System Exploration of SystemC Designs

Christian Genz Rolf Drechsler

Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
{genz,drechsle}@informatik.uni-bremen.de

Abstract

Due to increasing design complexity new methodologies for system modeling have been established in VLSI CAD. The SystemC methodology gains a significant reduction of design cycles by introducing an executable specification and a top down refinement strategy. But still the size and the complexity of SystemC models grow, making it harder to understand the basic ideas architects and their designs intend. This extends the familiarization phase for coworkers and project partners. In modern design flows, this can become a significant problem.

In this work we present an approach for interactive system exploration of SystemC designs and its implementation. The aim of our approach is to facilitate the orientation towards complex SystemC models without the need for simulation based techniques. Our tool accomplishes system exploration by allowing to navigate hierarchically through SystemC designs. It uses schematic visualization at different levels of abstraction to display the structure and the behavior of the design. Further support is given for different schematic views, a source code view, crossprobing, path fragment navigation and module exploration.

1 Introduction

The system description language SystemC has become a standard in system level design. SystemC's ability to reach higher abstraction levels than *hardware description languages* (HDLs), while still being able to represent HW-structures and the pragmatic approach of implementing executable C++ specifications make SystemC attractive for industry and academia. The methodology introduced by SystemC aims to close the design gap, which implies a reduction of time to market.

Since SystemC has been introduced, the *electronic design automation* (EDA) community heads for extensive tool and library support. In conjunction with attached libraries SystemC is capable of functional simulation, simulation based verification [10] (SCV library), transaction level modeling [2] (TLM library) and modeling of analog and mixed-signals [18] (AMS library). Among other tasks current tools support waveform tracing [15], synthesis [7], co-simulation [3], bounded model checking [8] and analysis [11, 1, 16]. In contrast to the libraries and tools supporting SystemC verification and analysis, the number of tools supporting system exploration of SystemC specifications is disproportionately smaller. Caused by this lack of tool support SystemC potentials are not fully exploited. Related work is discussed after the presentation of our approach in Section 6.

In this paper we introduce the tool *ViSyC*, that provides functionality for system exploration of SystemC models. In the following the term *system exploration* will be used as a concept that helps architects elaborating system designs. Simulation based attempts, as they are frequently used in design space exploration, assume an extended knowledge of the design for being able to interpret the results of elaboration. Unlike to these techniques system exploration only requires an essential understanding of VLSI CAD to explore the design. The advantages of our approach are:

- hierarchical visualization
- crossprobing
- path fragment navigation
- module exploration

Our tool supports the understanding of large *system on chip* (SoC) designs. They can be represented at different levels of abstraction via detail hiding, that encapsulates implementation details into three different schematic views. Thus, the internal structure, the behavior and the interface of a module are separated to simplify the representation of each single view. All three views preserve the hierarchy information that is given by the structure of the SystemC model. All elements of the schematic view are linked with a corresponding position in the source code view via bidirectional crossprobing, enabling intuitive path fragment navigation. The path fragment navigation is a technique that enables tracing of signals, ports, and operational elements by following their connected inputs or outputs. The functionality of the tool covers SystemC analysis and the generation of a database for visualization. Hence, the analysis is very complex, it is separated into two phases. The first phase is a transformation from an *abstract syntax tree* (AST) to an *intermediate representation* (IR). The second analysis phase is an interpretation of the function `sc_main`. The interpretation allocates internal memory images of the instantiated modules and builds up connectivity between these instances. We use the tool GateVision from Concept Engineering[1] as a backend for graphical system exploration. It

