Formal Techniques for SystemC Verification

Position Paper

Moshe Y. Vardi
Rice University, MS 132
6100 S. Main St.
Houston, TX 77005-1892
vardi@cs.rice.edu

ABSTRACT

SystemC has emerged lately as a de facto, open, industry standard modeling language, enabling a wide range of modeling levels, from RTL to system level. Its increasing acceptance is driven by the increasing complexity of designs, pushing designers to higher and higher levels of abstractions.

While a major goal of SystemC is to enable verification at higher level of abstraction, enabling early exploration of system-level designs, the focus so far has been on traditional dynamic validation techniques. It is fair to see that the development of formal-verification techniques for SystemC models is at its infancy. In spite of intensive recent activity in the development of formal-verification techniques for software, extending such techniques to SystemC is a formidable challenge. The difficulty stems from both the object-oriented nature of SystemC, which is fundamental to its modeling philosophy, and its sophisticated event-driven simulation semantics.

In this position paper we discuss what is needed to develop formal techniques for SystemC verification, augmenting dynamic validation techniques. By formal techniques we refer here to a range of techniques, including assertion-based dynamic validation, symbolic simulation, formal test generation, explicit-state model checking, and symbolic model checking.

Categories and Subject Descriptors

B.5.2 [**Hardware**]: Register-Level Implementation-Design Aids; J.6 [**Computer-Aided Engineering**]: Computer-Aided Design

General Terms

Verification

Keywords

Formal Verification, High-Level Model

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DAC 2007, June 4–8, 2007, San Diego, California, USA.
Copyright 2007 ACM 978-1-59593-627-1/07/0006 ...$5.00.

1. MOTIVATION

A micro-architecture is usually specified by a natural-language document, referred to as the *micro-architectural specification* document (MAS). It is typically accompanied by a micro-architectural simulators for certain parts of the micro-architecture. Over the years, there have been many attempts of limited success in the semiconductor industry to develop high-level models for micro-architectures. For example, the micro-architectural model of the PowerPC 604 was described as "semi-formal" [28], while the analogous model for the Power4 was used only for performance evaluation, but not for validation. One basic reason for the failure is the difficulty of establishing formal equivalence between a micro-architectural model and the RTL, as no technology available today can formally establish such equivalence. What should be done? Simply, develop a micro-architectural model without expecting full formal equivalence to be established between that model and the RTL. Rather, expect "compatibility" ("conformance", "compliance") between the micro-architectural model and the RTL, but not full formal equivalence.

What is the cost of not having a micro-architectural model? In effect, the RTL serves currently as both a micro-architectural model and its implementation. Developing the RTL is hard, since all the RTLers have is the MAS. Validating the RTL is hard, since both the micro-architectural model and its implementation are validated at the same time. When bugs are discovered, it is not immediately clear whether they are micro-architectural bugs or implementation bugs. The lack of a clear "golden" reference model is a serious obstacle to RTL verification. Also, developing the test environment is hard, since it has to test both the micro-architectural model and its implementation.

The micro-architecture is an extremely sophisticated algorithmic model. Validating it is bound to be hard. Implementing it and validating it directly in RTL is bound to be exceptionally hard, slow and expensive. The changing focus in the industry from frequency scaling to functionality enhancements, such as Intel's Virtualization Technology, Hyper-Threading Technology, Extended Memory 64 Technology, and LaGrande Technology increase further the gap between the micro-architecture and the RTL [32]. As G. Singer pointed recently [17], "RTL was established two decades ago. Since then, complexity has grown sevenfold." The right level to develop a 'golden specification model' is at the algorithmic/functional/transaction level. Today's RTL-based validation technology is clearly inadequate to the task

of validating the industry's future highly complex platform-oriented architectures. The motto should change from "design first, then verify" to "specify first, then design and verify'.

