Extracting Behavior and Dynamically Generated Hierarchy from SystemC Models

Harry Broeders
Delft University of Technology
Delft, The Netherlands
+31-15-2606308
J.Z.M.Broeders@hhs.nl

René van Leuken
Delft University of Technology, Circuits and Systems
Delft, The Netherlands
+31-15-2786696
T.G.R.M.vanLeuken@tudelft.nl

ABSTRACT
We present a novel approach to extract the dynamically generated module hierarchy and its behavior from a SystemC model. SystemC is a popular modeling language which can be used to specify systems at a high abstraction level. The module hierarchy of a SystemC model is dynamically constructed during the execution of the elaboration phase of the model. This means that a system designer can build regular structures using loops and conditional statements. Currently, most SystemC tools can not cope with SystemC models for which the module hierarchy depends on dynamic parameters. In our approach this hierarchical information is retrieved by controlling and monitoring the executing of the elaboration phase of the model using a GDB debugger. Thereafter, the behavioral information is retrieved by using a GCC plug-in. This plug-in produces abstract syntax trees in static single assignment form. This behavioral information is linked with the hierarchical information. Our approach is completely non-intrusive. The SystemC model and the SystemC reference implementation can be used without any modification. We have implemented our approach in a SystemC front-end called SHaBE (SystemC Hierarchy and Behavior Extractor). This front-end facilitates the development of future SystemC visualization, debugging, static verification, and synthesis tools.

Categories and Subject Descriptors
I.6.4 [**Simulation and Modeling**]: Model Validation and Analysis.

General Terms
Design.

Keywords
Automation, analyze, SystemC, system specification, front-end, elaboration, dynamic module hierarchy, AST, GCC, GDB.

1. INTRODUCTION
Many modern electronic systems are implemented as multi-processor systems on a single chip. Such systems are far to complex to describe their hardware and software at a low-level of abstraction. An established approach to cope with this complexity is to specify systems at the Electronic System-Level (ESL). A recent overview of ESL tools is given by Gajski et al. [4]. An ESL design flow typically starts with the development of a functional model of the system. This model is described, for example in C++ or Matlab, and verified by means of simulation. During the next step of the system-level design flow a design space exploration is performed to optimize design metrics under a given set of constraints. The resulting system specification usually consists of a complex hierarchical structure. The optimized model will then be further refined into implementation-level models for the system's software and hardware.

SystemC is developed by the Open SystemC Initiative (OSCI), is described in IEEE standard 1666-2005 [6], and is implemented as a C++ framework. The primary way to deal with complexity in SystemC is to apply modularization. The module hierarchy of a SystemC model is dynamically constructed during the execution of the elaboration phase of the model. This means that a system designer can build regular structures using loops and conditional statements. Writing code which dynamically generates the hierarchical structure of a system instead of statically laying out this structure is strongly preferred because the dynamic generation can be parameterized which makes the model easier to modify, easier to extend, and easier to reuse. Currently, most ESL tools can not cope with SystemC models for which the hierarchy depends on dynamic parameters. In theory, there is no need to restrict the designer to a subset of C++ for the part of the SystemC code which describes the construction of the model i.e. the elaboration phase, but in practice, these ESL tools do have such restrictions.

In this paper a novel approach to extract the dynamically generated module hierarchy and its behavior from a SystemC model is presented. In our approach the hierarchical information of a SystemC model is retrieved by executing the elaboration phase of the model under control of a debugger. Thereafter, the behavioral information of the model is retrieved by using a C++ compiler extension. Finally, the hierarchical information and the behavioral information are combined and stored as an Intermediate Representation (IR) which can be used by tools build upon this front-end. Our approach is completely non-intrusive, i.e. no changes are required in the standard tool flow. The SystemC model and OSCI's SystemC implementation can both be used as is. The only precondition is that both are compiled to include debug information.