[1] www.concept.de

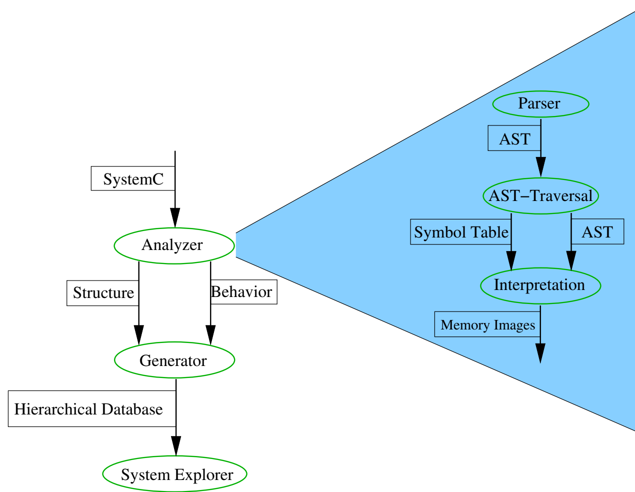

Figure 1. Overall flow within the tool

enables the designer to navigate through the database interactively. The overall flow is shown in Figure 1.

The paper is structured as follows: Section 2 introduces the reader to the standards of SystemC and its methodology. The way we analyze SystemC models is described in Section 3. Section 4 describes the background of visualization and the bidirectional connection between the source code and its schematic counterpart. In Section 5 our tool is applied to evaluate a system level description by the exploration of selected modules. Section 6 clarifies the alternatives to our approach, and discusses related work. It gives direct comparison to existing approaches and also discusses limitations. In Section 7 a brief summary of the paper is given and directions of future work are discussed.

2 SystemC

SystemC has been published by the *Open SystemC Initiative* (OSCI). OSCI is a non-profit association that has been formed by several industrial, academical and individual partners. The aim of OSCI is the standardization of SystemC as an open source standard for system level design. Since the SystemC library is open source, various kinds of modifications and extension libraries are publicly available, too [2, 18, 10, 14].

The system description language SystemC provides hardware constructs, implemented in a C++ class library. The hardware models specified using SystemC can be compiled on a large number of supported architectures using a standard C++ compiler. The compiled executables can be cycle accurate simulations as well as untimed algorithmic descriptions of the given design. The executable specifications can be used for evaluation, debugging and refinement purposes without the usage of a commercial simulator. Depending on the abstraction level the simulation speed can be a multiple of a functional equivalent HDL model. Because of its unrestricted C++ conformance each SystemC model can be combined with other software libraries. This allows system engineers to take advantage of HW/SW Co-Design and to refine their SoC designs with a high level of flexibility. Another benefit of SystemC, coming with its C++ conformance, is a wide range of abstraction levels that can be used to simplify huge system designs. Complex communication protocols and control logic can easily be separated from functional parts of the specification. For this reason SystemC offers techniques that can raise or lower the level of abstraction. The TLM library implements such a technique to support SystemC's efficient refinement methodology. For more details see [12].

SystemC combines HDL typical features, like concurrency as it appears in hardware, with software paradigms, like object orientation. Those features distinguish SystemC from VHDL, Verilog and SystemVerilog and enable system description capabilities. SystemC allows real polymorphism which includes the application of arbitrary memory access using pointers and dynamic memory allocation. Even the concept of virtual functions that binds overloaded class members to function pointers, is applicable in system descriptions. Special benefits, like channels, make SystemC ideal for describing complex communication protocols and their easy reuse.

3 SystemC Analysis

SystemC analysis aims to transform a specification into an abstract representation. Recent efforts in SystemC analysis follow two approaches to evaluation: simulation and parsing. The simulation approach has the severe drawback that it does not simultaneously cover all paths of the *control flow diagram* (CFD). For this reason we implemented an analyzer that is able to analyze SystemC programs without simulation. The analyzer maps relevant information to an IR that contains the structure and the behavior of the model. The analyzer can be split into syntactical analysis, which is done by the parser, and semantical analysis. Hence, the semantical analysis is quite complex, it is split again into AST traversal and a following interpretation phase. The AST traversal collects the structure of the program while the interpretation phase collects the behavior and dynamic objects.