Developing a micro-architectural model would separate the micro-architecture from its implementation. The task of creating a micro-architectural model should be assigned to a micro-architectural validation group, working together with the micro-architects. Micro-architectural validators would develop the model concurrently with the development of the MAS, by the micro-architects. Experience has shown that the process of creating a detailed model leads to a deeper understanding of the micro-architecture by the micro-architects and would reveal many bugs. The development of the micro-architectural model concurrently with the MAS may delay somewhat the start of the development of the RTL, but the benefit would be a more mature and transparent micro-architecture. Furthermore, it would reveal many bugs early on.

Furthermore, once a model for the micro-architecture exists, it can be validated against the architecture (say, ISA), using the full range of validation techniques, such as constrained-random stimulus generation, coverage-driven simulation, formal state-space analysis, assertion-based verification, and the like. It is to be expected that the main vehicle of validation at this level at this level would be dynamic techniques, just as dynamic validation is the main vehicle of validation of the RTL. (There is no reason at the moment to expect a ratio between the formal verification and dynamic validation efforts that is higher than the 10:90 ration for the P4, where formal verification has been used rather extensively [4]). Simulation at the micro-architectural level can be 2-3 orders of magnitude faster than RTL simulation. Still, formal methods ought to be an important component in the verification of micro-architectural models.

This paper calls for the development of formal techniques, including formal property verification, formal equivalence verification, automated test generation, and the like to be applied at the micro-architectural level. The formal-techniques community has made significant progress over the last few years in software verification; see [2]. In fact, there are certain abstraction techniques (e.g., uninterpreted functions) that are best applied at the micro-architectural level, rather than the RTL, since that level is too low [6]. The challenge today is to develop formal tools for micro-architectural model.

The micro-architectural level is the right level to start the validation effort. Designers tend to think at the architectural or micro-architectural level when debugging the design, not in terms of lines of RTL. The most obvious notion in a CPU design is that of an instruction. Which instructions are in which pipeline stages is critical to understanding whether the design behaves in a correct manner. More generally, coverage at this level means having a clear picture of what the processor is doing. This is where a temporal assertion language becomes handy [13, 14]. The most promising usage of temporal assertions is at the micro-architectural level. The micro-architectural validators would want to define many temporal assertions corresponding to interesting micro-architecture situations – for example, if a certain buffer is full, if some state machine is in an interesting state, and so forth. They would then query this "database" of coverage events with sophisticated requests [33]. For instance, what was the state machine for the Instruction Fetch module in state A when an interrupt occurred? Furthermore, under the previous scenario, was the interrupted instruction aligned on a 32-bit boundary? Did that happen when some other buffer in the LoadStore unit was full? Each of these properties can be expressed fairly simply using a temporal assertion language [13]. The right level for such an analysis is the micro-architectural level. On the other hand, an extensive micro-architectural validation effort would also benefit the RTL validation effort, as one would expect significant re-use of test vectors and event coverage.

As noted above, there is no hope today of fully formally establishing the equivalence of the RTL with the micro-architectural model. (In contrast, it may be feasible to synthesize parts of the micro-architectural model into RTL.) The RTLers, however, would benefit tremendously from having a micro-architectural model as the starting point of their implementation effort. They would benefit further from starting with a micro-architecture that has already undergone a significant validation effort and is much cleaner from bugs. When questions arise about the correct behavior of RTL, many of them would be resolvable by consulting the micro-architectural model. The RTL validators would benefit from the earlier development of a test environment for the micro-architectural model. Many of the assumptions/assertions written for the micro-architecture validation could be just translated into RTL checkers, facilitating greatly the development of the test environment. (For example, Synopsys says as follows: "Many designers create reference models in SystemC prior to coding RTL. Vera enables the use of a single, golden testbench to drive both SystemC and RTL representations of a design, ensuring consistency between the transaction-level model and the detailed implementation. Vera's transaction-level interface to SystemC enables users to quickly and easily connect a Vera testbench to a SystemC model and then re-use the same testbench when RTL is available.")