DAC'11, June 5-10, 2011, San Diego, California, USA

In the next section a short overview of SystemC is given. Other SystemC front-ends and their limitations are described in Section 3. In Section 4 our approach is explained and an open-source SystemC front-end called SHaBE is presented. The experimental results of using SHaBE are presented in Section 5. In the last part of this paper conclusions are drawn and directions for future work are given.

2. SYSTEMC

SystemC is implemented as a C++ framework which consists of a class library and a simulation kernel. The library consists of classes, macros, and templates which can be used to model a concurrent system using hardware-oriented data types and communication mechanisms. A SystemC model is structured by using modules. A module encapsulates a part of the system which is being modeled and has communication ports to communicate with other modules within the model. A module can contain other modules. Communication ports can be interconnected by using channels. The simulation kernel can be used to execute a SystemC model. This execution is divided into two phases: the elaboration phase and the simulation phase. During the elaboration phase the modules are instantiated and initialized by executing their constructors. During this initialization the connections between the modules are set up. Because the modules are instantiated and connected by executing C++ code any valid C++ language construct can be used. For example: the configuration of the modules can be read from a file or may depend on command line arguments. The behavior of a module is defined by one or more processes. A SystemC process is defined in the form of a C++ member function that is registered with the SystemC simulation kernel. OSCI provides an open-source implementation of the SystemC framework which is available free of charge. Using this implementation a SystemC model can be compiled by any standard conforming C++ compiler and can be executed. This execution simulates the model and provides information that can be used to dynamically verify the model. But simulation is not the only thing for which a SystemC model can be used. Users of such a model may want to visualize, debug, statically verify, or synthesize it. Tools which fulfill these needs must have a SystemC front-end which is able to retrieve the dynamically generated hierarchy and its behavior from the model.

3. RELATED WORK

There are three different approaches followed by existing SystemC front-ends: static, dynamic, and hybrid.

3.1 Static Approach

Front-ends which follow the static approach look upon SystemC as a C++ language extension. They provide a SystemC parser which is, in most cases, based on a C++ parser. Front-ends described in the literature which follow this approach are: KaSCPar [11], ParSyC [3], SCOOT [2], and SystemCXML [1]. The front-ends in this category use static analysis to retrieve the module hierarchy of the model and therefore can not handle models for which the module hierarchy depends on dynamic parameters.

3.2 Dynamic Approach

A SystemC front-end described in the literature which follows the dynamic approach is: Quiny [13]. Quiny uses a self-reflective approach to retrieve the hierarchy and the behavior of a SystemC model. The model is compiled and linked against the Quincy library which replaces the SystemC library. When this model is executed it subsequently produces the IR. Quincy can easily retrieve the hierarchy of the model during the execution of the elaboration phase. Due to limitations of the C++ language it is not possible for Quincy to retrieve the behavior of a model without help of the end-user.

3.3 Hybrid Approach

SystemC front-ends that fall into the hybrid category try to combine the best features of the static and the dynamic approach. The biggest challenge for this approach is to find the link between the information retrieved by the parser and the information found by executing the elaboration phase of the SystemC model. Front-ends described in the literature which follow this hybrid approach are: an unnamed successor of ParSyC [5], Pinapa [10], and its successor PinaVM [9].

3.3.1 Unnamed Successor of ParSyC

The authors of ParSyC briefly describe a hybrid approach in [5]. They use a PCCTS [12] based parser to collect the static information and a code generator to evaluate run time information. The described approach is split into four phases. First, the SystemC model is converted into an AST (Abstract Syntax Tree) by the parser. During the second phase recorder functions are added to this AST and an instrumented version of the original model is generated. The elaboration phase of this model is executed during the third phase. The injected recorder functions now record the state of all variables of the model after a change of their value. This dynamic information is added to the AST during the last phase.

