A Theoretical and Experimental Review of SystemC Front-ends

Kevin Marquet Matthieu Moy Bageshri Karkare

Verimag, Université Joseph Fourier/Grenoble INP, Grenoble, France
first.last@imag.fr

***Abstract*—SystemC is a widely used tool for prototyping Systems-on-a-Chip. Being implemented as a C++ library, a plain C++ compiler is sufficient to compile and simulate a SystemC program. However, a SystemC program needs to be processed by a dedicated tool in order to visualize, formally verify, debug and/or optimize the architecture. In this paper we focus on the tools (called front-ends) used in the initial stages of processing SystemC programs. We describe the challenges in developing SystemC front-ends and present a survey of existing solutions. The limitations and capabilities of these tools are compared for various features of SystemC and intended back-end applications. We present typical examples that front-ends should ideally be able to process, and give theoretical limitations as well as experimental results of existing tools.**

I. Introduction

SystemC is a C++ class library which facilitates modeling of systems at various levels of abstractions ranging from functional description to cycle-accurate modeling. The ability to design at higher abstraction levels is valuable due to increasing complexity of system design. Being a C++ library, SystemC provides typical high-level language features which make the task of system design easier and faster than lower-level hardware description languages. It also offers the concepts of timing and concurrency which are essential for hardware modeling. Therefore, it has become the *de facto* standard for modeling embedded systems, and has been approved as a standard by the IEEE consortium [1].

Writing SystemC programs is mainly motivated by the need to obtain the hardware platform earlier, allowing early software development. SystemC was primarily used for simulation, allowing validation of platforms functionality, and embedded software development. A typical C++ compiler suffices to generate an executable that performs simulation. However, as the complexity of embedded software and model have grown, the need has appeared for various other applications (like formal verification, visualization, ...), that require not only an execution of the platform, but also an access to its architecture and an abstract representation of its source code like an Abstract Syntax Tree. Typical applications of a SystemC front-end are visualization, introspection, optimization of simulation, verification and synthesis, as detailed below. For such applications, initial processing of a SystemC program is done by *front-end* tools. Depending on the target application, three issues have to be addressed by a SystemC front-end. First, retrieving the architecture of the platform. In a SystemC simulation, this architecture is obtained at run-time, by executing C++ statements that create and link SystemC components. This phase is called the *elaboration*. In figure 1, the `sc_main` function gives an example of such statements. Second, extracting the control structure of processes and recognizing SystemC specific constructs (the `process` function in the example). At last, making the link between the two, recognizing communications between components in the control structure.

```
SC_MODULE(Component) {
  sc_core::sc_out<bool> out;
  sc_core::sc_in<bool> in;
  bool isHead;

  void process() {
    /* Dynamic behavior of the process.
       May use in.read(), out.write() and
       arbitrary C++ statements ...*/
  }
  SC_CTOR(Component) {  // constructor
    SC_THREAD(process); // process declaration
  }
};

int sc_main (int argc, char *argv[]) {
  sc_signal<bool> s1("s1"), s2("s2"), s3("s3");
  Component C1("C1"), C2("C2"), C3("C3");

  C1.out(s1); C2.out(s2); C3.out(s3);
  C1.in(s3);  C2.in(s1);  C3.in(s2);
  C1.isHead = true;

  sc_start(200, SC_NS); // start simulation
  return 0;
}
```

Fig. 1. Elab-only: a simple example

This paper recalls possible applications of SystemC front-ends (sections II), and the challenges in developing such tools (section III). The main contribution is a theoretical (section IV) and experimental (section V) comparison of existing approaches and tools. We conclude in section VI.

The goal of this comparative study is twofold: first, a detailed survey should help future authors of tools requiring a front-end to pick the right tool. To the best of our knowledge, this paper is the most comprehensive survey in this area, both regarding the number of tools we compare,

and the level of detail of the analysis: we actually installed and tested all the tools we mention when they are available. We are mainly interested in tools usable to build research products, and therefore focus on front-ends which are distributed, with friendly licensing policy. Second, comparing the advantages and limitations of existing approaches should help building the next generation of SystemC front-ends. Besides, part of the conclusion of our study is that all the tools available today have drawbacks, and we are working on a new one which should overcome most of them.