The RTL implementation effort should be able to focus on implementing the micro-architecture, assuming that it is mostly validated. When micro-architectural bugs are discovered by the RTL validators, they should be corrected and the corrections should be validated at the micro-architectural level, by the micro-architects and the micro-architectural validators. Keeping the micro-architectural model and the RTL synchronized should not be viewed as a burden, but as the right way to view the design process. "Separation of concerns" is a major principle in design engineering. The micro-architectural model should be concerned with algorithmics, while the RTL should be concerned with its implementation. The micro-architectural model should be kept current as long as it is valuable to do so. It is possible that this model would be abandoned at some point, when the focus shifts fully to the RTL. It is also possible, however, that the micro-architectural model would be kept current throughout the design process, as this model itself has significant IP value. The micro-architectural model, rather than the RTL, is the model that can be re-used; design reuse can be more easily realized at that level than at the RTL.

In summary, developing a micro-architectural model can lead to reduced development costs and increased design quality even without requiring full formal equivalence between the micro-architectural model and the RTL. The key is to

focus a major part of the validation effort at the micro-architectural level. The tightness of the compatibility between that level and the RTL can evolve with verification technology.

2. WHAT LANGUAGE?

When contemplating a shift in methodology as advocated above, a major issue is the selection of the modeling language. What is required is a formalism that is expressive enough and enables both formal and dynamic verification. On the research side, various high-level modeling languages have been developed over the last decade, such as ACL2 [22], PVS [30], TLA [27], and Uclid [26], but none of these language is ready for serious industrial usage. Thus, it'd be better to adopt an industry-standard language such as SystemVerilog or SystemC. Unlike academic languages, industry standard languages will be more readily accepted by micro-architects and designers. Furthermore, EDA support for SystemC and SystemVerilog is increasing daily. In the long run, the industry benefits by using standardized languages.

The advantage of using SystemVerilog is the single point entry to both validation and design, avoiding the pain that accompanies language fragmentation. SystemVerilog evolved Verilog into an HDVL–Hardware Design and Verification Language–supporting a comprehensive verification environment [35]. Nevertheless, it is not clear that SystemVerilog is sufficiently high level to be used for micro-architectural specification. There is a risk that modeling in SystemVerilog would result in a too-low-level model, negating many of the benefits of micro-architectural modeling. Higher-level languages are Esterel [5] and BlueSpec [11], but the former never proved to be highly popular, while the latter is too new. On the other hand, SystemC has emerged lately as the leading language for system-level models, specifically targeted at architectural, algorithmic, transaction-level modeling [18]. It is by now a de facto, open (`www.systemc.org`), industry-standard modeling language, enabling a wide range of modeling levels, from RTL to system level. Its increasing acceptance is driven by the increasing complexity of designs, pushing designers to higher and higher levels of abstractions. The rest of this paper is focused, therefore, on using SystemC as a language for micro-architectural specification, focusing on the issue of formal verification.

3. FORMAL TECHNIQUES FOR SYSTEMC MODELS

While a major goal of SystemC is to enable modeling and verification at higher level of abstraction, enabling early exploration of system-level designs, the focus in the literature so far has been mainly on traditional dynamic validation techniques [23, 34]. It is fair to say that the development of formal-verification techniques for SystemC models is at its infancy. In spite of intensive recent activity in the development of formal-verification techniques for software, extending such techniques to SystemC is a formidable challenge. The difficulty stems from both the object-oriented nature of SystemC, which is fundamental to its modeling philosophy, and its sophisticated event-driven simulation semantics.

3.1 What is SystemC

SystemC is a system-level modeling language based on C++. It uses heavily not only the imperative features of C++, but also its object-oriented features: classes (including abstract classes and class templates), objects, methods (including virtual methods), and inheritance (including multiple inheritance). SystemC uses the object-oriented approach to achieve abstraction, modularity, compositionality, and reuse. The object-oriented paradigm in SystemC is not incidental, but central. It distinguishes SystemC from other modeling languages, such as SpecC [16].

