The Challenge of Hardware-Software Co-Verification*

Panagiotis Manolios

College of Computing
Georgia Institute of Technology
Atlanta, GA 30318
manolios@cc.gatech.edu

Abstract. Building verified computing systems such as a verified compiler or operating system will require both software and hardware verification. How can we decompose such verification efforts into mostly separate tasks, one involving hardware and the other software? What theorems should we prove? What specification languages should we use? What tools should we build? To what extent can the process be automated? We address these issues, using as a running example our recent and on-going work on refinement-based pipelined machine verification.

1 Introduction

The ultimate goal of the formal verification community is to mechanically verify computing systems from the subatomic level up to high-level specifications. In principle, we know that this is possible. It is possible to describe the standard model, quantum mechanics, string theory, and, in general, whatever physical or computational theory we desire, using first-order logic.

However, it does not currently seem feasible to do this: the human effort required is daunting. The differences in size and speed between the subatomic level and higher-level subsystems such as disk arrays are astronomical. In addition, the subatomic level is inherently continuous and probabilistic; in fact, current semiconductor devices not only depend on quantum effects, but even take advantage of them. On the other hand, higher-level abstractions tend to be discrete and (non)deterministic.

The main focus of this paper is on hardware-software co-verification, a central part of the verification challenge which exhibits many of the characteristics of the general problem, *e.g.*, it spans multiple abstraction levels. Hardware verification has been an active area of research for the last few decades and software verification is currently receiving renewed attention. Eventually, these now mostly disparate fields will have to be combined, if we are to truly verify computing systems. It is not just that it is desirable to have a verified hardware base for our software; many challenge problems, *e.g.*, building a verified compiler [7] or operating system, inherently involve both software and hardware.

In the remainder of this paper, we briefly expand upon some of the issues that arise in extending current work on hardware verification to enable software verification. Our viewpoints are shaped by our recent and on-going work on automating proofs of correctness for pipelined machines, which we use as a running example.

* This work was funded in part by NSF grants CCF-0429924, IIS-0417413, and CCF-0438871.

We start in Section 2 by considering what Dijkstra called the "pleasantness" problem: what theorems should we prove? We outline a theory of refinement, our answer to the pleasantness problem as it pertains to pipelined machine verification, in Section 3. We then look at pragmatics such as: what specification language to use (Section 4) and how to automate (Section 5) and evaluate (Section 6) the results. In Section 7, we discuss some of our recent work on hardware-software co-verification. This work has led us to start developing tools, a topic discussed in Section 8. We conclude in Section 9.

2 The Pleasantness Problem

One of the major challenges in verification is what Dijkstra called the "pleasantness" problem [5]: how do we determine the "right" theorems to prove? For example, what theorems establish that a device driver works correctly? Well, it depends, but it is worth noting that the pleasantness problem can be mitigated by good design. It is also worth noting that many problems are inherently complex. For example, what does it mean for floating-point arithmetic to be correct? It took many years to settle on the IEEE floating-point arithmetic standards 754 and 854, and William Kahan was awarded the Turing award for his contributions to this effort.

Let us consider the pleasantness problem in the context of pipelined machine verification: what set of properties establishes that a pipelined machine behaves correctly? Such a set might include a property that describes the behavior of the branch misprediction logic. This property might specify what should happen during a branch mispredict: what instructions are invalidated, what latches are affected, how the program counter is updated, etc. The problem with this approach is that it is not clear when one has "complete coverage," which leaves open the possibility that erroneous corner cases remain. Another problem involves the maintenance of such properties, as any design changes will necessitate an update of the properties. Designs will undergo numerous changes, making the tracking of such correctness properties problematic.

For the above reasons, we use a correctness criterion based on refinement that tackles the pleasantness problem by taking advantage of the instruction set architecture interface. This leads to a notion of correctness that is not affected by changes to the pipelined machine. The idea is to show that, to an external observer, the implementation behaves in a fashion that is consistent with the specification, the much simpler instruction set architecture. The instruction set architecture is arguably the most important interface in computer science. On the one hand, it has allowed programmers to think in terms of a machine that executes one instruction at a time. On the other, it has allowed hardware designers to build inherently parallel machines, with features such as superscalar execution and deep pipelines, which simultaneously process numerous instructions at various stages of completion.