II. Applications of a SystemC front-end

The main possible uses of a SystemC front-end are:

Visualization: GUI tools for system design provide graphical visualization of the architecture and may also provide a facility to edit the design. For graphical visualization, it is sufficient to run the elaboration and display the architecture. A full front-end can be required for more advanced features like source code browsing or graphical editing facility.

Introspection: Introspection is the process of extracting meta-data, i.e. the structural and run-time characteristics of a system. Structural data may include the module names, hierarchy information etc, while the runtime data includes dynamic information such as the number of process invocations, event generations etc. Using data introspection, a piece of code within the platform can access information about the objects containing the platform during simulation, which can ease design space exploration. Examples of tools for SystemC introspection include ReSP [2] and [3].

Optimization of simulation: Once the elaboration phase completed, the architecture of the system is known. This knowledge can be used to perform some optimization. For instance, speeding up the simulation with static scheduling [4], [5], or parallelizing it [6].

Verification: Typically, verification of hardware systems is performed using dynamic validation techniques such as simulation. In order to perform formal verification of the model using existing tools (such as model checkers), the SystemC front-end is required to extract both a detailed representation of process bodies, and the architecture (typically, one cannot reason formally about a statement like `port.write(...);` unless knowing which interface the port is bound to). Most SystemC formal verifiers read SystemC code with a front-end, and formalize its semantics to produce the input language of a model-checker [7], [8].

Synthesis: Some SystemC models can be synthesized into gate-level descriptions. A synthesizer for SystemC works like a compiler, and obviously requires exhaustive information about the source code. Examples of academic synthesizers are [9], [10].

III. Issues In Developing SystemC Front-ends

SystemC being a C++ library, a C++ front-end can parse a SystemC program, but for most applications other than basic simulation, this is insufficient to build a front-end. We now detail the issues of designing a SystemC front-end. First, retrieve the architecture of the system, second represent the C++ control structure in an abstract way, and the most difficult part: link those two structures.

A. Architecture

A SystemC system first defines an *architecture*, i.e. a set of components and connections between them. Components have a behavior defined by one or several processes and communicate with each other through ports. SystemC provides synchronization mechanisms like events and signals.

A SystemC program describes all these concepts using C++ objects declared in the SystemC library, and instantiated during the elaboration phase (i.e. at the beginning of execution). The first goal of a SystemC front-end is to retrieve this architecture. A regular C++ parser can only parse the program and generate an intermediate representation of the code. However, it cannot capture the meaning of the program in terms of components and interconnections, since this information is built at runtime. Such information is vital in some back-ends such as verification tools. Consider a code fragment in Figure 1, where three modules form a circular chain communicate through the use of boolean signals. A C++ parser can give a representation of the elaboration code, but not does not compute this information directly.

There are mainly two approaches to get the architecture: either *execute* the elaboration phase, and get the result of this execution, or *parse* it, and infer the result without executing it. The latter is usually done with static analysis of the elaboration code to determine the interconnections between modules, and would ideally require a full C++ interpreter.

Figure 2 illustrates a more complex case where components are instantiated in a loop and, consequently, the number of processes depends on a variable.

```
int sc_main(int argc, char **argv) {
   module1 *m[MAX];
   for(i = 0; i < MAX; ++i) {
      m[i] = new module1();
   }
}
```

Fig. 2. Architecture dependent on dynamic information

B. Dealing with C++

A SystemC program contain different processes; a front-end allows to build an *Intermediate Representation* (IR) from these processes. Various kinds of IR can be imagined depending on the goal of the tool, but it needs to contain both SystemC-specific treatments and others. SystemC being a C++ library, all C++ features must be supported by the front-end. This can be achieved by either writing a parser from

scratch using the language grammar (which is non-trivial in the case of C++), or by using an existing C++ front-end such as GCC, LLVM, EDG etc. The IR representing SystemC programs can be of various forms, typically an Abstract Syntax Tree or a Control Flow Graph. The choice of the IR can be important, as it must contain enough information to serve the purpose of the back-end.