The base layer of SystemC provides an event-driven simulation kernel. This kernel operates at the event level and switches execution between processes. The basic building block in SystemC is the module. A module is a container that contains one or more processes to describe the parallel behavior of the design. A module can also contain other modules, representing the hierarchical nature of the design. Processes execute concurrently; the code within each process executes sequentially. (SystemC has three kinds of processes: SC_METHOD, SC_THREAD, and SC_CTHREAD (clocked threads). Roughly speaking, SC_METHOD corresponds to synthesizable HDL, while SC_THREAD and SC_CTHREAD corresponds to behavioral HDL.) The execution of the processes is driven by the simulation kernel. A complete SystemC model includes the design and the test environment. After compilation the result is an executable file that simulates the design in the provided test environment.

Processes inside a module communicates via signals. Modules communicates via channels. The channels are abstract and are accessed via their interface methods. Modules have ports that are bound to interface methods. The simulation kernel, together with modules, ports, processes, events, channels, and interfaces constitute the core language of C++ This is accompanied by a collection of data types, such as 4-valued logic and vectors, bit and bit vectors, fixed-point numbers, arbitrary precision numbers, and used-defined types. Over this core, SystemC provides many library-defined elementary channels, such as signals, FIFOs, semaphore, Mutex, and the like. On top of this are defined more sophisticated libraries, including master/slave library, process networks, and the like. A transaction-level modeling library (TLM 1.0) was announced in 2005. SystemC has been developed with heavy intermodule communication in mind.

The semantics of SystemC combined the semantics of C++ with the simulation semantics of the kernel. The latter is highly nontrivial, as it has to take into account the combination of "microsteps" and "macrosteps" (which combine evaluation of variables and signals and their update). On one hand, the simulation semantics is event driven rather than cycle driven. At the same time, SystemC has a discrete model of time (the default time resolution is one picosecond), which means that it also has cycle-level semantics. A formalization of SystemC's simulation semantic using Distributed Abstract State Machines is provided in [29]. See also [15] for work on the relationship between SystemC and Abstract States Machines. It is fair to say, however, that SystemC does not have a fully formal semantics, which poses a challenge to the development of formal techniques for SystemC.

3.2 Verification of SystemC Designs

The "workhorse" of SystemC validation is dynamic validation. SystemC models are meant to be simulated. The SystemC Verification Standard provides API for transaction-

based verification, constrained and weighted randomization, exception handling, and other verification tasks [23, 34]. The standard also includes HDL-connection APIs, to enables SystemC testbenches to be used for Verilog or VHDL designs. This means that testbenches developed for high-level models can be reused after the high-level models have been refined into RTL.

This paper calls for the development of formal techniques to augment standard SystemC verification. Let us consider various formal techniques.

- ASSERTION-BASED VALIDATION: In assertion-based validation [14] one writes properties in a formal language, e.g., PSL [13] or SVA [36]. The simulation engine then monitors these properties during the simulation. For example, Intel reported recently of its Fedex tool [1], which is used to monitor assertions written in the formal language ForSpec. There are no nontrivial technical barriers to incorporating assertion-based validation in SystemC dynamic validation [20]. This requires also integrating a BDD package into SystemC; see [12] for a report of such integration. Extending assertion-based verification to SystemC would mean that the *same* assertions can be used in a SystemC environment and in a RTL environment. As argued earlier, such reuse ought to be an important component of the overall design-verification effort.

- EXPLICIT-STATE MODEL CHECKING: The distance between assertion-based dynamic validation and assertion-based explicit-state model checking is not very large conceptually. In dynamic validation we generates a set of test runs and monitors the assertions along them. In explicit-state model checking we exercise the design exhaustively – by keeping track of all nondeterministic choices (e.g., input values), we ensure that all of them get exercises. There is also a need to monitor the program states visited, to ensure termination of the search process [10]. Model checkers such as the Java Pathfinder [21], accomplish by rewriting the execution engine (JVM). There is no significant technical barrier to extending explicit-state model checking to SystemC, but the implementation effort would be nontrivial, as the simulation kernel would need to be modified to exercise the design exhaustively.

  The real limitation of explicit-state model checking is the state explosion problem [10]. To deal with large state spaces we need to introduce abstraction technique, but automating such techniques generally requires the use of symbolic model checking (BDD or SAT-based) [9].

