Overcoming Limitations of the SystemC Data Introspection

Christian Genz
Institute of Computer Science
University of Bremen
28359 Bremen, Germany
Email: genz@informatik.uni-bremen.de

Rolf Drechsler
Institute of Computer Science
University of Bremen
28359 Bremen, Germany
Email: drechsle@informatik.uni-bremen.de

***Abstract*—Today verification, testing and debugging of SystemC models can be applied at an early stage in the design process. To support these techniques gaining required information of the respective model, the *SystemC Verification Library* (SCV) implements a concept called data introspection. Unfortunately data introspection holds problems that arise with increasing usage of language features. Native C++ data types for instance will not appear in meta-data extracted by introspection facilities.**

In this paper we propose a non-intrusive analysis concept to overcome the drawbacks of traditional data introspection. The presented approach is a hybrid technique joining a parser to collect statical information and a code generator to evaluate run time information.

***Index Terms*—SystemC, data introspection, analysis, intermediate representation**

I. Introduction

Meanwhile the system description language SystemC is a widely accepted standard for hw/sw co-designs. In fact SystemC has even been approved as a standard by the IEEE consortium [1]. Complementary to *Hardware Description Languages* (HDLs) SystemC offers concepts and techniques to system architects, which have been available for software programming languages exclusively in the past (e.g. object orientation). Thus, the time required for development can be shortened drastically by parallelizing several development steps, e.g. the generation of test suites, the synthesis, the programming of applications and the creation of driver software. One premise for this gain of productivity is the availability of C++ features to SystemC architects. Typical C++ features are polymorphism that support a clever reuse of existing algorithms and data structures, as well as the inclusion of arbitrary software libraries into prototypical hw/sw co-designs. In addition, a wide range of abstraction levels is supported, allowing the description of cycle accurate hw-structures as well as untimed software algorithms.

But even though the power and the flexibility of the language represent its greatest benefits, these attributes have vast impact on the complexity of SystemC design analysis. As can be seen in Figure 1 the analysis is crucial for many SystemC applications like synthesis [2], verification [3], [4], debugging [5] or visualization [6], [7]. They all require an analysis concept to collect meta data from the model to operate on.

SystemC supports the extraction of meta data by using the kernel interface at run time. Additionally the *Open SystemC Initiative* (OSCI) introduced the SCV library in December 2002. The SCV delivers an extension of SystemCs analysis capabilities named data introspection. While data introspection uses run time information to assemble its meta data of a given model, the meta data itself lacks information that has been lost during the compilation process. We implemented a tool realizing the analysis concept presented in this paper. Our tool preserves SystemC applications from ”reinventing the wheel” by overcoming the following disadvantages of the SCV.

- The circuit behavior, i.e. the operational semantic of the analyzed SystemC program, shall be described in meta data.
- Not only SystemC data types but also user defined types shall appear in the analysis results. Without knowledge of inheritance and member types inside classes the observed design hierarchy would be incomplete.
- All names of declarations (as variables, data types and functions) must be known after analysis. Otherwise the names of ports, modules and signals are not stored in the meta data or may be ambiguously named using internal SystemC names (`sc_object`).

The paper at hand is structured as follows: Section II outlines existing approaches in this area and compares them to ours. Since the approach presented in this paper is separated into several transformations, Section III, containing the main contribution, is subdivided into four parts. The first part of Section III introduces the architecture of the approach. In the following parts of Section III the details of our analysis are described. In Section IV we exemplify the extraction of meta

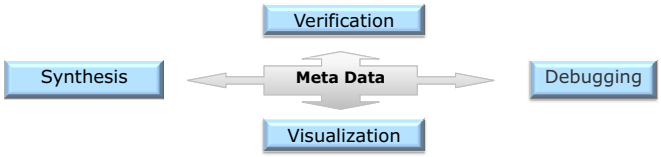

Fig. 1. Applications for analysis

This work was supported in part by the German Federal Ministry of Education and Research (BMBF) and by Concept Engineering GmbH, Freiburg, Germany within the project Herkules under contract no. 01 M 3082.

data from a given SystemC source file. Finally this work is concluded by a summary in Section V.

II. Related Work

Some existing approaches are based on simulation techniques only (e.g. data introspection or kernel usage). But simulation is not able to collect information that is lost after C++ compilation, like array dimensions or declaration names. Others offer extended facilities to extract the static architecture of an analyzed model, too. Clearly the first category cannot solve the problems mentioned in Section I.

Große et.al. extract hierarchy information of a model in [8] accessing the SystemC kernel at execution time. All objects that have been declared with a data type that is derived from `sc_object` can be found in a list of the SystemC kernel after elaboration. This list can be accessed through the kernel interface and stores important details of the models hierarchy. By traversing this list the underlying modules and their connecting signals can be accessed. Afterwards these data is used to partially display the structure of the model.