3.1 Extraction of Syntactical Information

The parser is realized by use of PCCTS [13]. The input is a standard C++ source file, implementing a SystemC model. The output of the syntactical analysis is an AST, where an example can be seen in Figure 2. The AST is an acyclic directed graph. Hierarchy information is stored in the nodes of the AST. Except for root and leaf nodes, each node in the AST has exactly one parent and one or two children. Each down edge references a leaf, or an adjacent node, that has further details concerning the parent node. Each right edge also references a leaf, a following statement or a following declaration.

Besides referencing its children, each node of the AST refers to an appropriate token structure. The token structure includes a token type, a token value and further details of the read word. The advantage of the AST is to have a datatype, that enables the analyzer to split the

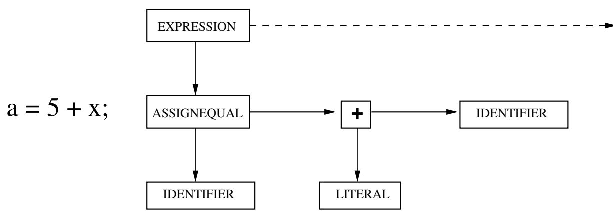

Figure 2. AST example

analysis into several passes. Each AST node holds location information including the source file, the line of code and the character position of the underlying token. These are very important details in the process of visualization, that builds connections between symbolic objects and their corresponding source code fragments.

Most SystemC models use preprocessor directives to preserve a clear design style and to reuse existing structures and algorithms. Unfortunately common preprocessor frontends [17] manipulate the input stream of the parser. Frequently used directives like `#define` are eliminated at its point of definition and inserted at its points of usage in the source code. The parser and its backends have no opportunity to identify such substitutions. To avoid these problems we implemented a preprocessor frontend for recognition of preprocessor directives during syntactical analysis. The usage of widespread keywords like `SC_MODULE` is implemented by the grammar, used in the parser, not by preprocessor substitution. The SystemC grammar is an advantage of our approach that avoids many effects, that are caused by C++ workarounds in the SystemC library. One example of these workarounds in SystemC is the usage of internal module names, which are not necessarily equal to the declaration names. Another example is the limitation of parameters, that can be used in named port bindings.

3.2 Extracting Structural Information

With the completion of syntactical analysis the corresponding AST is available. To come to knowledge about the structure of the model we traverse this AST to collect details of the types and the modules, that are used in the design. During the AST traversal all occurrences of named declarations like types, functions and variables are stored in a symbol table. The symbol table is a hierarchical container that offers efficient functionality for searching named declarations and error checking. All elements hold a reference to their corresponding sub-AST to enable crossprobing later on. Initially the symbol table consists of an empty scope, which gets filled up during AST traversal, and common SystemC data types like `sc_int<n>`. To avoid multiple declarations of the same type preprocessor directives including the SystemC header files can be ignored.

Hierarchy in the symbol table is given by scopes, that emerge when traversing the AST of a function, a type declaration (`struct/class/module`) or a block statement. Each level of the hierarchy is associated to a private name-space with a disjoint set of variables, types and functions. A SystemC evaluation approach based on execution is aware of everything but of declared names, since objects are referenced by their address through compilation. To provide object names for analysis such an approach is forced to attach additional name information, organized in a single namespace. Because no element of this namespace has knowledge of the parent scope, hierarchy information is lost. The deficit of hierarchy causes all equally named objects of different scopes to be misleadingly identified as the same object during evaluation. Another problem caused by introducing a single namespace is the disability to ensure the equivalence between declaration name and additional name of an object, which may lead to confusing effects. With scopes the symbol table prohibits the reuse of equal names at the same hierarchy level and reuse is allowed at an other level only.

The structure of the design is basically formed by the declarations and memory allocations caused by global variables, stack variables, or heap variables. Since SystemC implements C++ programs, the structure of a model can be build using dynamic memory allocation to instantiate modules and signals. Hence, dynamic modules and other SystemC objects that are allocated using the `new` operator, are not passed to the symbol table. Because our approach does not rely on simulation, we face this problem with an additional phase that interprets the specification partially as explained in Section 3.3.