The SystemC library defines different functions allowing synchronization and communication between processes. Through these operations, SystemC processes can wait a given time (`wait(t)`), write through communication channels (`port.write(data)`), wait and notify events (`wait(e)`/`e.notify()`). A SystemC front-end must be able to detect these operations and mark them as special in the IR.

A SystemC front-end should be able to retrieve these SystemC constructs. As function calls, those specific constructs are easy to detect. A harder task is to capture the semantics: the difficulty here is not only to find those constructs but to establish which components are involved. This requires to make the link between the control structure and the architecture.

C. Linking architecture and control

For instance, if a process performs a `port.write(data)`, it is easy to analyze that a component writes a *data* to *someone*. Determining the target component as well as the data is more difficult, depending on the complexity of the code computing *port* and *data*. Consider the architecture described in figure 2 where components are instantiated in a loop. If a process performs `modules[i].out.write(data)`, an ideal SystemC front-end would be able to retrieve which module is concerned by the writing. This requires that the architecture has been retrieved as well as the control structure (with SystemC special constructs), but also to make the link between them. In the current example, this means evaluating `modules[i].out` which is difficult. In the same manner, the semantics of a `wait(t)` instruction depends on `t`, which can be the result of a complex code.

IV. Existing Tools and Approaches for SystemC Front-ends

We now present existing front-ends, their capabilities with respect to their goals.

A. Tools Using a Dedicated Grammar for SystemC

A first approach to parse SystemC programs is to consider SystemC as a language, therefore rely on a specific grammar considering SystemC classes and methods as keywords. The first drawback of this method is due to the complexity of the C++ grammar and typing rules. As seen in section III-B, a SystemC program may contain any arbitrary C++ code, therefore this approach requires to implement the whole C++ grammar in addition of the SystemC constructs, which is known to require a huge effort. As a consequence, the solutions based on this approach, presented below, are highly incomplete with respect to C++, and require to rewrite SystemC designs with given guidelines. Generally, this excludes some C++ specific features such as dynamic memory allocation, pointers, recursion, loops with variable bounds etc.

A more fundamental issue is that a purely static analysis solution can only handle programs where the architecture is built in a simple way (see section III-A).

- ParSyC: Developed by University of Bremen, ParSyC [9] is a SystemC parser using PCCTS [11] (Purdue Compiler Construction Tool Set); it is a part of an integrated environment for system design called SyCE [12].
  ParSyC generates an Abstract Syntax Tree (AST) which is then converted to an intermediate representation. The intermediate representation is close to the Abstract Syntax Tree but is built using classes corresponding to the constructs in the SystemC code. It acts as a starting point for other tools in SyCE, such as ViSyC which is used for visualization and DeSyC which is used for automatic debugging. The SyCE suite also contains CheckSyC [13] which is a verification tool for formal equivalence checking, property checking and generating checkers for simulation or synthesis.
- KaSCPar: There are two tools in the KaSCPar [14] suite: SC2AST and SC2XML. SC2AST is essentially a C++ front-end with dedicated grammar rules for SystemC constructs. It generates the Abstract Syntax Tree for the given SystemC application. SC2XML first generates the AST using SC2AST and then generates an elaborated description of the application. This elaborated description contains two parts: functional and structural. The functional part contains information about the threads, constructors etc (mainly the control structure). The structural part is obtained by parsing the AST and contains the hierarchically composed system structure including connections between system elements. The suite uses the GNU preprocessor to expand the SystemC macros.
  KaSCPar is a partially open-source tool. The source code for SC2XML, which is the more interesting component of the suite, is not available.
- sc2v: sc2v [10] is an open source tool for automatic translation of SystemC models to synthesizable Verilog code. It is written using lex and yacc tools, and targets only an RTL subset of SystemC.
- A front-end [15] for the Behavioral Synthesizable SystemC subset (BSSC) has been developed with the goal of providing an easily customizable and extendable SystemC parser. BSSC is closer to C than to C++, and the parser supports only basic constructs of C++. Supporting full C++ would require a major effort. Also, although the paper states that the tool is open-source, neither the tool nor the source code is not available.
- SystemC-Perl, or SystemPerl [16] for short, is an extended version of SystemC language facilitating auto-