There is still the question of what kind of refinement theorem to prove and, more generally, the question of what correctness statements constitute a good answer to the pleasantness problem. An important property of such correctness statements is that, once established, they enable us to ignore the internals of the system under consideration in subsequent verification efforts. For example, a suitable notion of correctness for pipelined machines would allow us to reduce the proof that software running on a

pipelined machine satisfies its specification, to a proof that the software runs correctly on the instruction set architecture. To actually achieve this decomposition requires a notion of correctness that preserves not only safety properties, but also liveness properties. To see why, suppose that we have a proof of correctness of the pipelined machine which does not preserve liveness properties. Now, consider proving that a simple program, say to sort an array of numbers, is correct. This requires a total correctness proof at the instruction set architecture level. But, it also requires taking the details of the pipelined machine into account in order to establish that no deadlock or livelock occurs for any execution of this particular program. Variants of the well-known Burch and Dill notion of correctness for pipelined machines [4] suffer from this problem [13]. The refinement theorems we prove do not, as they preserve both safety and liveness properties.

It is especially important to prove theorems that encapsulate the behavior of systems when many layers of abstraction are involved, as otherwise, the verification problem becomes unmanageable. An early pioneering body of work on the use of theorem proving to verify systems from the netlist level up to a high-level language is the CLI stack [24].

Finally, we briefly discuss performance and dependability properties, considered by many to be beyond the reach of formal verification. For example, how do we *prove* that a microprocessor performs well? The problem with this question is that it is vague, *not* that it is beyond the reach of formal methods. This is really just an instance of the pleasantness problem. The best known methods of making the performance question precise depend on the use of benchmarks, sets of programs meant to be "representative" of the kinds of applications the microprocessor will be used to run. Microprocessor performance is then measured with respect to the benchmarks. Performance is now very easy to reason about formally: just execute the model of the microprocessor on the benchmarks and keep track of the time. Similarly, if we know what is meant by "dependability," then we can analyze dependability properties formally.

3 Refinement-Based Verification of Pipelined Machines

In this section we informally review the theory of refinement we use to manage the pleasantness problem in the context of our work on pipelined machine verification; for a full account see [14, 15]. A theory of refinement defines when a concrete implementation refines (implements) an abstract specification. In applying refinement to pipelined machine verification, the idea is to show that MA, a machine modeled at the microarchitecture level, a low level description that includes the pipeline, refines ISA, a machine modeled at the instruction set architecture level. A refinement proof is relative to a *refinement map*, r, a function from MA states to ISA states. The refinement map, r, shows us how to view an MA state as an ISA state, *e.g.*, the refinement map has to hide the MA components (such as the pipeline) that do not appear in the ISA. That MA refines ISA means that for every pair of states w, s such that w is an MA state and $s = r(w)$, we have that for every infinite path σ starting at s, there is a "matching" infinite path δ starting at w, and conversely. That σ and δ "match" implies that applying r to the states in δ results in a sequence that is equivalent to σ up to finite stuttering (repetition of states). This notion of refinement is based on stuttering bisimulation and implies that related states satisfy the same next-time-free temporal logic formulas (*e.g.*, $\mathrm{CTL}^* \setminus \mathsf{X}$) [2].

Stuttering is a common phenomenon when comparing systems at different levels of abstraction, *e.g.*, if the pipeline is empty, `MA` will require several steps to complete an instruction, whereas `ISA` completes an instruction during every step. We note that stuttering bisimulation differs from weak bisimulation [23] in that weak bisimulation allows infinite stuttering. Distinguishing between infinite and finite stuttering is important, because (among other things) we want to distinguish deadlock (which usually indicates an error) from stutter.

The above formulation of refinement requires reasoning about infinite paths, something that is difficult to automate [25]. WEB-refinement is an equivalent formulation that can be more readily verified mechanically, as it only requires local reasoning involving `MA` states, the `ISA` states they map to under the refinement map, and their successor states [14]. WEB-refinement is a generally applicable notion. However, since it is based on bisimulation, it is often too strong a notion and in this case refinement based on stuttering *simulation* should be used (see [14, 15]).

An important feature of our theory of refinement is that it is compositional. This allows us to verify machines in stages: if M refines M', which refines M'', then M refines M'' (with respect to the composition of the refinement maps).