3.3 Extracting Behavioral Information

The behavior of a SystemC model is given by statements that are placed in function definitions. Each function definition in the symbol table references an AST that holds such a statement block. The behavioral analysis starts at `sc_main`. Each statement of the function is analyzed and interpreted. As the top level routine which defines the starting position of the model, the `sc_main` function allocates top modules, signals and builds up connectivity.

Instances of static and dynamic variables are stored to a memory image. Each variable in the memory image has a value that can be read or written by the operational AST that is interpreted. Like other variables, that are typed as a struct or a class, modules have a constructor. The module initialization includes the interpretation of the constructor AST. This is where the process type is set and where the function to be executed is called. The block of this function is traversed recursively and stored into the memory image as a sequence of operations. At the end of this procedure all modules and their ports are known and interconnected by a continuous sequence of expressions.

4 SystemC Visualization

From the computed memory images the tool generates a binary file that holds a database. The generation of this database runs a mapping mechanism that converts SystemC objects to viewable symbols, dependent on the class the objects are derived from. Because SystemC follows an object oriented paradigm it is able to define an arbitrary number of derived types. Since the number of viewable symbol classes is restricted by the exploration backend,

different types have to be grouped and mapped to a single symbol. The database can be displayed with an interactive GUI for design exploration from Concept Engineering.

Besides a schematic view, that uses the symbols of the database, the GUI offers a source code view. Each symbol in the schematic view corresponds to a passage of the source code view. *ViSyC* implements a bijective function that does a mapping between symbols and their corresponding source code passages. The GUI uses a crossprobing technique to implement arbitrary navigation between symbols, their declaration and their instantiation. Another technique that benefits from the bijective map is the path fragment navigation. The path fragment navigation uses the link between corresponding symbols and source code passages and enables system designers to follow data paths or control flow paths intuitively.

In order to get a short and compact representation of the SystemC specification, we need to extract the whole hierarchy. Once having this hierarchy, the circuit can be described at various levels of detail. It is important to note that all hierarchy information is given by the design and not generated by our tool. By using the extracted hierarchy without modifications *ViSyC* conserves the semantic equivalence between the source model and its output, the schematic view. The semantic equivalence again is a premise for crossprobing, that ensures each object of the model to have a counterpart in the symbolic view and vice versa.

The visualization of the hierarchy follows the different scopes of the design. Only the top level view shows all modules connected to their signals. The black box of a module or a channel is its most abstract symbolic representation and can be explored. By entering the module all members including ports and submodules are shown including their connections. The connections between ports are established by sequences of operational symbols, that have been extracted from the processes behavior. Loop statements and conditional statements are also represented by black boxes including conditional information. Statements that consist of one or more expressions are mapped to a sequence of symbols. This way arithmetical expressions and logical expressions can be displayed in a continuous flow. An expression that calls a function is mapped to a black box that is explorable.

Mapping signals to viewable symbols is done by a mechanism that creates single wires or buses according to the signal's width. Each wire or bus can be connected to an arbitrary number of ports. When a channel has been chosen instead of a bit signal, the symbol is a black box again. The exploration of the black box shows the implementation of the channel. Standard C++ or SystemC data types are not explored, because these basic types are known.

5 Evaluation of a RISC-CPU

To demonstrate the different abstraction levels of our approach, we perform system exploration on the SystemC source code of a RISC-CPU. The source code is part of the OSCI SystemC package and it is freely available [12]. The RISC-CPU consists of twenty eight source files, that define the ten modules. The SystemC model is implemented using *register transfer level* (RTL). For evaluation of the model we navigate through three different abstraction levels that clarify the structure and the behavior of the model.