Quiny [9] is a so called self-synthesizing approach. Similar to [8] Quiny operates during run time of the processed system model. But in difference to the approaches mentioned above Quiny is not limited to the use of data introspection. Quiny rather computes a complete *Intermediate Representation* (IR) of an analyzed model. The tool consists of wrappers for SystemC macros, data types and C++ keywords. Before compiling the analyzed model, the included wrappers are expanded by the preprocessor. During elaboration phase the expanded code fragments build up an IR that collects meta data of the executed model. While many C++ data types are concatenated keywords, Quiny cannot substitute those data types non-intrusively.

A further tool which was developed for analysis only is KaSCpar [10] (Karlsruhe SystemC Parser). KaSCpar is a collection of various programs realizing syntactical analysis as well as an elaboration. KaSCpar derives a parse tree from a SystemC model that contains the operational semantic of the program. But KaSCpar is not feasible of capturing state spaces.

In contrast to all other work presented above, PINAPA [3] is a hybrid approach. PINAPA is made especially for the purpose of extracting information from a given SystemC program and is based on the *GNU Compiler Collection* (GCC). The integration of the GCC allows a comfortable way to derive a parse tree from a SystemC program. In a following elaboration phase the nodes of the parse tree that symbolically represent a value are connected to the corresponding value. Since the parse tree is not able of capturing dynamic structures that grow during run time, they are not observed completely. As well as all other related work PINAPA is not able of supporting cross probing.

Our approach is fully non-intrusive. Neither the SystemC library that is needed for elaboration, nor the used compiler is modified. Also the behavior and the architecture of the analyzed SystemC model stay untouched. Thus, the methodology proposed is suited for dynamic types which grow unforeseeable at run time. This is an important feature since SoC designs tend to contain software partitions sharing this characteristic.

III. State Extraction

To obtain a complete hierarchy (as mentioned in Section I), this work distinguishes between two different kinds of hierarchy. The first one is a statical one that can be derived by parsing. The second one is a dynamical hierarchy that has to be examined using run time information. To derive the dynamic hierarchy of a model the approach presented in this work extracts its start state. This state is part of each valid SystemC model that can be simulated. The observed system state is defined by a concrete set of values for all variables, declared in the program. These variables again define the state space of the respective model at start time of its simulation that is reached when the function `sc_start` is called.

A. Architecture

As can be seen in Figure 2 the extraction methodology is partitioned into four phases. First the syntactical analysis derives a given SystemC program and generates a parse tree that holds the static architecture of the input program. The parser and an additional scanner have been developed for this application and support special features, like cross probing. Both tools have been implemented using the *Purdue Compiler Construction Toolkit* (PCCTS) [11].

The parse tree is also called *Abstract Syntax Tree* (AST) and is used for communication among the different phases. But since our elaboration only allows SystemC programs as input, we implemented the inverse function of the parser. The AST-synthesis generates a SystemC program from its parse tree to establish communication between instrumentation and elaboration.

In order to obtain the result of the elaboration in the form of an AST, the model that is to be elaborated has to be annotated first. Afterwards the set of automatically generated functions that are annotated to the model implement our elaboration. The generation of those functions is realized by the instrumentation.

The outcome of the finally applied state extraction is the unrolling of all generic nodes inside the AST, i.e. to derive concrete values from all expressions. In contrast to the SystemC data introspection our technique expands the AST that has been computed by the parser before, instead of only storing values of variables in the structures of the kernel.

B. AST-Synthesis

The AST represents an acyclic graph. Consisting of six different node types it implements a possibility to express any arbitrary combination of control flow operation and data dependency. And because all kinds of declarations that are allowed in C++ can be represented by the AST, also state spaces can be represented in an AST graph.

While the semantic of the program is directly transferred to the AST, each manipulation of the AST will have an influence on the corresponding SystemC document that is a