matic expansion of text to avoid needless repetitions in the code. The entire suite comes with a preprocessor which expands the SystemPerl files into C++ code or stand-alone SystemC code. The suite also contains a netlist extractor which describes the hierarchical interconnections among SystemC modules.
In case of SystemPerl, the user is expected to provide hints in the program for the preprocessor to identify the constructs to be expanded. Netlist extraction does not involve processing of procedure bodies. Thus, SystemPerl is not useful for typical back-end applications.

B. Tools Based on existing C++ front-end

Some works are based on a full, existing C++ front-end, addressing in this way the problem of the complexity of C++ of grammar-based tools:

- Proprietary tools: Semantic Designs use a SystemC front-end which supports full C++ syntax, builds Abstract Syntax Tree and provides facilities to process the syntax tree. Synopsys developed a SystemC front-end which is used in the SystemC compiler CoCentric. It parses the constructors and the main function, as well as the body of the modules with the EDG C++ front-end, and infers the structure of the program from the syntax tree of the constructors. Unfortunately, the tool is not available for download or for evaluation. To the best of our knowledge, the approach they use has not been published.
- SCOOT: SCOOT [5] is a model extractor for SystemC based on a C++ front-end developed by the authors. It includes static scheduling tools, allowing source-to-source optimizations, and integrates verification back-ends to the CBMC model-checker and the SATABS tool. The intermediate format extracted from the SystemC program is basically a CFG annotated with information related to the architecture. Static analysis techniques are used to determine the module hierarchy, sensitivity lists for processes and port bindings. The source code of this tool or the details of the analysis techniques used in the front-end are not available.
- SystemCXML: SystemCXML [17] project aims at retrieving structural information of SystemC models. It uses Doxygen to interpret the SystemC source and generates an intermediate description in XML. This intermediate description called ASLD (Abstract Syntax Language Definition) captures SystemC structural information such as hierarchy, ports, signals, types etc. SystemCXML cannot not derive the architecture information from SystemC programs. The SystemC coverage of this tool is limited by the Doxygen markups used. The output of SystemCXML is useful in back-end applications such as visualization, but the IR used to represent the source code includes only tags meant for pretty-printing the code, not for further analysis.
- Quiny: Unlike the above approaches, Quiny [18] uses an unmodified C++ compiler, but modifies the SystemC library to use the operator overloading feature of C++ to return the expressions instead of evaluating the operation at run-time. Thus, the intermediate code as well as the architecture information is produced by compiling and running the platform.
Since not all language constructs can be overloaded, this requires redefinition of keywords such as `if`, `else` etc. in order to modify their execution-time behavior. Also some C++ operators such as `?:` are not and can not be handled. This limits the usefulness of this tool.

C. Hybrid (Static/Dynamic) Approaches

- Pinapa: Pinapa [19] uses a hybrid approach where on the one hand the SystemC program is parsed using GCC to get the Abstract Syntax Tree and on the other hand the elaboration phase is executed to get the architecture information. Outputs of these two separate phases (i.e. the in-memory data structures) are linked together so that a single output intermediate form can be produced for the intended application such as verification or visualization.
Pinapa can parse any arbitrary SystemC program, but it cannot generate useful output for some constructs such as pointers to SystemC objects or complex array index expressions.
While we think the approach of Pinapa is good, the tool is not easy to use because of technical issues (lack of modularity of GCC in particular) and non-technical ones (license issues prevent the distribution of Pinapa in a compiled form).
- In a recent work [3] by the authors of ParSyC projects, a hybrid technique is briefly presented which uses a PCCTS based parser (supporting a subset of C++) to collect the static information and a code generator to evaluate run time information.