5.1 Top View

The top view of a SystemC description assembles the modules that are instantiated globally or in `sc_main` and their connecting signals. The top level of the RISC-CPU can be seen in Figure 3. All modules, as well as other SystemC objects, are annotated with their declaration name. The internal SystemC name, that is attached to modules, ports and signals is not displayed. The application window in Figure 3 is separated into two fields. The left field of the window displays a hierarchical list that enumerates all instantiated modules. All entries of the list give access to a source code passage that declares the type of the corresponding module. The right field of the window displays the schematic view. The objects of this view are linked to their instantiation in the source code.

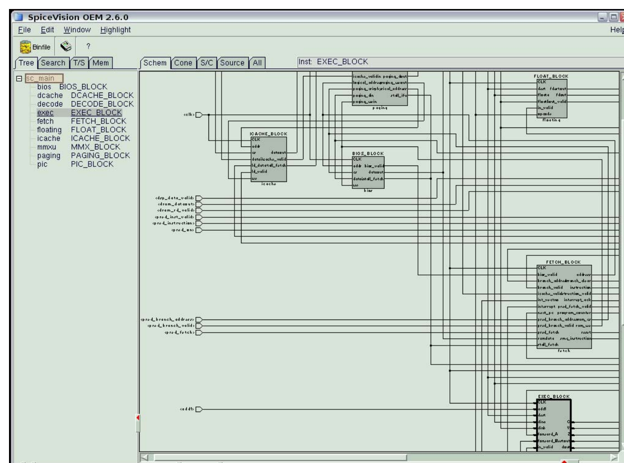

Figure 3. Top view of the RISC-CPU

5.2 Module View

To get a more detailed impression of the structure it is possible to explore single modules. The `exec` module of the CPU is an operational unit that keeps the instruction set. The input ports of the module are aligned to the left whereas the output ports are aligned to the right in alphabetic order. Between input and output ports the behavior of the module is displayed by schematic symbols. The process of the `exec` module is expanded by the entry function that is handled as an `SC_CTHREAD`, shown in Figure 4. Basically the entry function implements a selection statement that is sensitive to the opcode signal. The selection statement is represented by its functional blocks and a set of multiplexers. By use of the opcode the multiplexers are able to assign a calculated value of the respective functional block to its corresponding output port.

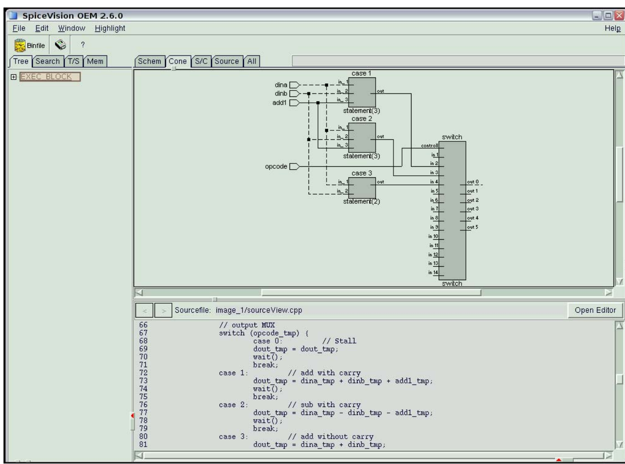

Figure 4. Module view of the RISC-CPU

5.3 Word Level View

The word level consists of a sequence of logic and algorithmic operations. It represents the behavior level of the CPU. This view can use logic and algorithmic operators only. Because the symbolic mapping depends on the hardware implementation of these operators and is not given in the system description, they are not explorable like modules. Figure 5 shows the case 0 of the selection statement that implements an arithmetic addition with carry-bit.