We have been pleasantly surprised by how many opportunities there have been to exploit the generality of our theory of refinement. For example, that the refinement map used is just a parameter of our theory has enabled us to explore alternative refinement maps, some of which led to orders of magnitude improvements in verification times [21]. That our theory is compositional has allowed us to verify complex machines one feature at a time, making it possible to obtain tremendous savings in terms of verification times and in terms of the complexity of counterexamples when errors are discovered [18].

4 Specification Languages

Having addressed what to prove, we next consider what specification language to use. The available specification languages are quite varied, with foundations ranging from higher-order logics, to first-order logics, to constructive type theory, to decidable fragments of various logics, to temporal logics, etc. The main issues are not so much issues of fundamental power, rather they are about expressiveness and convenience. A good analogy is the situation in programming languages, where languages are judged on their ability to effectively describe computational processes, not on their fundamental power, as many simple languages encompass all that can be effectively computed. Similarly, most of mathematics can be embedded in first order logic, say ZFC, and since our focus is on *mechanical* verification, any proof theory where the notion of proof is decidable can be easily handled in a simple first-order setting.

The connection between programming languages and specification languages is deeper than the analogies above indicate. The systems to be verified are written in a particular programming language. To verify such systems, we must be able to embed the programming language in our specification language. In fact, some specification languages are just extensions of a programming language, *e.g.*, ACL2 [9, 11] can be thought of in this way, as it allows any ACL2 program to be used as part of a specifica-

tion. This makes it simpler for a single person to both write code and be involved in the verification process, something that we expect will eventually become routine practice. We also expect that new languages will eventually be developed with verification as a first-class concern: they will have formal semantics, a proof theory, various libraries and APIs providing basic verification functionality, proof checkers, theorem provers, verified modules and libraries, etc. In fact, now seems like a good time to create such a language, something that will require researchers with expertise in programming languages and verification.

In our work, we found it important to have a general-purpose specification language that allows us to clearly state the theorems of interest, that allows us to efficiently execute and test models, that has structuring mechanisms to manage large scale verification efforts, and that has existing libraries of theories. It is also important that the theorem proving engines used are highly efficient. This topic is discussed in more detail in the next section. We did not find a single tool that suited all our needs and decided to integrate UCLID with ACL2, as we discuss in Section 8.

5 Automation

A major verification challenge concerns automation. While it is not possible to build a general system which given a theorem produces a proof, it is possible to build, tune, or extend systems so that they can be used in a highly automated fashion on a sufficiently restricted class of problems.

As an example, we consider our experiences with pipelined machine verification. Applying our refinement theorem requires, among other considerations, the construction of a suitable refinement map and well-founded rank functions. While there is no general recipe for doing this, we have been exploring how to automate these constructions in the context of pipelined machine verification. The main idea is to discover widely applicable schemes that can be easily (even mechanically) specialized for the particular machine in question. The commitment refinement map [13, 14] is an example of a refinement map that can be easily specialized for particular machines. This map produces an instruction set architecture state from a pipelined machine state by simply invalidating all the partially executed instructions and projecting out the instruction set architecture components. An inductive invariant is required, but here, too, a general scheme can be given: a state satisfies the invariant if, after invalidating all of its partially executed instructions, we can reach an equivalent state within a fixed number of steps determined by the pipeline length. Finally, a rank function is needed and, again, a general scheme is used that gives the number of steps the pipelined machine must take before it changes state visible at the instruction set architecture level.

The next step is to simplify the statement of the refinement theorem. Here too, we can specialize and simplify matters by strengthening the main refinement proof obligation. The result is a formula expressible in the CLU logic [3], which can be decided by the UCLID tool [12]. The major restriction is that the models we use are at the *term level*: they abstract away the datapath, require the use of numerous abstractions, implement a small subset of the instruction set, and are far from executable.

Using the WEB-refinement framework as described up to this point has allowed us to significantly extend what can be done in a highly automated fashion. For example, a big advantage over previous work is that we can handle liveness; in fact, we show that with our approach the time spent proving liveness accounts for only 5% of the total verification time [16].

We discovered that the refinement maps used for pipelined machine verification can have a drastic impact on verification times. This led to the introduction of a new method of defining the commitment refinement map which gives a 30-fold improvement in verification times over the standard flushing and commitment refinement maps [19]. We also discovered a new class of refinement maps, that partly commit and partly flush, that can provide several orders of magnitude improvements in verification times over pure flushing or pure commitment refinement maps [21].

