 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 testbenchs providing which takes a data file as input with the values and their probability.

5.1.1 Testbench generation Example

We briefly describe the testbenchs generated using the FIR example from SystemC distribution. In particular, we set focus on the computation block of the FIR. We present Figure 5 that shows three testbenches using the *unconstRand*, *simpleRand* and *distRand* modes. The *unconstRand* generates unconstrained randomized testbenchs, the *simpleRand* constrain the randomization using *keep_out* and *keep_only* constructs with legal ranges specified from an input data file and the *distRand* defines *SCV_bags* that give a probabilistic distribution for the randomization. Once, the automated testbench generated, it is integrated and compiled to test the FIR block. The integration is performed manually by defining the appropriate interface between the generated testbench and the FIR block. Figure 5 displays snippets of these three modes of operation.

We intend to improve our automated testbench generation capabilities by first implementing additional clients such as coverage monitors and simulation performance monitors to better analyze the SystemC model. These additional clients will assist the Testbench Generator in making more intelligent and concentrated testbenchs.

5.2 d-VCD

Implementing runtime reflection mandates alterations to the existing SystemC source. This is unavoidable if runtime information has to be exposed for SystemC and we justify this change by having two versions of SystemC. We call our altered version SystemC-V, which the designer can use for the purpose of verification, d-VCD, and debugging of SystemC models. However, for fast simulation

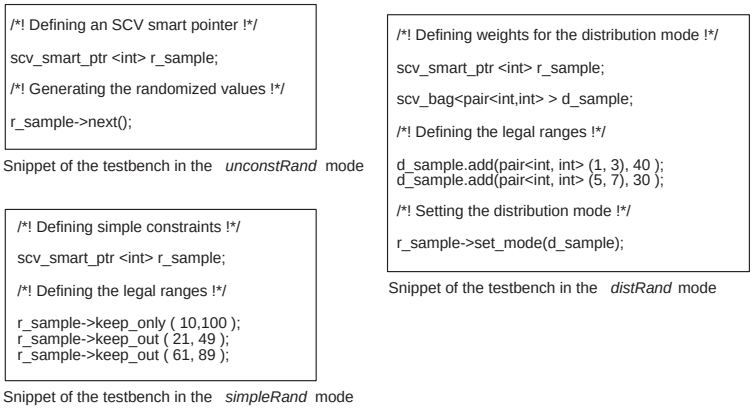

Figure 5: Code snippets for generated testbenchs

the same model can be compiled with the original unaltered version of SystemC by simply altering the library target in the Makefiles.

The d-VCD client displays signal value changes for a module "as they happen". Regular VCD viewers display VCD information from a file generated by the simulation. However, we enable the d-VCD viewer to update itself as the signals of a focused module in the SystemC model changes. Every signal value change for the module in focus communicates with the d-VCD. Likewise, at every delta cycle we send the process names on the runlist to the d-VCD. Figure 6 shows a screenshot of a GUI for the VCD using Qt [14]. To enable SystemC-V to expose this information we altered the `sc_signal` class along with adding an extra classes. Before discussing brief implementation details it is necessary to understand how we utilize the RP. In order to gain access to the reflected information, we instantiate an object of class `module` and use it as our introspective mechanism. Member functions are invoked on the instance of `module` to set the focus on the appropriate module and to introspect the characteristics of the module as explained in Section 5.2.

To facilitate SystemC for exposing runtime characteristics, we implement class `fas_sc_signal_info` that stores the signal name (`sig_name`), signal type (`sig_type`) and classification type (`sig_class`) with their respective set and get member functions. SystemC has three class representations for `sc_signal`, where the first one is of templates type `T`, the second is of type `bool` and the third is of type `sc_logic`. Each of these classes inherit the `fas_sc_signal_info` class minimizing changes to the original source. In fact, the only changes in the original source are in the constructor and the `update()` function. We require the

user to use the explicit constructor of the `sc_signal` class such that the name of the signal variable is the same as the parameter specified in the constructor. This is necessary such that an object of `sc_signal` concurs with the introspected information from the RP. We also provide a PERL script that automatically does this. The `update()` function is responsible for generating SystemC events when there is a change in the signal value and an ideal place to transmit the data to the d-VCD client. So, if the signal type is a SystemC type then the `to_string()` converts to a `string` but if it is classified as a C++ type then it is converted using `stringstream` conversions.