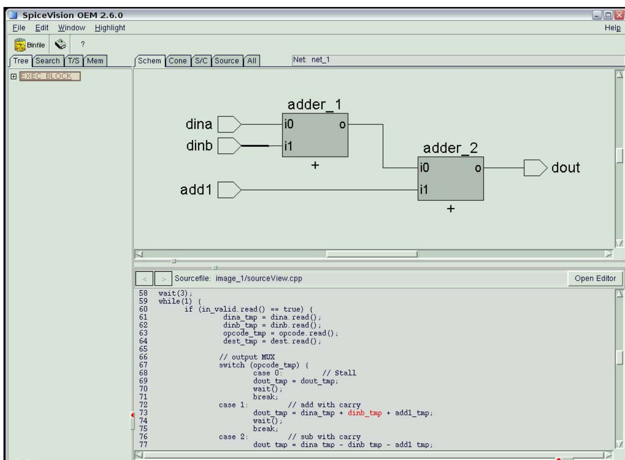

Figure 5. Behavior of the RISC-CPU

6 Discussion

Few approaches have been presented in the field of SystemC analysis and visualization. These approaches differ from ours in the degree of detail of the visual representation and in the type of analysis. None of them accomplishes system exploration, as it is described in Section 1. This section discusses the related work to show the advantages and restrictions of our approach.

6.1 Related Work

One of the first approaches that accomplishes SystemC design visualization has been introduced in [9]. The implementation of Große et al uses the SystemC kernel to analyze SystemC models during execution. An interactive graphical backend facilitates the design visualization. Even though models can be specified using the C++ features, analysis and visualization are limited to SystemC objects. Only the data flow can be viewed, no behavioral information is available. Since this approach has to execute the model without further information of declarations, it is not aware of detailed positional information regarding the objects. Hence, crossprobing facilities are very restricted.

The tool gSysC [5], an extension library for SystemC, is a recent work at the University at Lübeck. When using gSysC instead of SystemC, the underlying application is able to capture run time information of the executed specification analogously to [9]. Additional functionality of gSysC offers the graphical evaluation of state variables and the visualization of a flat module list. Similar to [9] this approach uses run time evaluation, but the hierarchy information is completely lost.

ParSyC [7] is a synthesis tool that does RTL synthesis on a SystemC subset. The subset is a combination of restricted C++, essential SystemC data types and SystemC control mechanisms. The output of the synthesis is unoptimized BLIF (as used in [6]). Instead of run time evaluation this approach uses a separate analyzer to extract structure and behavior of the source model. To reduce the complexity of input models ParSyC prohibits features like pointers, polymorphism and the declaration of templates. Hence, ParSyC does not achieve our requirements regarding SystemC analysis.

Another approach that includes an extra SystemC parser has been published by Snyder and is called SystemPerl [16]. SystemPerl is a Perl library that summarizes four major packages with SystemC support. The parser extracts the netlist interconnectivity and other information from an unpreprocessed file. But SystemPerl is not sufficient for our requirement of an analyzer frontend because it does not extract code from the body of a process. Like SystemPerl also SystemCXML [1] is a free tool with SystemC analysis capabilities. An interpretation of the SystemC source is done by Doxygen [19], that generates a respective XML representation. The XML files are used by SystemCXML to capture structural information of the model into an IR. Unfortunately the analysis is not detailed enough to capture the behavioral information of the model.

Pinapa [11] implements a free SystemC analyzer built upon a modification of the GNU C++ compiler and a modified SystemC library. Pinapa enables a fine grained analysis by executing the model in combination with an AST traversal procedure. Being based on execution, Pinapas analysis covers only parts of the design that are accessed in execution. Thus the quality of analysis depends on the reliability of a stimuli generator.

6.2 Discussion of our Approach

Besides reliably interpreting SystemC descriptions, instead of simulation, our approach accepts SystemC as a language, which enables us to ignore some of its workarounds (see Section 3.1). The advantage of our interpretation over simulation approaches is that we are not dependent on a stimuli generator for an analysis that covers the complete model. Our analysis is even able to handle system descriptions without a stimuli generator. The backend of our analysis decides whether a part of the description is part of the model. With this information our visualization backend is able to hide unnecessary details that would only confuse the designer.