3.3.2 Pinapa

Pinapa is implemented as a patch for SystemC and a patch for GCC. The patched version of GCC produces the AST of the SystemC model and the patched SystemC version is used to execute the elaboration phase to retrieve the structural information from the model. Pinapa then links this structural information with the behavioral information from the AST. Pinapa uses outdated versions of GCC and SystemC and is therefore itself outdated.

3.3.3 PinaVM

PinaVM uses LLVM-GCC [7] to compile the SystemC source code into LLVM bitcode. It uses this bitcode to execute the elaboration phase which reveals the module hierarchy of the SystemC model. The SystemC constructs in the bitcode which describes the behavior of the model are recognized by PinaVM. This information is linked to the module hierarchy which was found by executing the elaboration phase. To accomplish this PinaVM recognizes the mangled names of calls into the SystemC library such as `read`, `write` and `wait`. When PinaVM has identified the function calls into the SystemC library it must link specific parameters of such a call with the SystemC module hierarchy. For example, in a call to the `write` function of an output port, the parameter `this` which specifies the port which is written to must be identified. The value of this parameter can be the result of any arbitrary computation. The key idea of PinaVM is to identify the bitcodes which are used to compute the parameter of interest and then to construct a new LLVM function which contains those bitcodes and produces the value of the parameter. Once build, this function is executed and the value of the parameter can be linked with the appropriate object in the model's module hierarchy.

3.4 Conclusions Drawn from the Survey of SystemC Front-Ends

From our survey of SystemC front-ends we draw the conclusion that PinaVM is the only available open-source SystemC front-end which can retrieve the module hierarchy as well as the behavior of all modules from a SystemC model. But this approach is limited to models in which the ports which are being used in the behavioral description can be determined statically. This conclusion is confirmed in [8] and [9]. So for all currently available open-source SystemC tools, the system designer can not utilize the full expressive power of C++ to describe the dynamic generation of the hierarchical structure of the model.

4. Extracting the Behavior and the Dynamically Generated Hierarchy

We present our solution which 1) controls the elaboration phase of the execution of the model and extracts the model hierarchy, 2) retrieves the Abstract Syntax Tree (AST) from the SystemC model, and 3) combines all collected information to produce the dynamically generated hierarchy and the behavior of the SystemC model.

The hierarchical and behavioral information of a SystemC model is extracted in three steps. During the first step the generated hierarchy is retrieved by executing the elaboration phase of the SystemC model under the control of a debugger. The C++ member functions used to describe the behavior of the SystemC processes, and the source files in which these member functions are defined, are also identified during the first step. In step two the AST of the functions which describe the behavior of the model are retrieved. In the last step of our approach the ASTs of all analyzed functions are combined with the information found in the first step. Finally an IR in XML is produced which contains all hierarchical and behavioral information from the model.

We have implemented our approach described in a SystemC front-end called SHaBE (SystemC Hierarchy and Behavior Extractor), Figure 1. In Section 4.1, 4.2, and 4.3 the three steps of our approach are explained.

4.1 Extracting the Dynamically Generated Module Hierarchy

SHaBE uses the open source GNU debugger (GDB) to retrieve the module hierarchy from a SystemC model. This debugger has a special interface called GDB/MI which is specifically intended to support the development of systems which use GDB as a component. SHaBE uses this interface to run and monitor the elaboration phase of the executable model. During this execution SHaBE builds up an internal data structure which represents the module hierarchy of the model. This is achieved by using GDB commands to set breakpoints, continue the execution, inspecting the stack, inspecting function arguments, inspecting the members of an object, etc. The source code of the SystemC library and kernel has been carefully analyzed to determine the function calls which need to be monitored. As an example we describe how the C++ names of all ports within the SystemC module hierarchy can be found. In SystemC ports must be defined inside modules. Both ports and modules are derived from the same base class. SHaBE uses the GDB/MI interface to place a breakpoint at the constructor of this base class `sc_object`. When this breakpoint is hit, the sequence of constructor calls is traced back. This will reveal the most derived type of the object which is created. If this object is derived from `sc_module` its members must be inspected to find the ports of this module. Because the base class constructor `sc_object` is executed before the constructor of the derived class it is not possible to inspect the data members of the derived class. It is possible though, by using GDB, to find the names and addresses of all data members declared in the derived class. This is possible because in C++ the memory for an object is allocated before the constructor for this object is called. For plain data members the name and address is stored in a look-up table inside SHaBE. This table can be used to look up the name of a port, using its address, when it is created later on. The C++ names of all submodules, channels, and exports can be found using the same method.