- SYMBOLIC SIMULATION: In symbolic simulation we execute the program in an abstract setting, using symbols, rather than concrete values for variable. Each symbolic simulation path represents a whole class of possible program executions. By heaving a symbolic representation of this class of executions, we can reason about it symbolically, generate test cases and more [25]. Recent progress in symbolic simulation for Java [31], suggests that this technique might be applicable to SystemC. One had to recall, however, that SystemC is aimed at concurrent systems, while Java is aimed more an sequential systems. This is a worthwhile direction of research. Perhaps a bit less ambitious is the techniques of statically analyzing dynamic execution paths in programs [8]. This technique has recently been quite successful in software development and might be adaptable to SystemC.

- SYMBOLIC MODEL CHECKING: Symbolic model checking goes a step further in verifying temporal properties of designs; instead of searching the state space explicitly, it is represented and searched by means of symbolic reasoning [10]. Initial progress in applying symbolic model checking to SystemC models is reported in [19]. The difficulty of extending symbolic model checking to SystemC is that symbolic model checking requires that we have formal semantics that describes the transition relation of the design. This is quite nontrivial for SystemC due to the heavy use of object-oriented machinery and the fairly involved simulation semantics. Extending recent progress on software model checking [2], which is typically aimed at C programs, to SystemC is a worthwhile research project. Recent progress at extending symbolic reasoning techniques to object-oriented languages such as Java [7] and Spec# [3] should be built upon.

- EQUIVALENCE VERIFICATION: A more ambitious goal is the development of techniques to formal verify the equivalence of SystemC models and RTL models, analogously to current technology for formally establishing equivalence of RTL models with Netlist models [24]. This is a significant research challenge. A more modest goal would be to establish looser notions of equivalence–referred to earlier as "compatibility", "conformance", or"compliance"–between SystemC models and RTL models. For example, we mentioned earlier that at the SystemC level it is quite natural to specify directly interesting micro-architectural events. Compatibility between SystemC and RTL models requires than that such micro-architectural events be also monitored at the RTL.

4. REFERENCES

[1] R. Armoni, D. Korchemny, A. Tiemeyer, and M. V. Y. Zbar. Deterministic dynamic monitors for linear-time assertions. In *Proc. Workshop on Formal Approaches to Testing and Runtime Verification*, volume 4262 of *Lecture Notes in Computer Science*. Springer, 2006.

[2] T. Ball, B. Cook, V. Levin, and S. Rajamani. SLAM and static driver verifier: Technology transfer of formal methods inside microsoft. In *Proc. 4th Int'l Conf. on Integrated Formal Methods*, Lecture Notes in Computer Science 2999, pages 1–20, 2004.

[3] M. Barnett, R. DeLine, M. Fähndrich, K. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *J. of Object Technology*, 3(6), 2004.

[4] B. Bentley. High level validation of next-generation microprocessors. In *Proc. IEEE Int'l Workshop on High Level Design Validation and Test*, pages 31–35, 2002.

[5] G. Berry and G. Gonthier. The esterel synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[6] J. Burch and D. Dill. Automatic verification of pipelined microprocessor control. In *Proc. 6th Int'l Conf. on Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 68–80. Springer, 1994.

[7] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. R. Kiniry, G. Leavens, K. Leino, and E. Poll. An overview of JML tools and applications. *Int'l J. on Software Tools for Technology Transfer*, 7(3), 2005.

[8] W. Bush, J. D. Pincus, and D. Sielaff. A static analyzer for finding dynamic programming errors. *Software Practice and Experience*, 30(7):775–802, 2000.

[9] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proc 12th Int. Conf. on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2000.

[10] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[11] N. Dave. Designing a reorder buffer in Bluespec. In *Proc. 2nd ACM/IEEE Int'l Conf. on Formal Methods and Models for Co-Design*, pages 93–102, 2004.

[12] R. Drechsler and D. Große. Reachability analysis for formal verification of SystemC. In *Proc. Euromicro Symp. on Digital System Design*, pages 337–340, 2002.