The lack of all tools listed in Section 6.1 and ours as well is that they are bound to one single SystemC version, or to a modification. Without the inclusion of the SystemC library ViSyC is not aware of any changes in the future of SystemC. To assure models to run that are build on a future SystemC version, this version has to be compatible to the current release (SystemC-2.1.0). But considering the development of SystemC as a difficult process that consumes much time and effort, this lack may be negligible.

ViSyC enables the exploration of SystemC designs. Visualization is a technique for representing a complex context in a symbolic way. The schematic view, used in our tool, is an aid that lets the user decide which kind of abstraction he/she wants to use for exploring the design. While pure visualization is able to represent one abstraction level only, our system exploration is interactive and allows GUI supported tracing of each symbol in the design. Besides the interaction, a bidirectional correlation between source code and symbols in schematic view is supported. This allows the tracing of code to its connected symbols and vice versa.

7 Conclusion

In this paper we presented an approach for interactive system exploration of SystemC models. Our approach is implemented as the tool *ViSyC*[4], that generates three schematic views from the source code of a model. The schematic views extend the hierarchical structure as well as the behavior of the model and enable system designers to navigate through different abstraction levels. Each part of a schematic view corresponds to a source code passage in the source code view. Via bidirectional crossprobing these connections can be traced intuitively.

Our approach does not rely on stimuli generators or a modified SystemC library and covers the complete model. It supports extensive information extraction from SystemC designs. This makes *ViSyC* a clever extension to SystemC and offers a platform for other development steps. Our future research on *ViSyC* will concentrate on debugging facilities. In detail our next steps will include a schematic run time evaluation that enables *ViSyC* to visualize selected configurations of a running system model.

Acknowledgements

The authors would like to thank Lothar Linhard and Gerhard Angst from Concept Engeneering for the friendly support and the cooperation.

References

[1] D. Berner, H. Patel, D. Mathaikutty, J.-P. Talpin, and S. Shukla. SystemCXML: An extensible SystemC front end using XML. Technical Report 06, FERMAT@Virginia Tech, Apr. 2005.

[2] L. Cai and D. Gajski. Transaction level modeling: an overview. In *IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis (CODES+ISSS)*, pages 19–24, 2003.

[3] CoWare Inc. ConvergenSC. http://www.coware.com.

[4] R. Drechsler, G. Fey, C. Genz, and D. Große. SyCE: An integrated environment for system design in SystemC. In *IEEE International Workshop on Rapid System Prototyping*, pages 258–260, 2005.

[5] C. Eibl, C. Albrecht, and R. Hagenau. gSysC: A graphical front end for SystemC. In *European Conference on Modelling and Simulation*, pages 257–262, 2005. Source available at http://www.iti.uni-luebeck.de/ albrecht/gSysC/.

[6] Electronics Research Laboratory, University of California at Berkeley. *OCTTOOLS-5.2 Part II Reference Manual*, Mar. 1993.

[7] 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)*, pages 148–154, 2004.

[8] D. Große and R. Drechsler. CheckSyC: An efficient property checker for RTL SystemC designs. In *IEEE International Symposium on Circuits and Systems*, pages 4167–4170, 2005.

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

[10] C. Ip and S. Swan. A tutorial introduction on the new SystemC verification standard, 2003. Available at http://www.systemc.org.

[11] 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 (EMSOFT '05)*, pages 317–324, 2005.

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

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

[14] H. Patel and S. Shukla. Towards a heterogeneous simulation kernel for system level models: A SystemC kernel for synchronous data flow models. *IEEE Transactions in Computer-Aided Design*, 24(8):248–253, Aug. 2005.

[15] W. Snyder. Dynotrace. Available at http://www.veripool.com/dinotrace/.

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

[17] The GNU Project. The gnu compiler collection. http://www.gcc.gnu.org.

[18] A. Vachoux, C. Grimm, and K. Einwich. SystemC-AMS requirements, design objectives and rationale. In *Design, Automation and Test in Europe*, pages 388–393, 2003.

[19] D. van Heesch. Doxygen. Available at http://www.doxygen.org.