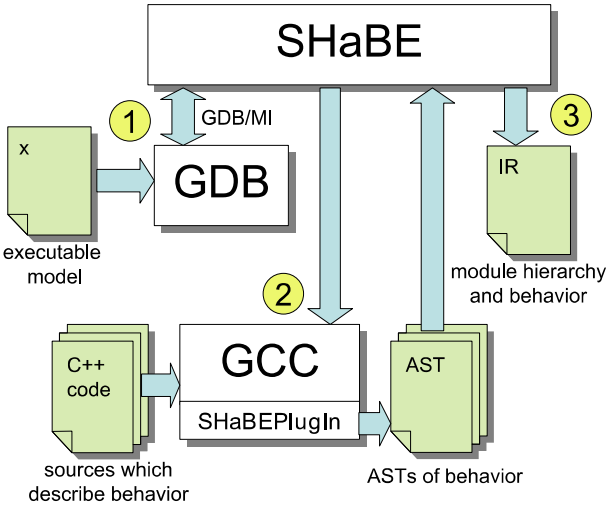

Figure 1. The architecture of SHaBE (SystemC Hierarchy and Behavior Extractor).

A complication is the fact that a submodule, channel, port, or export declared within a SystemC module can be stored in a pointer or array variable. Arrays are split into individual elements which are identified by attaching their index, in between square brackets, to the array name. When the address of a SystemC object is placed inside a pointer data member of a module, using GDB, a watchpoint is set on this pointer variable. When this watchpoint is hit, the value which is written into the pointer is reported by GDB. This address is used by SHaBE to see if a SystemC object was created at this address before. When such an object is found, its name is set to the name of the pointer prefixed with the dereference operator. SHaBE is able to find all object names except the names of objects which are accessed via a pointer to an array of objects. This is caused by the limitation of the C++ language to differentiate between a pointer to an array of objects and a pointer to a single object.

4.2 Extracting the Behavior

The behavior of the SystemC model is extracted by using the GNU C++ compiler GCC. Since version 4.5.0 GCC can be extended without modifying its source code by using a plug-in. A GCC plug-in can schedule the execution of specific code in between, or as a replacement of, some passes of the GCC compilation process using the pass manager. We have developed a plug-in for GCC named SHaBEPlugIn which executes code after the Static Single Assignment (SSA) pass of GCC is finished. At that moment the AST in SSA format is constructed but no optimizations are performed. SHaBEPlugin converts this non-optimized AST into an IR. Many optimization algorithms are

either enabled or strongly simplified by the use of SSA. Therefore future tools build upon SHaBE can easily apply such optimizations.

The AST tree is simplified by SHaBEPlugin. Calls to functions and overloaded operators of the SystemC library are represented as simple nodes in the resulting IR. At this moment SHaBEPlugin can only recognize a limited number of functions and operators. For example, all `read`, `write`, and `wait` function calls are recognized.

A member function which defines the behavior of a process can call other user defined functions. At this moment these calls appear in the IR as call nodes.