Some tools need only the architecture of the platform, and can use a purely dynamic approach (execute the elaboration and get the result). Such tools are not strictly speaking front-ends and are omitted here and included only in [20] by lack of space. [20] also includes a summary of the front-end applications and theoretical capabilities in the form of a table.

V. Experiments

We now present different test-cases illustrating the challenges SystemC front-end face on as well as the importance of each one. Then we compare the theoretical capabilities of existing front-ends to their effective capabilities. Last, we analyze the results and detail the difficulty, for each tool, to handle new capabilities.

A. Examples

The examples presented here are not supposed to represent an exhaustive set of complex benchmarks for SystemC. They are rather small tests identifying each one a single

problem we ideally want to be handled by a front-end. The complete source code is available from the web [21].

The first example, and most simple, includes the same architecture as the one presented previously in figure 1, without the `SC_THREAD` declaration. This code allows to test the behavior of front-ends on very simple applications and serves as a basis for other examples.

The *elab-easy* uses the same simple architecture (explicit bindings between modules) and contains simple process. The *elab-easy-int* example, is the same application except that data written through signals are integers instead of booleans, testing this different construct. *elab-easy-sc_stop* is a simple example in which a process read data during a given time then calls `sc_stop()`. In order to test yet another constructs, *elab-clock* include a simple process depending on an *sc_clock* to wake up.

In the *elab-easy-array*, components are stored in arrays; the architecture is built by iterating through these arrays.

In the *elab-port-bool* example, ports are stored into arrays rather than components. The *elab-pointer* example accesses the array through indices depending on input values.

In the *elab-instances* example, we access an array of pointer to components, filled in by a simple loop.

The *signal* example tests the creation of modules from another module, rather than in the `sc_main` function. The *event* example (not provided here, but given in [20]) just tests the recognition of events.

At last, we test the *RAM* platform, a bigger example involving a CPU accessing a memory, with the use of clocks, ports and signals.

B. Experimental Results

We experimented existing SystemC front-ends on examples described previously. Results are given in table I. For each tool and each example, this table indicates:

- ✓ if the example could be analyzed.
- ✍ if the example could be analyzed but with (small) adaptation of the test-case. Typically, this is often necessary for grammar-based tools which do not recognize some syntaxes.
- ≈ if it works partially. The concerned case is detailed below.
- `Easily` if the example could not be analyzed, but could be managed with a small implementation work.
- `Doable` if the example could not be analyzed, if this is not a theoretical limitation of the approach, but requires a huge implementation to work. Typically, writing a SystemC front-end handling all C++ and SystemC constructs is an approach that could work but is absolutely not realistic.
- 💣 if the example could not be analyzed, does not seem to be a limitation of the approach, but the case of the error could not be found (in other words, this seems to be a bug in the tool).
- ✘ if the example could not be analyzed and if it is a fundamental limitation of the approach.

ParSyc is not freely available, so we couldn't try it. In addition, the tools *sc2v*, *SystemPerl* require to write applications from scratch and do not really handle the "SystemC language". Those two tools appear in grey in the table, with results corresponding to their theoretical capabilities.

C. Analysis of Results

The main result given by the table is that there does not exist a perfect front-end able to parse any SystemC program, retrieve the architecture and build an intermediate format for that program.

SystemCXML experimental limitations correspond to theoretical ones. It does not give an intermediate representation, does not detect events, requires a strict syntax (use of `struct module (X) : struct sc_module` instead of `SC_MODULE(X)` does not work for instance).

KaSCPar seems to be the most widely used tool amongst those presented, but it is not maintained anymore. We did not receive any answer from the authors to our questions concerning errors we encountered on our tests as well as on the examples they provide. In addition of these disappointing experimental results, the limitations of the approach have been presented above.

Pinapa gives the best results, theoretically as well as experimentally. However, we required the help of the authors to be able to install and execute examples correctly, so the comparison is not completely fair. In the case of *elab-pointer*, Pinapa managed to detect the `write` construct, was unable to determine statically which module is targetted, but decorated the AST corresponding to this access with the correct expression. A few more details:

- the authors corrected one bug in Pinapa to handle *elab-port-bool*;
- Pinapa doesn't follow function calls, but decorate only SystemC constructs that appear directly in the function registered as a process. A *write()* into a function whose call is nested in an other won't be detected.