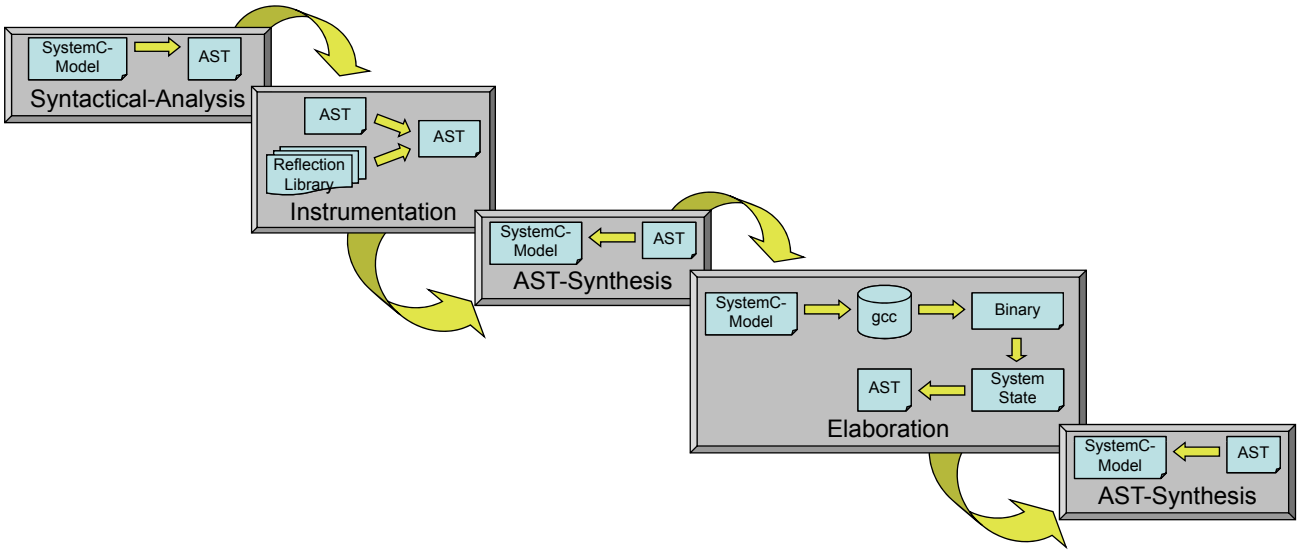

Fig. 2. Architecture of the approach

result of the AST-synthesis. Thus, with a fair amount of effort each operation in the program can be captured during run time by just adding an additional statement to the AST. The synthesis of an AST is much less complex than the syntactical analysis. While the analysis frequently faces the problem of ambiguities when deriving a parse tree, the generation of a program from its AST is unambiguous. To be able to support cross probing facilities as described in Section I, tokens that form the AST, integrate additional information besides line numbers. Among the additional information are byte positions and corresponding file names of the token context. With this information the generated code of the AST-synthesis is written to files that use the same names as the input files of the syntactical analysis.

C. Instrumentation

The instrumentation of the source code is a generation of a set of functions. These functions again will be executed in a binary that has been compiled from the output files of the AST-synthesis. These functions are called *recorder functions* in the following. Each recorder function causes the recording of a state variable after a change of its value. Therefore the recorder function stores each modified value in the AST of the corresponding variable during elaboration.

All user defined data types are compounds of primitive types e.g. `int` and pointers. Hence, only entities that are declared as native types or pointers are going to be recorded when changing their value. To store value changes of variables without side effects in the AST, the propagation of the value changes happens directly after computing the respective expression but before the elaboration of adjacent expressions.

In case the stack frame changes during elaboration (e.g. when passing function calls, statement blocks or overloaded operators), also the stack frame of the corresponding AST has to change. Thus, also more recorder functions have to be generated, that enlarge or shrink the respective AST when entering or leaving a statement block.

To be able to expand an AST of a SystemC program during run time, an inclusion directive is generated into the AST that includes the source code of our syntactical analysis. Additionally the AST is extended by a sequence of instructions. This sequence causes the statical (syntactical) analysis of the model during elaboration. So before elaboration starts an exact copy of the AST that was also used as input of the instrumentation phase is handed over to the simulator. A simplified representation of the combination between statical and dynamical analysis with the help of instrumentation can be seen in Figure 3.

D. State Elaboration

Instead of elaboration by pure interpretation, like done in [6], our approach follows a hybrid strategy. After the static analysis has finished, a simulative analysis is applied for the purpose of elaboration. Compared to an interpreting elaboration an execution has the advantage of being able to

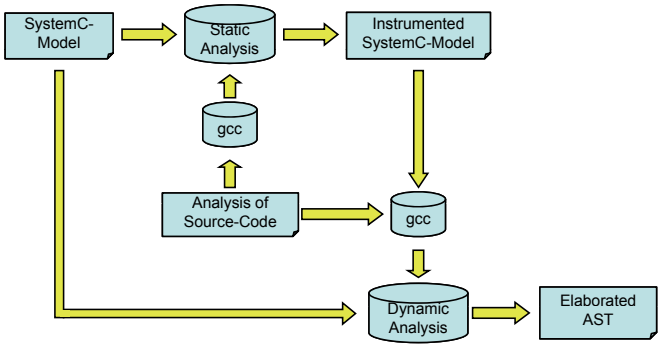

Fig. 3. Hybrid analysis

```
1  struct Euclid : public sc_module
2  {
3    sc_in<unsigned int>  portA;
4    sc_in<unsigned int>  portB;
5    sc_out<unsigned int> portC;
6    unsigned int         valA;
7    unsigned int         valB;
8
9  void calc ()
10 {
11   valA = portA.read();
12   valB = portB.read();
13   while ( valA & valB )
14     max(valA,valB) -= min(valA,valB);
15
16   portC.write( valA );
17 }
18
19 SC_CTOR ( Euclid )
20 {
21   SC_METHOD( calc );
22   sensitive << portA << portB;
23 }
24 };
```

Fig. 4. Euclidean algorithm

elaborate expressions, without knowing the respective source code. This becomes important when using system calls or when linking the program to external libraries, which are common techniques in system designs.