The explicit constructors invoke `classify_type()` which classifies the signal into either a SystemC type or a C++ type. We use the classification to convert all C++ and SystemC types values to a `string` type. This is such that multiple VCD viewers can easily interface with the values returned from SystemC-V and they need not be aware of language specific datatypes. Since all SystemC datatypes have a `to_string()` member function, it is easy to return the string equivalent for the value. However, for C++ datatypes we employ a work around using `stringstream` conversion to return the string equivalent. Now, we are successfully able to translate any native C++ and SystemC datatypes to their string equivalent. However, the compilation fails when a SystemC model uses signals of C++ and SystemC types together because for C++ datatypes the compiler cannot find a `to_string()` function. An immediate solution to this involves implementing a templated container class for the templated variables in `sc_signal` class such that the container class has a defined function `to_string()` that allows correct compilation. We add class `containerT<T>` as a container class and replace variable instances of type `T` to `containerT<T>` to circumvent the compilation problem. We interface the runtime VCD with the Qt VCD viewer implemented by us, shown in Figure 6.

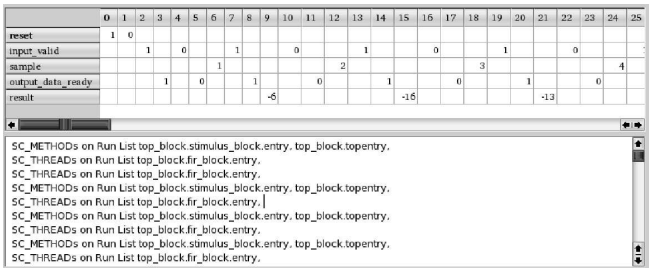

Figure 6: d-VCD Output

5.2.1 Runtime Runlist Information

d-VCD is a good example of runtime reflection in SystemC and we extend it further by exposing runlist information at runtime as another example of runtime reflection. Being able to see the processes on the process runlists is a significant advantage during debugging and we provide this capability by altering the SystemC `sc_simcontext` class. Figure 6 also shows the output for the processes on the runlist. Exposing the process name to the d-VCD client itself does not require the Reflection Provider since the process names are available in SystemC `sc_module` class via the `name()` member function. However, using the RP we provide the user with more concentrated visuals of model execution by enabling the user to specify which particular module's processes should be displayed. This requires querying the RP for the name of the modules in focus and only returning the names of the processes that are contained within those modules. Implementing this capability required stepping through SystemC runlist queues and monitoring whether they match the modules of interest and transmitting the name to the d-VCD client.

6 Conclusion

We present a methodology where we employ the use of public-domain tools such as Doxygen and Apache's Xerces-C++ to present introspective capabilities in SystemC. We describe our approach in detail and also present two clients that utilize the R-I capabilities implemented in SystemC. Clients such as the Testbench Generator primarily use the structural reflection information to generate automated testbenchs, but it is essential to perform runtime reflection to aid in better debugging tools such as the d-VCD. We intend to use runtime reflection to facilitate performance metric calculation further allowing us to locate time consuming *hotspots*. Overall, R-I in SystemC opens up possibilities for a wide range of tools from integrated development environments to better visualization tools, GUI based SystemC modeling, validation and verification.

Our future work involves providing more clients such as coverage monitoring clients that interact with the Testbench Generator client to refine the testbenchs and performance metric clients that analyze hotspots in the model. We are investigating the possibility of using Adaptive Communication Environment and The ACE ORB [1, 12] as middleware for the interactions between the RP and clients.

References

[1] ACE. Adaptive Communication Environment. http://www.cs.wustl.edu/ schmidt/ACE.html.

[2] 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.

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

[4] Edison Group. Edison C++ Front-End. http://www.edg.com.

[5] J. Lapalme, E. M. Aboulhamid, G. Nicolescu, L. Charest, F. R. Boyer, J. P. David, and G. Bois. .NET Framework – A Solution for the Next Generation Tools for System-Level Modeling and Simulation. In *Design and Test Automation in Europe*, 2003.

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

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

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

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

[10] SPECC Team. SpecC. Website: http://www.ics.uci.edu/specc/.

[11] B. Stroustroup. *The C++ Programming Language*. Addison-Wesley, 2003.

[12] TAO. Real-time CORBA with TAO (The ACE ORB). http://www.cs.wustl.edu/ schmidt/TAO.html.

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

[14] Troll Tech. Qt. Website: http://troll.no.