All of the above work can be automated. In fact, we have a Web-based tool for generating complex pipelined machine models, including the correctness statements [20].

We end this section with a final example showing how to leverage the compositional nature of our theory of refinement. We developed a set of convenient, easily-applicable, and complete compositional proof rules and showed how this allows us to greatly extend the applicability of decision procedures by verifying a complex, deeply pipelined machine that state-of-the-art tools cannot currently handle. Our approach allows us to reduce the previous monolithic approaches to pipelined machine verification into a sequence of much simpler refinement steps. Not only are there benefits in terms of verification times, but even counterexamples are generally much simpler [18].

6 Evaluation

Any verification effort will invariably require many decisions, including which specification language to use, what theorems to prove, what theory to develop, etc. In this section, we make some observations about the evaluation of such efforts and advocate the use of end-to-end evaluations.

We start with a list of basic evaluation questions. First, what was mechanically verified and at what level? For example, there is a big gap between the trivial proof that an abstract floating-point adder is correct and a proof that a netlist description of the floating-point adder in a current microprocessor is correct. The devil *is* in the details. In addition, it often seems that work which depends on special paper and pencil "meta" theorems is valued more than work which develops such theorems inside a formal framework. But, if the point is to mechanically verify as much as possible, the latter approach should be preferred, even if it was not "automatic."

Another basic question is: how much human effort was required? Measuring this can be subtle, but the practice of classifying methods as either being "fully" automatic or not is counterproductive. For example, we have found that aspects of our work that are considered "automatic" by the research community (*e.g.*, defining refinement maps) have taken far longer than aspects that are not considered automatic (*e.g.*, defining invariants). One should account for all user time, including the time to define and formalize the problem and to determine and mechanically verify the theorems constituting correctness.

Claims by authors should be backed up with enough data to replicate the work reported. This is a basic rule of science that is too often ignored. If there is a good reason why the data cannot be released, then every effort to release a sanitized version of the reported work should be made. When we co-edited a book on the applications of ACL2 [8], we required every submission to include ACL2 proof scripts justifying every formal claim made. This material is available on the Web [10]. For example, Russinoff and Flatau used ACL2 to verify several of the floating-point arithmetic operations in the AMD Athlon processor. Obviously, AMD did not want to release these proof scripts, as they contain Athlon floating-point designs. However, the authors were able to release a precise description of their RTL language and the library of theorems used. They also defined, verified, and released a sanitized version of the floating-point multiplier [26]. Others researchers who could neither release their proof scripts nor produce sanitized versions of their work were not able to contribute.

As a general principle, the evaluation of verification efforts should focus on end-to-end arguments. By this we mean that the stated contribution should be related to the larger context in which the verification is taking place. For example, consider a paper that shows how abstraction α leads to faster verification times than abstraction β. However, if abstraction α is harder to mechanically justify than abstraction β, then from an end-to-end perspective, the use of abstraction α is a net loss. As another example, a negative end-to-end evaluation of a method that provides increased automation is possible under various scenarios, *e.g.*, the method may require a complex preprocessing step, or may generate counterexamples that are hard to understand, or may require extensive tool support, etc.

The end-to-end evaluation should also consider how the work can be used in the context of long-term verification. For example, there are often one-time verification costs such as embedding the semantics of some language into a theorem prover or developing a library of theorems applicable to a wide class of related problems. The floating-point work we mentioned previously is a good example, as these one-time costs were leveraged in subsequent verification efforts, leading to drastic reductions in manual effort required to verify subsequent floating-point operations.

7 Hardware Verification that Enables Software Verification

Recently, we have started thinking about how to prove that low-level programs executing on a pipelined machine behave correctly. The idea is to use WEB-refinement to prove that software running on a pipelined machine satisfies its specification by first proving that the pipelined machine refines the instruction set architecture and then showing that the software running on the instruction set architecture satisfies its specification. But, this requires the use of executable pipelined machine models, because the correctness of software depends on the semantics of instructions. However, in order to take advantage of decision procedures, previous work on hardware verification has focused on term level models that abstract away the datapath, require the use of numerous abstractions, implement a small subset of the instruction set, and are far from executable. To bridge the gap between term level models and bit-level, executable mod-

els is a major challenge, requiring that all of the abstractions employed in term level modeling are mechanically justified. We now briefly discuss the issues.