[13] C. Eisner and D. Fisman. *A Practical Introduction to PSL*. Springer, 2006.

[14] H. Foster and D. Lacey. *Assertion-Based Design*. Kluwer, 2003.

[15] A. Gawanmeh, A. Habibi, and S. Tahar. Enabling SystemC verification using abstract state machines. In *Proc. Languages for Formal Specification and Verification, Forum on Specification and Design Languages*, 2004.

[16] A. Gerstlauer, R. Dömer, J. Peng, and D. Gajski. *System Design – A Practical Guide with SpecC*. Springer, 2001.

[17] R. Goering. A call to action for the EDA industry. *EETimes*, June 2005.

[18] T. Groetker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Springer, 2002.

[19] D. Große and R. Drechsler. Formal verification of LTL formulas for SystemC designs. In *Proc. Int'l Symposium on Circuits and Systems*, pages 245–248, 2003.

[20] D. Große and R. Drechsler. Checkers for SystemC designs. In *Proc. 2nd ACM/IEEE Int'l Conf. on Formal Methods and Models for Co-Design*, pages 171–178, 2004.

[21] K. Havelund and T. Pressburger. Model checking JAVA programs using JAVA pathfinder. *Int'l J. on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

[22] W. Hunt and J. Sawada. Verifying the FM9801 microarchitecture. *IEEE Micro*, 19(3):47–55, 1999.

[23] C. Ip and S. Swan. A tutorial introduction on the new SystemC verification standard. Technical report, www.systemc.org, 2003. White Paper.

[24] Z. Khasidashvili, M. Skaba, D. Kaiss, and Z. Hanna. Theoretical framework for compositional sequential hardware equivalence verification in presence of design constraints. In *Proc. Int'l Conf. on Computer-Aided Design*, pages 58–65. IEEE Computer Society / ACM, 2004.

[25] J. King. Symbolic execution and program testing. *Comm. ACM*, 19(7):385–394, 1976.

[26] S. Lahiri, S. Seshia, and R. Bryant. Modeling and verification of out-of-order microprocessors in UCLID. In *Proc. 4th Int'l Conf. on Formal Methods in Computer-Aided Design*, volume 2517 of *Lecture Notes in Computer Science*, pages 142–159. Springer, 2002.

[27] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[28] J. Monaco, D. Holloway, and R. Raina. Functional verification methodology for the PowerPC 604 microprocessor. In *Proc. 33rd Design Automation Conference*, pages 319–324, 1996.

[29] W. Mueller, J. Ruf, D. Hoffmann, J. Gerlach, T. Kropf, W. Rosenstiehl, and W. Mueller. The simulation semantics of systemc. In *Proc. Conf. on Design, Automation and Test in Europe*, pages 64–70. IEEE Press, 2001.

[30] S. Owre, J. Rushby, and N. Shankar. Pvs: A prototype verification system. In *Proc. 11th Int'l Conf. on Automated Deduction*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752. Springer, 1992.

[31] C. Pasareanu and W. Visser. Verification of Java programs using symbolic execution and invariant generation. In *Proc. 11th Int'l SPIN SWorkshop on Model Checking Software*, volume 2989 of *Lecture Notes in Computer Science*, pages 164–181. Springer, 2004.

[32] T. Ramanathan and V. Thomas. Platform 2015: Intel processor and platform evolution for the next decade. Technical report, Intel, 2005. White Paper, Platform 2015.

[33] A. Raynaud. Code coverage techniques – a hands-on view. *EETimes*, February 2003.

[34] L. Singh and L. Drucker. *Advanced Verification Techniques : A SystemC Based Approach for Successful Tapeout*. Springer, 2004.

[35] S. Sutherland, S. Davidmann, and P. Flake. *SystemVerilog For Design: A Guide to Using SystemVerilog for Hardware Design and Modeling*. Springer, 2003.

[36] S. Vijayaraghavan and M. Ramanathan. *A Practical Guide for SystemVerilog Assertions*. Springer, 2005.