4.3 Combining the Hierarchical Information with the Behavioral Information

As described in Section 4.1, SHaBE retrieves the C++ names of all ports and channels from a SystemC model. These ports and channels can be used in the description of the behavior of the model which is analyzed in the second step of SHaBE. Ports are used to access the channels which are bound to these ports during the elaboration phase of the execution of the model. These bindings are also retrieved in the first step of SHaBE. If the channels and ports used in the member functions that describe the behavior of the model are stored in a plain data member, then the name found in the analysis of the member function are the C++ names of these data members. In this case ports and channels found in the second step can be simply linked with the port and channels found in the first step by using the fully qualified C++ names of these ports and channels.

If ports and/or channels are stored inside arrays or accessed via pointers the association of ports and channels between the two steps of SHaBE is a bit more involved. Arrays of ports and arrays of channels are split into individual objects in the first step of SHaBE. The name of these objects is the expression needed to access such an individual object within the array. When such an array element is accessed in the member function which describes the behavior of a SystemC process then the subscript operator appears in the IR produced by the second step of SHaBE. If the subscript used in the subscript operator is a compile time constant then SHaBE can associate the access with a specific array element found during the first step. In this case the subscript operator can be removed from the behavioral IR produced by the second step of SHaBE. But, if the subscript used in the subscript operator is a variable then the subscript operator models a multiplexer or demultiplexer. A multiplexer is modeled if the subscript operator is used as an rvalue and a demultiplexer is modeled if the subscript operator is used as an lvalue. In this case, the array access found in the second step of SHaBE will be associated with all elements of the array found during the first step of SHaBE. Which of these elements is selected during run-time depends on the values of run-time variables and this behavior is captured in the IR produced by the second step of SHaBE.

If a port or channel is accessed via a pointer then a similar method is used to remove the dereference operator from the behavioral IR produced by the second step of SHaBE. When the object which is referred to by the pointer can not be determined statically the dereference behavior is captured in the IR. A tool which is build upon SHaBE can replace reads and writes to ports with read and writes to the channels which are bound to these ports. This makes it possible to construct the complete Control Data Flow Graph (CDFG) of the model.

4.4 Implementation Details and Dependencies

Both the SHaBE program as well as the SHaBEPlugIn is implemented in C++. Because SHaBE uses GDB it can only handle executable models which can be debugged by GDB. SHaBE uses GDB to set breakpoints on certain functions in the SystemC library and to inspect certain data members of classes defined in the SystemC library. Therefore, SHaBE depends on a specific implementation of the SystemC library i.e. the OSCI implementation of SystemC version 2.2.0. SHaBE is developed on Ubuntu 10 using GCC 4.5 and GDB 7.1. The source code of SHaBE and all test programs can be downloaded free of charge from http://shabe.sourceforge.net.

5. RESULTS

In Figure 2 a generic model for an N^{th} order Finite Impulse Response (FIR) filter with symmetric coefficients is given. The data type and the order of the filter are template parameters of this model. The FIR filter module uses three different kinds of submodules: module `S` models a simple adder, module `M` models a simple multiplier which multiplies an input value with a constant filter coefficient, and module `D` models a register. The hierarchical structure of the filter is dynamically generated during the elaboration phase by executing the constructor of the `FIR` module.

```
template <typename T, unsigned int ORDER>
SC_MODULE(FIR) {
  sc_in_clk clk;
  sc_in<T> sample;
  sc_out<T> result;
  FIR(sc_module_name name,
                      const T coeff[ORDER]);
  ~FIR();
private:
  static const unsigned int MULT=ORDER/2+1;
  M<T> *m[MULT];
  S<T> *s[ORDER];
  D<T> *d[ORDER];
  sc_signal<T> mout[MULT], sout[ORDER-1],
                                  dout[ORDER];
};
```

Figure 2. Generic SystemC model of an N^{th} order FIR filter with symmetric coefficients.

Figure 3 shows the implementation of the constructor of this module. The filter coefficients are passed to the module using a constructor parameter. These coefficients can be unknown at compile time because they can be, for example, read from a file during the elaboration phase of the model.