The hybrid approach followed by Pinapa is interesting as it allows to extract the architecture of any program. However, analyzing the AST given by GCC is a limited approach, as it does not allow to retrieve information in simple C++ constructs dependant on purely static data (an array indexed by an operation on constants for instance).

Although knowing the target of each communication between two modules is undecidable in the general case, we think it is possible to handle most of the cases. Indeed, as it is difficult to accomplish using static analysis solutions, we could *execute* pieces of code computing those information.

Additionally, the intermediate representations given by all available solutions are more or less based on the AST. However, it has been showed [22] that, for verification purpose, the SSA form could give good results.

VI. Conclusion

We presented the motivations and challenges for the design of SystemC front-ends. We detailed existing solutions

	Pinapa	SystemCXML	KaSCPar	Quiny	Scoot	SystemPerl	sc2v
elab-only	✓	✓	✓	✍	✓	✘	✍
elab-easy	✓	✓	💣	✍	✓	✘	Doable
elab-easy-int	✓	✓	💣	Easily	✓	✘	Doable
elab-easy-uint	✓	✓	💣	Easily	✓	✘	Doable
elab-easy-array	✓	✘	💣	Doable	💣	✘	✘
elab-easy-sc_stop	✓	✓	💣	Easily	Easily	✘	Easily
elab-port-bool	✓	✘	💣	Doable	Doable	✘	✘
elab-pointer	≈	✘	✘	Doable	✘	✘	✘
elab-instances	✍	✘	💣	Doable	Easily	✘	✘
elab-clock	Easily	✓	💣	Easily	✍	✘	Easily
signal	✍	✘	✘	✘	✍	✘	✘
event	✓	✓	💣	Doable	✓	✘	Doable
fifo	✘	✘	✘	✘	✘	✘	✘
RAM	Doable	✓	✘	✘	Easily	✘	✘

TABLE I
CAPABILITIES OF SYSTEMC FRONT-ENDS

and their theoretical capabilities and limitations, and gave experimental results. We showed that although the need to analyze SystemC designs is increasing with the complexity of embedded systems, available tools are not able to take as input any arbitrary SystemC models.

Three tools give better results than others. Scoot gives good results, but the source is not open and does not provide its intermediate representation. In addition, the approach is limited because it is completely static. Therefore, it is difficult to use it as a basis for a new tool although it represents an interesting research work. For other purposes, KaSCPar is a good choice for small examples, although it seems to be unmaintained and has, again, the limitations of static tools. It was notably used for verification [23], [7]. Pinapa has the most powerful approach but experience technical difficulty which would require a non neglectable engineering effort. It was initially written for verification [8], and recent work [24] base upon Pinapa to emulate SystemC programs on a FPGA.

We also summarized good ideas in existing solutions and added new ones to overcome remaining limitations of existing approaches. We are currently developing a tool called PinaVM incorporating these ideas, based on the compiling infrastructure LLVM [25].

REFERENCES

[1] "IEEE std 1666 - 2005 IEEE standard SystemC language reference manual," *IEEE Std 1666-2005*, pp. 0_1–423, 2006.

[2] G. Beltrame, C. Bolchini, L. Fossati, A. Miele, and D. Sciuto, "ReSP: a non-intrusive transaction-level reflective MPSoC simulation platform for design space exploration," in *Proceedings of the 2008 conference on Asia and South Pacific design automation*. IEEE Computer Society Press Los Alamitos, CA, USA, 2008, pp. 673–678.

[3] C. Genz and R. Drechsler, "Overcoming limitations of the systemC data introspection," in *DATE*. IEEE, 2009, pp. 590–593.

[4] R. Buchmann, F. Petrot, and A. Greiner, "Fast cycle accurate simulator to simulate event-driven behavior," in *Electrical, Electronic and Computer Engineering, 2004. ICEEC '04. 2004 International Conference on*, Sept. 2004, pp. 35–38.