However, the elaboration and by this the state space extraction is limited to variables and functions that are declared inside the analyzed model. Other declarations i.e. identifiers that have been declared in an external library exclusively, do not appear in the resulting AST. But only known entities that do occur in the AST can be annotated with recorder functions. And finally only nodes that are attached to recorder functions can be expanded to values. All values of elaborated state variables are written to the AST automatically because the instrumentation code for those variables is compiled automatically, too. Hence, after elaboration the AST becomes a tree whose leaves comply to one of the following:

- constant values,
- state variables,
- undefined functions or
- undefined variables.

Consequently all control- and data operations of the AST are placed between the root and the leaves. Variables, which describe the state space, can be traversed in depth first search order after elaboration to be extracted.

IV. Example

To demonstrate a practical run of our analysis concept, we applied the respective implementation to SystemC models. As an example in the following we report the results for a model that computes the Euclidean algorithm within a combinational process (see Figure 4).

Only the ports are declared using SystemC types here. The computation itself is done on two integer variables. The loop that calculates the greatest common devisor (line 13) is not a SystemC construct as well. And note that the Functions `min`/`max` are user defined. Thus the interface of the module is clear to SCV. But considering the behavior data introspection can only observe a black box without manual annotations.

After parsing our approach considers `valA` and `valB` as part of the architecture. Running elaboration phase attached SystemC applications (e.g. [6]) are not only aware of the changing values in signals. They do have a direct knowledge of the control sequence that caused this change of the value.

V. Summary

In this paper we presented an analysis concept to extend SystemC models with non-intrusive reflexion capabilities. Our approach facilitates the state extraction of SystemC programs without being limited regarding the abstraction level of the model. Advantages over pure simulative or statical analysis techniques as [5], [6] have been shown, as well as the necessity for a combination of statical and dynamical strategies.

Future work in this area will concentrate on extending this approach with syntactical control mechanisms for the instruction phase. By this, arbitrary modifications can be applied to different models in an automatic way that is less error prone and time consuming than the manual insertion of macros to insert additional operations into SystemC models.

References

[1] *IEEE Standard SystemC Language Reference Manual*, IEEE Computer Society, 2006.

[2] G. Fey, D. Große, T. Cassens, C. Genz, T. Warode, and R. Drechsler, "ParSyC: An efficient SystemC parser," in *Workshop on Synthesis And System Integration of Mixed Information technologies (SASIMI)*, 2004, pp. 148–154.

[3] M. Moy, F. Maraninchi, and L. Maillet-Contoz, "PINAPA: An extraction tool for SystemC descriptions of systems-on-a-chip," in *ACM international conference on Embedded software*, 2005, pp. 317–324.

[4] D. Große, R. Ebendt, and R. Drechsler, "Improvements for constraint solving in the SystemC Verification Library," in *Great Lakes Symposium on VLSI*, 2007, pp. 493–496.

[5] F. Rogin, C. Genz, R. Drechsler, and S. Rülke, "An Integrated SystemC Debugging Environment," in *Embedded Systems Specification and Design Languages: Selected contributions from FDL'07*, E. Villar, Ed. Springer-Verlag, 2008, pp. 59–71.

[6] C. Genz, R. Drechsler, G. Angst, and L. Linhard, "Visualization of SystemC Designs," in *IEEE International Symposium on Circuits and Systems*, 2007, pp. 413–416.

[7] C. Genz and R. Drechsler, "System exploration of SystemC designs," in *IEEE Annual Symposium on VLSI*, 2006, pp. 335–340.

[8] D. Große, R. Drechsler, L. Linhard, and G. Angst, "Efficient automatic visualization of SystemC designs," in *Forum on Specification and Design Languages*, 2003, pp. 646–657.

[9] T. Schubert and W. Nebel, "The quiny SystemC frontend: Self-synthesizing designs," in *Forum on Specification and Design Languages*, Sep. 2006, pp. 317–324.

[10] *KaSCpar*, FZI Karlsruhe, http://www.fzi.de/sim/kascpar.html.

[11] T. Parr, *Language Translation using PCCTS and C++: A Reference Guide*. Automata Publishing Company, 1997.