The structure of the filter is dynamically created in the constructor. The multipliers, adders and delay submodules are dynamically created by calling `new`. The hierarchical structure of the `FIR` is created by binding the ports of these submodules to internal signals of the `FIR` module or to ports of this module. These connections are made during the execution of the elaboration phase. For example, the output of the last adder submodule must be connected to the output port `result` of the filter. The output of every other adder submodule `*s[i]` must be connected to an internal signal `sout[i]`. This is described by the `if` statement shown in Figure 3. Please note that the submodules must be dynamically created by using `new` because they have a

non empty constructor as required by the SystemC standard [6]. Therefore these submodules can not be placed in a static array.

```
FIR::FIR(sc_module_name name,
         const T coeff[]): sc_module(name) {
  for (unsigned int i(0); i<MULT; ++i) {
    m[i] = new M<T>("", coeff[i]);
    m[i]->in(sample);
    m[i]->out(mout[i]);
  }
  for (unsigned int i(0); i<ORDER; ++i) {
    s[i] = new S<T>("");
    s[i]->in1(dout[i]);
    s[i]->in2(i<MULT-1 ? mout[i+1] :
                         mout[ORDER-(i+1)]);
    if (i == ORDER-1)
      s[i]->out(result);
    else
      s[i]->out(sout[i]);
    d[i] = new D<T>("");
    d[i]->clk(clk);
    d[i]->in(i==0 ? mout[0] : sout[i-1]);
    d[i]->out(dout[i]);
  }
}
```

Figure 3. Dynamic generation of the hierarchical structure of the filter.

Figure 4 shows an instantiation of the `FIR` module. The filter coefficients are read from a file and passed to the module `fir`. The date type to be used (`sc_fixed<32, 2>`) and the order of the filter (`5`) are passed as template parameters to the `FIR` module.

```
sc_fixed<32, 2> coeff[3];
read_from_file(coeff);
FIR<sc_fixed<32, 2>, 5> fir("fir", coeff);
```

Figure 4. Instantiation of a 5th order FIR filter.

This instantiation generates the module hierarchy shown in Figure 5 during the elaboration phase of the model.

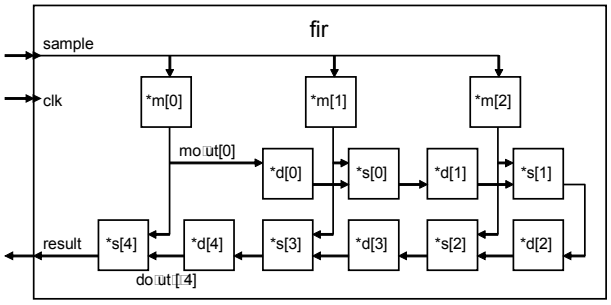

Figure 5. Dynamically generated hierarchical structure of a 5th order filter extracted by SHaBE.

The names of the modules, signals, and ports shown in figure 5 are the expressions which are used to access these objects from the C++ code which describes the behavior of the model. The `clk` input of the `fir` module is connected to all `d` modules but these internal signals are not shown in Figure 5. To prevent clutter only a few internal signal names are shown. The hierarchy is extracted from the model by SHaBE.

The behavior of the filter is determined by its hierarchical structure and by the behavior of its submodules. For example, the behavior of each adder is described in the member function shown in Figure 6. The objects `out`, `in1`, and `in2` are ports of the adder module `S`.

```
template<typename T>
void S<T>::behavior() {
  out.write(in1.read() + in2.read());
}
```

Figure 6. The description of the behavior of an adder.

SHaBE extracts an AST which describes this behavior, see Figure 7a. This information can be combined with the hierarchical information. For example, the AST for the submodule `*s[4]` is shown in Figure 7b. Please note that the actual signals which are read and written are shown in the AST of Figure 7b. This makes it possible to construct the complete dataflow graph of the FIR filter from the IR generated by SHaBE.