First, term-level models abstract away the datapath, hiding much of the real complexity in an executable model. For example, decoding is modeled using a set of uninterpreted functions. However, decoders for bit-level machines are complicated and notoriously difficult to get right in modern designs.

Even the ALU is modeled using uninterpreted functions, but to prove theorems about software, we need a model of the machine in which the ALU is interpreted.

Another form of abstraction concerns the instruction set itself, which is abstracted away by only modeling one instruction per instruction class. But, again, we really need a model with the full instruction set in order to verify software.

The refinement theorem cannot be expressed in UCLID. Instead, we check what we call the "core theorem," whose proof accounts for most of the verification time. The core theorem requires "polluting" both the pipelined machine and the instruction set architecture by adding extra inputs, control logic, and state to control when and how the refinement map is applied, among other things. This is quite complicated and it is easy to introduce errors, as we have often discovered. A proof is required to show that refinement proofs based on polluted models imply refinement proofs for the original models.

As a final example, we consider branch prediction. Branch prediction schemes are sometimes abstracted using an integer to represent the state of the branch predictor and uninterpreted functions which, given the current state of the branch predictor, return the next state and a guess (taken or not) [27]. This seems simple enough, but using this abstraction turns out to be quite cumbersome. Here's why. The branch predictor depends on the program counter, which depends on the program we are executing and if all we have is one integer to represent the state of the branch predictor, we have to use some kind of Gödel encoding scheme to encode the state of the machine with a single integer. The amount of work required to justify the abstraction is more than the savings it provides. Furthermore, if we have an infinite memory, this abstraction is not sound. A much simpler abstraction which is easily justified just makes nondeterministic choices.

We end with two final observations. First, having an efficiently executable pipelined machine can be quite useful in industrial settings, as it makes it possible to have a single "golden" reference model that can be used both for simulation-based testing and for formal verification. For example, Rockwell Collins used ACL2 to develop, test, and validate executable, bit- and cycle-accurate microprocessor models that ran at close to C speeds [6]. Second, as mentioned in Section 2, it is crucial that the notion of refinement used for hardware-software co-verification preserves liveness properties.

8 Tools

Addressing the above issues requires the use of a tool that can describe executable bit-level designs, can reason about total correctness, can manage the proof process, etc. ACL2 or any industrial-strength theorem prover can be used for this purpose. However, specialized decision procedures have the potential to significantly extend what can be handled automatically. For example, in one experiment, a proof that took about 3 seconds with UCLID required $15\frac{1}{2}$ days with ACL2 [17]. We therefore integrated UCLID

with ACL2, and were able to use ACL2 to reduce the proof that an executable, bit-level machine refines its instruction set architecture to a proof that a term level abstraction of the bit-level machine refines the instruction set architecture, which is then handled automatically by UCLID. We also used our system to develop, execute, test, and verify a dynamic programming solution to the Knapsack problem. Thus, we can exploit the strengths of the two systems to prove theorems that are not possible to even state using UCLID and that would require heroic efforts using just ACL2 [22].

An interesting observation is that verification tools have matured to the point where they can handle complex enough subproblems to make the kind of coarse-grained integration described above worthwhile. This allows us to avoid the well-known problems with fine-grained integration [1]. We see many opportunities currently for this kind of tool integration, *e.g.*, we are currently looking at combining static analysis techniques with theorem proving.

9 Conclusions

Building truly reliable systems will require hardware-software co-verification. In this paper we have outlined some of the issues, challenges, and opportunities, using as a running example our recent work on automating refinement proofs involving pipelined machines.