[5] N. Blanc, D. Kroening, and N. Sharygina, "Scoot: A tool for the analysis of SystemC models," in *TACAS*, 2008, pp. 467–470.

[6] Y. Bouzouzou, "Semantics-preserving parallelization of the SystemC scheduler for reduced simulation times," Master's thesis, UJF, 2007, diplôme de Recherche Technologique.

[7] P. Herber, J. Fellmuth, and S. Glesner, "Model checking SystemC designs using timed automata," in *CODES/ISSS '08: Proceedings of the 6th IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis*. New York, NY, USA: ACM, 2008, pp. 131–136.

[8] M. Moy, F. Maraninchi, and L. Maillet-Contoz, "LusSy: an open tool for the analysis of systems-on-a-chip at the transaction level," *Design Automation for Embedded Systems*, 2006, special issue on SystemC-based systems.

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

[10] J. Castillo, P. Huerta, and J. I. Martinez, "An open-source tool for SystemC to verilog automatic translation," vol. 37, pp. 53–58, 2007.

[11] T. J. Parr, H. G. Dietz, and W. E. Cohen, "Pccts reference manual: version 1.00," *SIGPLAN Not.*, vol. 27, no. 2, pp. 88–165, 1992.

[12] R. Drechsler, G. Fey, C. Genz, and D. Große, "SyCE: An integrated environment for system design in SystemC," *IEEE International Workshop on Rapid System Prototyping,*, vol. 0, pp. 258–260, 2005.

[13] D. Große and R. Drechsler, "Checksyc: an efficient property checker for rtl SystemC designs," in *ISCAS (4)*, 2005, pp. 4167–4170.

[14] "http://www.fzi.de/index.php/de/component/content/article/238-ispe-sim/4350-sim-tools-kascpar-examples."

[15] D. P. Scarpazza, C. Brandolese, L. Pomante, and P. D. Felice, "Parsing systemC: an open-source, easy-to-extend parser," in *IADIS International Conference on Applied Computing*, 2006, pp. 25–28.

[16] "Systemperl," http://www.veripool.org/wiki/systemperl.

[17] D. Berner, J. pierre Talpin, H. Patel, D. A. Mathaikutty, and E. Shukla, "SystemCXML: An extensible SystemC front end using XML," in *Forum on specification and design languages (FDL)*, 2005.

[18] T. Schubert and W. Nebel, "The quiny SystemC front end: Self-synthesising designs." in *FDL*. ECSI, 2006, pp. 135–143.

[19] M. Moy, F. Maraninchi, and L. Maillet-Contoz, "Pinapa: an extraction tool for SystemC descriptions of systems-on-a-chip," in *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*. New York, NY, USA: ACM, 2005, pp. 317–324.

[20] K. Marquet, M. Moy, and B. Karkare, "A theoretical and experimental review of SystemC front-ends," Verimag Research Report, Tech. Rep. TR-2010-4, 2010.

[21] "http://greensocs.sourceforge.net/pinapa/download/files/frontends-testcases.tar.gz."

[22] L. Besnard, T. Gautier, M. Moy, J.-P. Talpin, K. Johnson, and F. Maraninchi, "Automatic translation of C/C++ parallel code into synchronous formalism using an SSA intermediate form," in *Ninth International Workshop on Automated Verification of Critical Systems (AVOCS'09)*. Electronic Communications of the EASST, September 2009.

[23] R. Behjati, H. Sabouri, N. Razavi, and M. Sirjani, "An effective approach for model checking SystemC designs," in *ACSD*, J. Billington, Z. Duan, and M. Koutny, Eds. IEEE, 2008, pp. 56–61.

[24] S. S. Sirowy, B. Miller, and F. Vahid, "Portable SystemC-on-a-chip," in *CODES+ISSS '09: Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis*. New York, NY, USA: ACM, 2009, pp. 21–30.

[25] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *CGO '04: Proceedings of the international symposium on Code generation and optimization*. Washington, DC, USA: IEEE Computer Society, 2004, p. 75.