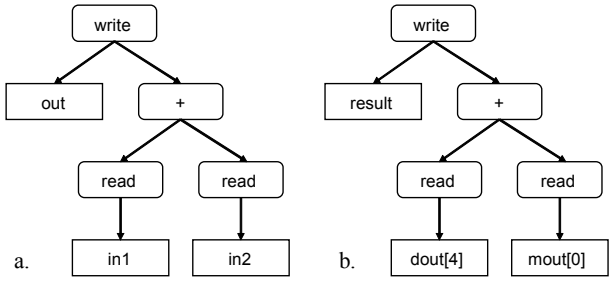

Figure 7. AST of a) the adder module S and b) the adder module instantiation `*s[4]`.

The first step of SHaBE has a time complexity of $O(n \log n)$, where n is the number of SystemC objects used in the model. A breakpoint is hit and some inspections are performed each time an object is created. Such an inspection can include a lookup by address of an object found earlier. A search tree, implemented in the standard C++ type `map` is used to store all objects. The address of an object is used as the search key. Therefore, the lookup will take $O(\log n)$ time and will have to be applied for a maximum of n times which results in an overall time complexity of $O(n \log n)$.

The time required by the second step of SHaBE is comparable with the time needed by the GCC C++ front-end.

The time complexity of the third step of SHaBE is $O(n \log n)$ where n is the number of SystemC objects used in the model. All objects found in the second step must be linked by name to the objects found in the first step.

Figure 8 shows the execution time of SHaBE for the FIR filter presented in Figure 2 for different values of the template parameter `ORDER` which specifies the order of the filter. The duration of the second step of SHaBE is independent of the value of `ORDER` because the functions which describe the behavior of the FIR filter are independent from the order of the filter. The number of `sc_objects` in the model (n) is directly proportional to order of the filter. Figure 8 shows that the time complexity of the first and last step of SHaBE is $O(n)$. This is better than the expected complexity of $O(n \log n)$ which can be explained by the fact that the look-up time is very small compared with the time needed to communicate with GDB.

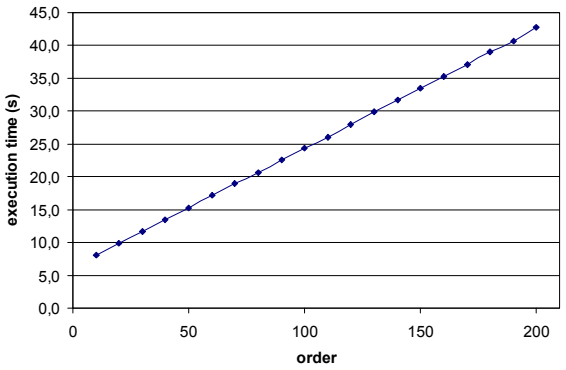

Figure 8 Execution time of SHaBE for different orders of the FIR filter presented in Figure 2.

Table 1 shows the execution time of SHaBE for some SystemC models. The column gcc –S shows the execution time of the front-end of the GNU GCC compiler. All execution times were measured on a PC equipped with 2 GB of RAM and an AMD Athlon 64 processor running at 2 GHz. For the models shown in Table 1 the execution time of SHaBE is at most 1.6 times the execution time which is needed by the front-end of the GCC C++ compiler.

Table 1. Execution times of SHaBE

Model	# SystemC objects	# Source lines	Execution time (s)	
			gcc -S	SHaBE
FIR[1]	94	214	4.7	7.5
FIR[2]	37	296	8.3	8.4
RISC CPU[3]	310	1960	20.8	29.0

[1] FIR model instantiated in Figure 4.
[2] RTL version of a FIR filter provided by OSCI.
[3] RISC CPU provided by OSCI.