References

1. R. S. Boyer and J. S. Moore. Integrating decision procedures into heuristic theorem provers: a case study of linear arithmetic. In *Machine intelligence 11*, pages 83–124. Oxford University Press, Inc., 1988.
2. M. Browne, E. M. Clarke, and O. Grumberg. Characterizing finite Kripke structures in propositional temporal logic. *Theoretical Computer Science*, 59, 1988.
3. R. E. Bryant, S. K. Lahiri, and S. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In E. Brinksma and K. Larsen, editors, *Computer-Aided Verification–CAV 2002*, volume 2404 of *LNCS*, pages 78–92. Springer-Verlag, 2002.
4. J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In *Computer-Aided Verification (CAV '94)*, volume 818 of *LNCS*, pages 68–80. Springer-Verlag, 1994.
5. E. W. Dijkstra. Science fiction and science reality in computing, 1986. EWD 952. See URL `http://www.cs.utexas.edu/users/EWD`.
6. D. Greve, M. Wilding, and D. Hardin. High-speed, analyzable simulators. In Kaufmann et al. [8], pages 113–135.
7. T. Hoare. The verifying compiler: A grand challenge for computing research. *J. ACM*, 50(1):63–69, 2003.
8. M. Kaufmann, P. Manolios, and J. S. Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishers, June 2000.
9. M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, July 2000.

10. M. Kaufmann, P. Manolios, and J. S. Moore. Supporting files for "Computer-Aided Reasoning: ACL2 Case Studies". See the link from URL `http://www.cs.utexas.edu/-users/moore/acl2`, 2000.
11. M. Kaufmann and J. S. Moore. ACL2 homepage. See URL `http://www.cs.-utexas.edu/users/moore/acl2`.
12. S. Lahiri, S. Seshia, and R. Bryant. Modeling and verification of out-of-order microprocessors using UCLID. In *Formal Methods in Computer-Aided Design (FMCAD'02)*, volume 2517 of *LNCS*, pages 142–159. Springer-Verlag, 2002.
13. P. Manolios. Correctness of pipelined machines. In W. A. Hunt, Jr. and S. D. Johnson, editors, *Formal Methods in Computer-Aided Design–FMCAD 2000*, volume 1954 of *LNCS*, pages 161–178. Springer-Verlag, 2000.
14. P. Manolios. *Mechanical Verification of Reactive Systems*. PhD thesis, University of Texas at Austin, August 2001. See URL `http://www.cc.gatech.edu/~manolios/-publications.html`.
15. P. Manolios. A compositional theory of refinement for branching time. In D. Geist and E. Tronci, editors, *12th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2003*, volume 2860 of *LNCS*, pages 304–318. Springer-Verlag, 2003.
16. P. Manolios and S. Srinivasan. Automatic verification of safety and liveness for XScale-like processor models using WEB-refinements. In *Design Automation and Test in Europe, DATE'04*, pages 168–175, 2004.
17. P. Manolios and S. Srinivasan. A suite of hard ACL2 theorems arising in refinement-based processor verification. In M. Kaufmann and J. S. Moore, editors, *Fifth International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2004)*, November 2004. See URL `http://www.cs.utexas.edu/users/moore/acl2/workshop-2004/`.
18. P. Manolios and S. Srinivasan. A complete compositional reasoning framework for the efficient verification of pipelined machines. In *ICCAD-2005, International Conference on Computer-Aided Design*, 2005. To appear.
19. P. Manolios and S. Srinivasan. A computationally efficient method based on commitment refinement maps for verifying pipelined machines models. In *ACM-IEEE International Conference on Formal Methods and Models for Codesign*, pages 189–198, 2005.
20. P. Manolios and S. Srinivasan. A parameterized benchmark suite of hard pipelined-machine-verification problems. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, 2005. To appear.
21. P. Manolios and S. Srinivasan. Refinement maps for efficient verification of processor models. In *Design Automation and Test in Europe, DATE'05*, pages 1304–1309, 2005.
22. P. Manolios and S. Srinivasan. Verification of executable pipelined machines with bit-level interfaces. In *ICCAD-2005, International Conference on Computer-Aided Design*, 2005. To appear.
23. R. Milner. *Communication and Concurrency*. Prentice-Hall, 1990.
24. J. S. Moore. Special issue on system verification. *Journal of Automated Reasoning*, 5(4), 1989.
25. K. S. Namjoshi. A simple characterization of stuttering bisimulation. In *17th Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 1346 of *LNCS*, pages 284–296, 1997.
26. D. M. Russinoff and A. Flatau. RTL verification: A floating-point multiplier. In Kaufmann et al. [8], pages 201–231.
27. M. N. Velev and R. E. Bryant. Formal verification of superscalar microprocessors with multicycle functional units, exceptions, and branch prediction. In *Proceedings of the 37th conference on Design Automation*, pages 112–117. ACM Press, 2000.