6. CONCLUSIONS

A novel method to extract the behavior and the dynamically generated hierarchy from a SystemC model is described in this paper. SystemC designers can write C++ code which dynamically generates the hierarchical structure of their models during the elaboration phase. Currently available SystemC front-ends can not cope with SystemC models for which the module hierarchy depends on dynamic inputs. The open-source SystemC front-end SHaBE which is introduced in the paper can handle models which use dynamic input such as console input, file input, and command line arguments. The experiments have shown that the execution time of SHaBE is comparable with the time which is needed to compile the model. This tool can be used as a front-end for future SystemC development tools which can visualize, debug, statically verify, or synthesize a model. A designer can use this tool in combination with the freely available SystemC framework provided by OSCI and the freely available GNU compiler and debugger. Furthermore, a designer can use our tool without any modification to the source code of the model nor the library, i.e. our tool is non-intrusive.

7. FUTURE WORK

A possible application which uses SHaBE as a front-end, would be a tool which converts a SystemC model comprising a parameterized dynamic hierarchy into a SystemC model with a fully expanded hierarchy. The resulting SystemC model can then be further processed using existing tools.

Also, it would be interesting to investigate if the approach used in SHaBE can also be applied to SystemC models which use the SystemC Transaction Level Modeling (TLM) and/or the SystemC Analog/Mixed-signal (AMS) extensions.

8. ACKNOWLEDGMENTS

Special thanks to Bas van den Aardweg who implemented the first version of SHaBEPlugIn.

9. REFERENCES

1. Berner, D., Talpin, J.-P., Patel, H.D., Mathaikutty, D., and Shukla, S.K. SystemCXML: an extensible SystemC front end using XML. *Forum on specification and Design Languages (FDL)*, Lausanne, Switzerland, September 27–30, 2005, 405–409.
2. Blanc, N., Kroening, D., and Sharygina, N. SCOOT: a tool for the analysis of SystemC models. *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Budapest, Hungary, March 29–April 6, 2008, 467–470.
3. Fey, G., Grosse, D., Cassens, T., Genz, C., Warode, T., and Drechsler, R. ParSyC: an efficient SystemC parser. *Synthesis And System Integration of Mixed Information technologies (SASIMI)*, 2004, 148–154.
4. Gajski, D.D., Abdi, S., Gerstlauer, A., and Schirner, G. *Embedded System Design*. Springer US, Boston, MA, 2009.
5. Genz, C. and Drechsler, R. Overcoming limitations of the SystemC data introspection. *Design, Automation & Test in Europe (DATE)*, Nice, France, April 20–24, 2009, 590–593.
6. IEEE. *Std 1666 – 2005 IEEE Standard SystemC Language Reference Manual*. 2006.
7. Lattner, C. and Adve, V.S. LLVM: a compilation framework for lifelong Program Analysis & Transformation. *Code Generation and Optimization (CGO)*, Palo Alto, California, March 2004, 75–88.
8. Marquet, K., Moy, M., and Karkare, B. A theoretical and experimental review of SystemC front-ends. *Forum on specification and Design Languages (FDL)*, Southampton, UK, September 14–16, 2010, 124–129.
9. Marquet, K. and Moy, M. PinaVM: a SystemC Front-End Based on an executable intermediate representation. *Embedded software (EMSOFT)*, Scottsdale, Arizona, USA, October 24–29, 2010, 79–88.
10. Moy, M., Maraninchi, F., and Maillet-Contoz, L. PINAPA: an extraction tool for SystemC descriptions of systems-on-a-chip. *Embedded software (EMSOFT)*, Jersey City, NJ, USA, September 18–22, 2005, 317–324.
11. Parfuntseu, P. *KaSCPar Karlsruhe SystemC Parser Suite Online documentation*. 2006.
12. Parr, T.J. *Language translation using PCCTS and C++: A reference guide*. Automata Publishing Company, San Jose, CA, USA, 1997.
13. Schubert, T. and Nebel, W. The Quiny SystemC front end: self-synthesising designs. *Forum on specification and Design Languages (FDL)*, Darmstadt, Germany, September 19–22, 2006, 135–143.